Reinforcement Learning for Interactive Theorem Proving
Creating an Artificial Student

Cottaar, Jolijn

Award date:
2020

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Department of Mathematics and Computer Science

Reinforcement Learning for Interactive Theorem Proving

Creating an Artificial Student

Jolijn Cottaar

Supervisors:
dr. J.W. Portegies
dr. C. Hojny

Eindhoven, July 2020

Abstract

In this thesis we have created an artificial student, who learns to provide mathematical proofs. The learning is based on the Reinforcement Learning algorithms Sarsa, Q-Learning and Epsilon Soft. The artificial student uses the interactive theorem prover Coq and it is only able to prove intuitionistic propositional logic lemmas.

We laid the theoretical basis using a Markov Decision Process for our student. Subsequently we created a prototype for the artificial student. And finally we have compared the algorithms and various other variables.

We have done limited testing concerning comparing certain aspects of our program. We have seen that the Epsilon Soft algorithm learns most quickly from all the algorithms in the first 15 episodes. Due to runtime problems we are not able to show what happens in episodes later on. Similarly we have found that the Name state space is best at finding equivalent states with changing proposition names. A low epsilon chosen seems to have the effect of increasing the average number of actions in an episode, but also making sure more episodes end in a completed proof, while a high epsilon does the opposite. In reward functions we have found that reward functions which give a positive reward for a completed proof has the best effect on the learning.

Contents

Chapter 1

Introduction

Proving lemmas and theorems is seen as the hardest part of any mathematics course. For first-year math students it is always a shock to the system to have to learn something entirely new that is so hard. Thus it is seen as one of the stumbling blocks of the first year of mathematics.

At the same time professors have little time to spend on education, since the pressure to publish and gain funding is ever increasing. To help both sides of this problem we took the first step in the development of an artificial teacher helping out with teaching students to prove. Additionally we have researched how reinforcement learning can contribute in this area.

In this project we used the interactive theorem prover Coq [1]. In a previous mathematics bachelor project, Beurskens [10] has made the first steps of making Coq more accessible and easier to understand for students. Continuing on this, a group for computer science and mathematics double majors have created the program Waterproof [4] which enables students to create readable and understandable proofs using Coq. This program is already used in courses at the TU/e.

We focused on a subset of mathematics, namely the intuitionistic propositional logic. Intuitionistic propositional logic differs from the more commonly used classic propositional logic on a few points. Intuitionistic logic does not contain the Aristotelian law of excluded middle ($A \vee \neg A$) or the classical law of double negation elimination ($\neg\neg A \rightarrow A$). It does however contain the law of contradiction ($(A \rightarrow B) \rightarrow ((A \rightarrow \neg B \rightarrow \neg A))$ and *ex falso sequitur quodlibet* ($\neg A \rightarrow (A \rightarrow B)$) [16] [15].

The main idea of our project is to create an artificial student which uses Coq to learn how to prove certain lemmas. To mimic the learning of a 'real' student most closely we look at the artificial learning algorithms known as Reinforcement Learning [18]. Reinforcement Learning is mostly based on learning from experience.

The overall goal is to create an artificial teacher, which can help the artificial student and ultimately a real life student. This artificial student is the first step on the road to this.

Parallel to this project we have worked on a prototype artificial teacher for which I refer to the thesis of my colleague-student S. McCarren [14].

Literature Study and Earlier Work

There has been considerable work done in using reinforcement learning in combination with interactive theorem proves.

For example, TacticToe [19] is an automated tactical prover. It is build on top of the interactive theorem prover HOL4 [5], created by Cambridge University. It uses Monte Carlo Tree Search to solve theorems concerning classical first order logic.

E Prover [7] is an automated theorem prover that uses deep neural networks to solve classical order theorems. Automated theorem provers are not dependent on an interactive theorem prover, as automated tactical provers are and as our artificial student will be.

Brian Groenke [12] in 2018 used Q-Learning, a reinforcement learning algorithm, to create an automotive theorem prover that focuses on Core Logic. We will also use Q-Learning in this project.

Kusomoto, Yahata and Sakai [13] also have used Coq, to create an automated tactical prover to solve intuitionistic propositional logic. This research is very similar to our own, the main difference is they go into deep learning, while we use other reinforcement algorithms that do not require neural networks.

Overview Report

We start with explaining how a proof is constructed in the interactive theorem prover Coq (Chapter 2). Then we proceed to defining a Markov Decision Process (Chapter 3), which is a vital first step to be able to use reinforcement learning, from which we will then explain the three algorithms we use (Chapter 4).

In Chapter 5 we take the step from theory to practice. We will go a bit further into one aspect of the program, namely how we define states to be equivalent in Chapter 6.

In Chapter 7 we compare aspects of our program and end with our discussion (Chapter 8) and conclusion (Chapter 9).

Chapter 2

A Proof in Coq

2.1 Introduction

Coq [2] is an interactive theorem prover and proof assistant. This means that it can verify mathematical proofs created by an external source, for example a student. It was created about 30 years ago, and has been constantly optimized by an active community [1].

In this chapter, we explain those aspects of Coq that attribute to a proof construction. The definitions of these aspects are mostly derived from the definitions and explanations found in *Coq'Art* by Bertot and Casteran [9].

Coq is based on type theory, instead of the more commonly used set theory. Every term that can be encountered or worked with in Coq has a certain 'type', which restricts the operations that may be performed on it. Most of the types we encounter are Propositions. Types are mentioned a few times in this chapter, but will not be important later on [17].

The figures of proofs and intermediate steps were created with CoqIDE, an integrated development environment for easy communication with Coq and constructing proofs [6].

2.2 Environments

During a proof, Coq keeps track of certain information about the proof and about the mathematical world in which the proof is being constructed. There are three main parts in which this information is contained:

1. Global environment
2. Local environment
3. Goal.

Global Environment

The global environment can be defined as follows.

Definition 2.2.1. Global Environment [9, p. 29]
A *global environment* consists of the initial environment, the imported libraries and all global declarations and definitions made by the user. This will be denoted as E.

To properly understand the definition of the global environment we need the following definition of a global declaration and definition.

Definition 2.2.2. Global declarations and definitions [9, p. 29]
A *global declaration* adds a certain identifier with a certain type to the global environment. A *global definition* adds a certain identifier with a certain value and a certain type to the environment.

The initial environment contains definitions of certain types and functions. The imported libraries contain the axioms of the intuitionistic propositional logic.

Both the initial global environment and the imported libraries are static, thus the only way for the global environment to change is if a global declaration or definition is made. Since we will not use any global declarations or definitions, the global environment will not change.

Local Environment

The local environment can be defined as follows.

Definition 2.2.3. Local Environment [9, p. 19]
A *local environment* consists of all local declarations and definitions made by the user. This will be denoted as Γ.

Definition 2.2.4. Local declarations and definitions [9, p. 19]
A *local declaration* adds a certain identifier with a certain type to the local environment. A *local definition* adds a certain identifier with a certain value and a certain type to the local environment.

The local environment contains all the variables we have defined and the hypotheses containing these variables we have introduced. This thus contains all the knowledge we already have collected and hypotheses we already know to be true.

In Figure 2.1 the local environment is everything above the horizontal line.

Goals

Finally the last part we look at is the goal.

Definition 2.2.5. Goal [9, def. 8]
A *goal* is a pair of a local environment Γ and a type G that is well-formed in this local environment.

As an example in Figure 2.1 the goal is shown in its entirety, local environment Γ is above the horizontal line, the well-formed type $G = Q \wedge P$ is beneath it.

```
1 subgoal
P, Q : Prop
H : P /\ Q
______________
Q /\ P
```

Figure 2.1: An example of a goal

During a proof it is possible to have multiple subgoals simultaneously, each containing their individual local environment. All of the subgoals need to be proven separately to complete the proof. As soon as a subgoal has been completed it disappears from the list of subgoals, along with its local environment.

An example with multiple subgoals can be seen in Figure 2.2, where there are two subgoals namely P and Q, where Q is the active subgoal.

```
2 subgoals
P, Q : Prop
H : P /\ Q
______________
Q
______________
P
```

Figure 2.2: An example with multiple subgoals

2.3 Tactics

In Coq you have access to a collection of tactics, which are tools to construct a proof.

Definition 2.3.1. Tactic [9, def. 9]
A *tactic* is a command that can be applied to the active goal. The effect is to produce a new (possibly empty) list of goals or change something in the local environment.

Definition 2.3.2. Proposition [9, def. 4]
Every type P whose type is the sort **Prop** is called a *proposition*.

Definition 2.3.3. Hypothesis[9, def. 5]
A *hypothesis* is a local declaration of the shape $H : P$, with H an identifier and P a proposition.

Hypotheses are always in the local environment, and show us what is known to be true during a proof.

Definition 2.3.4. Theorems or Lemmas [9, def. 7]
Global definitions of identifiers with as type a proposition are called *theorems* or *lemmas*.

2.4 An Example

As an example we have used CoqIDE to create a simple proof. We executed three tactics, after each tactic a figure was made. The green colored text has already been executed.

INTROS takes a goal of the form $A \to B$, and adds A to the local environment Γ and underneath the line leaves B. In this case $A = \forall P, Q : \text{Prop}, P \wedge Q$ and thus the tactic introduces P and Q as propositions and the hypothesis $H : P \wedge Q$. APPLY H takes the hypothesis named H and uses it to solve the goal. QED concludes every proof and can only be done if there are no more subgoals.

For example in Figure 2.4 the local environment contains three hypotheses P, Q and H. The goal is P. Right before the QED tactic is executed, we can see there are no more subgoals and the local environment is emptied.

```
Lemma test: forall P Q : Prop, P /\ Q -> P.
Proof.
intros.
apply H.
Qed.
```

```
1 subgoal
______________________________________
forall P Q : Prop, P /\ Q -> P
```

Figure 2.3: First step of a proof in Coq

```
Lemma test: forall P Q : Prop, P /\ Q -> P.
Proof.
intros.
apply H.
Qed.
```

```
1 subgoal
P, Q : Prop
H : P /\ Q
______________________________________
P
```

Figure 2.4: Second step of a proof in Coq

```
Lemma test: forall P Q : Prop, P /\ Q -> P.
Proof.
intros.
apply H.
Qed.
```

```
No more subgoals.
```

Figure 2.5: Third step of a proof in Coq

```
Lemma test: forall P Q : Prop, P /\ Q -> P.
Proof.
intros.
apply H.
Qed.
```

Figure 2.6: Fourth step of a proof in Coq

Chapter 3

Markov Decision Process

3.1 Introduction

Most reinforcement learning algorithms are based on a Markov Decision Process. Our Markov Decision Process is defined as an episodic process. An episodic process means that the Markov Decision process naturally splits up into episodes, where each episode has a natural beginning and end. For example if this was a process to learn a virtual student to play chess, each episode would be one game. In our case every episode will be a run-through of a proof.

To properly define a Markov Decision Process we need to specify the following objects

1. The State Space $\mathcal{S}$, a set containing all the possible states.
2. The Action Space $\mathcal{A}$, a set containing all the possible actions.
3. A Reward function.
4. A Transition function.

We also need to look at policies and value functions.

3.2 The State Space

The state spaces contains all possible states, which we have defined as follows.

Definition 3.2.1. State
A *state* is a list of goals (Definition 2.2.5), each consisting of a local environment Γ (Definition 2.2.3) and a well-formed type G. The state will be denoted by $\{[\Gamma_1 \vdash G_1], ..., [\Gamma_n \vdash G_n]\}$, for n subgoals.

Thus the state space contains all possible combinations of local environments Γ and well-formed types G, within the limits of intuitionistic propositional logic.

An example of a state with one subgoal which we can encounter can be seen in Figure 3.1.

```
1 subgoal
A, B, C : Prop
H : A \/ B -> C
______________________________________(1/1)
(A -> C) \/ (B -> C)
```

Figure 3.1: A state

Definition 3.2.2. Initial State
The *initial state* is the state where the episode starts. This will be denoted by $[\Gamma_0 \vdash G]$ with Γ_0 an empty local environment and G the to be proven lemma.

The initial state is the first state that is sent to the artificial student at the start of a proof. The local environment is completely empty, since the artificial student has not made any local declarations or definitions yet. G is the lemma or theorem that has to be proven. Thus the initial state depends on what the theorem is and will be the same every time a new proof gets started when doing multiple run-throughs of one lemma.

We can see the initial state in Figure 3.2, the local environment is empty, the goal is the lemma to be proven.

```
1 subgoal
_____________________________________(1/1)
forall A B C : Prop, (A \/ B -> C) -> (A -> C) \/ (B -> C)
```

Figure 3.2: The initial state

Definition 3.2.3. Terminating State
The *terminating state* is the state where the episode ends.

This will be either the state with no subgoals, denoted by $[\Gamma_0 \vdash \emptyset]$ or the state where the action space is empty, further explained momentarily. In Figure 3.3 an example of a terminating state can be found.

```
No more subgoals.
```

Figure 3.3: A terminating state after completing a proof

3.3 The Action Space

The action space is a set of tactics (Definition 2.3.1), which are the actions available to the artificial student to try to change the state and reach a terminating state. We chose a certain subset of the available tactics and added some extra tactics of our own, with the goal of making sure all possible intuitionistic propositional logic theorems can be solved.

In order to do this, we turn to the study of contraction-free Sequent Calculi of Roy Dyckhoff [11]. Dyckhoff explains the need for exactly twelve tactics and he proves that using these twelve tactics the proof will always terminate. Dyckhoff has shown that any provable theorem can indeed be proven and if a theorem is not provable these actions should be able to show this. For a more extensive look at these actions and the proof see the work of McCarren [14, chapter 2].

Thus the action space is a subset of the following twelve actions, with Γ as the local environment, G is an arbitrary goal, A, B , C, D are propositions:

$$\text{intro} : [\Gamma \vdash A \to B] \implies [A, \Gamma \vdash B] \tag{3.1}$$
$$\text{assumption} : [A, \Gamma \vdash A] \implies [\Gamma_0 \vdash \emptyset] \tag{3.2}$$
$$\text{contradiction} : [false, \Gamma \vdash G] \implies [\Gamma_0 \vdash \emptyset] \tag{3.3}$$
$$\text{split} : [\Gamma \vdash A \wedge B] \implies \{[\Gamma \vdash A], [\Gamma \vdash B]\} \tag{3.4}$$
$$\text{left} : [\Gamma \vdash A \vee B] \implies [\Gamma \vdash A] \tag{3.5}$$
$$\text{right} : [\Gamma \vdash A \vee B] \implies [\Gamma \vdash B] \tag{3.6}$$
$$\text{destructand} : [A \wedge B, \Gamma \vdash G] \implies [A, B, \Gamma \vdash G] \tag{3.7}$$
$$\text{destructor} : [A \vee B, \Gamma \vdash G] \implies \{[A, \Gamma \vdash G], [B, \Gamma \vdash G]\} \tag{3.8}$$
$$\text{imply1} : [(A \to B), A, \Gamma \vdash G] \implies [B, A, \Gamma \vdash G] \tag{3.9}$$
$$\text{imply2} : [(C \wedge D) \to B, \Gamma \vdash G] \implies [C \to (D \to B), \Gamma \vdash G] \tag{3.10}$$
$$\text{imply3} : [(C \vee D) \to B, \Gamma \vdash G] \implies [C \to B, D \to B, \Gamma \vdash G] \tag{3.11}$$
$$\text{imply4} : [(C \to D) \to B, \Gamma \vdash G] \implies \{[D \to B, \Gamma \vdash C \to D], [B, \Gamma \vdash G]\} \tag{3.12}$$

The first six of these necessary tactics are already implemented in Coq where they execute the required alterations to our state. The alterations defined above are the bare necessities this action has to do to adhere to the rules of Dyckhoff.

Since we have not defined these tactics ourselves, but used the premade tactics provided by Coq, these tactics alter more of the goal than what we showed here. For example we know for a fact the tactic ASSUMPTION first does an intro tactic before doing the above alteration, and some of the others do similar things. Initial testing has shown that for these tactics the extra alterations do not impact the provability of the theorem.

In this process we have exchanged the original tactic DESTRUCTION with two tactics DESTRUCTIONAND and DESTRUCTIONOR. We did this because the extra alterations this tactic does could be a problem. The problem here was that destruct needs to be done on a hypothesis in the local environment and if the artificial student does this action on the wrong hypothesis it is possible to get into a state where the student is stuck. We have made sure with our new destruct that it will always choose the right hypothesis and thus adher to the rules of Dyckhoff.

The last four actions IMPLY 1 through 4 are created by ourselves to do the actions for which we do not have predefined tactics yet. These do exactly the alteration Dyckhoff requires us to.

Specific Action Spaces

The action space is dependent on the state the artificial student is currently in, not all actions are applicable in each state. For example the tactics, that work on a hypothesis in the local environment, need to be defined per state. Thus we will refer to the action space as $\mathcal{A}_s$ for the action space in a certain state s or $\mathcal{A}_i$ for the action space at time i, where s_i is the current state.

We can denote a terminating state s, where the artificial student has no more actions, as $\mathcal{A}_s = \emptyset$.

3.4 The Reward Function

The reward function $R : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ finds the rewards during an episode. Our reward functions will be based on the last state visited, the new state, and the action space of the state.

A reward for a state-action pair at a certain time step i is denoted as $R(s_i, a_i) = R_i$. A reward is a real valued number.

The first type of reward function we work with is such that each action generates either a positive or negative reward. For example we can use a reward function that gives a reward of $+1$ to every state-action pair, that gives us a new state after doing the action in that particular state. It gives a reward of -1 to the state-action pair if it generates an error from Coq or the state does not change.

The second type of reward functions gives a reward if an episode is completed. Thus it will for example give a reward of $+1$ to the last state-action pair done in an episode, if it actually gives a complete proof. If the artificial student gets stuck in a proof, thus there are no more actions that can be done to reach a completed proof, none of the state-action pairs will get a reward. Another option is to give a negative reward to episodes that do not end in a completed proof.

3.5 Transition function

The transition function is $T : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$. This function is responsible for calculating the next state, based on the previous state and the action done in that state.

This transition function gives the next state by doing the above mentioned tactics and then returning the new goal, or possibly goals.

3.6 Value Functions and Optimization

The artificial student chooses an action during an episode using a policy. This policy $\pi : \mathcal{A} \times \mathcal{S} \to [0, 1]$ calculates the probability for each action in a certain state for it to be chosen.

The policy used is determined by the chosen algorithm and this will define which actions the student takes. During these run-throughs the student will learn from the feedback given after the actions. This feedback will be in the form of a reward.

The goal of having a Markov Decision process and then running a certain reinforcement learning algorithm is to optimize a policy.

For this we use the expected return from a certain time t on-wards. So we define the sum of the rewards after time t as G_t:

$$G_t \coloneqq R_{t+1} + R_{t+2} + ... + R_T \tag{3.13}$$

with R_i the reward at time step i and T as the final time step of the episode.

An additional concept that we will use in some of policies defined later, is the concept of discounting. The contribution to the total reward is less when the step generating the reward is farther away in the future. So if we are in time step t then the reward in time step $t + 1$ is worth more than the reward in $t + 6$ for example. For this we use the discount factor $\gamma \in (0, 1]$.
We then define the discounted sum of the rewards as:

$$\begin{aligned} G_t &\coloneqq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... && (3.14)\\ &= R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + ...) && (3.15)\\ &= R_{t+1} + \gamma G_{t+1}. && (3.16) \end{aligned}$$

The state value function for state s, under policy π is the expected return when starting in state s and afterwards following policy π. It is denoted by $v_\pi(s)$. We define the state value function as the expected values of all the possible rewards possible given we start in state s.

$$\begin{aligned} v_\pi(s) &\coloneqq \mathbf{E}[G_t | S_t = s] && (3.17)\\ &= \mathbf{E}\left[\sum_{n=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s\right] && (3.18) \end{aligned}$$

Another way to look at what the best possible action could be is to look at the expected reward value of a certain action state pair. We call this the state-action value function, denoted as $q_\pi(s, a)$, for taking action a in state s under a certain policy π.

$$\begin{aligned} q_\pi(s, a) &\coloneqq \mathbf{E}[G_t | S_t = s, A_t = a] && (3.19)\\ &= \mathbf{E}\left[\sum_{n=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a\right] && (3.20) \end{aligned}$$

We combine the results of these calculations in a Q-matrix. This matrix contains all the $Q(s, a)$, the value of the state-action value function for state s and action a.

The goal is of course to find an optimized policy, such that every choice gives the highest possible reward. We define a policy π' as being better or equal to another policy π if and only if $v_\pi(s) \leq v_{\pi'}(s)$ for all $s \in S$. There is not necessarily one optimal policy, but we denote them all as π^*. All optimal policies share the same state value function and the same optimal state-action value function defined as follows:

$$\begin{aligned} v_{\pi^*}(s) &\coloneqq \max_\pi v_\pi(s) \forall_{s \in \mathbf{S}} && (3.21)\\ q_{\pi^*}(s, a) &\coloneqq \max_\pi q_\pi(s, a) \forall s \in \mathbf{S}, a \in \mathbf{A}(s) && (3.22) \end{aligned}$$

3.7 An Episode of the Markov Decision Process

Each episode of our process consists of a certain number of cycles, as seen in Figure 3.4 for a certain policy π. In a cycle the artificial student starts in the initial state $s_0 \in \mathcal{S}$ and at time 1 takes a certain

action $a_1 \in \mathcal{A}$, given by the policy π. Then the environment calculates the new state $T(s_0, a_1) = s_1 \in \mathcal{S}$ and the reward $R(s_0, s_1, a_1) = R_1 \in \mathcal{R}$ and sends both to our artificial student. Subsequently the agent chooses action $a_2 \in \mathcal{A}$, again using the policy, and so forth. At some point the state sent to the artificial student will be the terminating state and this will conclude one episode.

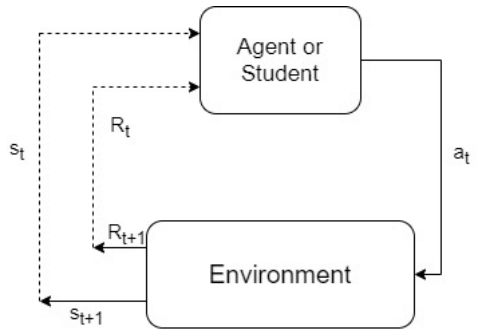

Figure 3.4: One step in the Markov Decision Process

Chapter 4

Reinforcement Learning Algorithms

4.1 Introduction

In this chapter we explain the three algorithms we have implemented. Each algorithm contains a policy to choose actions and a way to update the Q-matrix, which contains the values of the state-action pairs.

There are three main branches of Reinforcement Learning, i.e. Dynamic Programming, Monte Carlo methods and Temporal-Difference learning, as described in Chapter 4.2. Most of this information is gathered from the book *Reinforcement Learning: An Introduction* by Sutton and Barto [18].

In Chapter 4.2.1 we'll explain the Monte-Carlo style algorithm Epsilon Soft and in Chapter 4.2.2 the Temporal-Difference algorithms Sarsa and Q-Learning.

4.2 Branches of Reinforcement Learning

Dynamic Programming (DP) has been in development since the '50s [8]. It is a collection of algorithms that can be used to find optimal policies given a perfect model of the problem. As seen in Chapter 3 we have defined our state space as an infinite space, namely all possible combinations of goals. Since we cannot perfectly model an infinite space, we are not able to use any DP algorithms for our artificial student.

Monte Carlo (MC) methods require only experience to be able to learn. This means it uses sample sequences of states, actions and rewards from interaction to learn what to do. A model does not have to be completely defined. The main idea is for the artificial student to try out actions and evaluate them based on the feedback or rewards they receive. MC methods do not require any previous knowledge of the environment.

Temporal-Difference (TD) learning combines ideas from Monte Carlo and Dynamic Programming. Similar to Monte Carlo, TD methods can learn from experience, without needing a model of the environment. This is why we can still use this method of reinforcement learning for our model. An advantage that TD has over Monte Carlo methods is that the implementation goes more natural. After every step the return of a state-action pair is known, instead of having to wait until the end of an episode to be able to calculate it. Similar to Dynamic Programming, TD methods uses bootstrapping, which entails that the values of states are estimated in part by the estimations of the states visited afterwards.

4.2.1 Epsilon Soft

The Monte Carlo method we have implemented, we have named Epsilon Soft [18, p.101]. The name refers to the policy, and thus the way actions are chosen. The way the Q-values are updated uses the discounted sum of rewards as explained in Chapter 3.

There is a probability $\epsilon \in [0, 1]$ which is decided by the environment or user of the program beforehand. This probability dictates whether the artificial student chooses either one of the following two options:

- With probability ϵ the artificial student chooses a random action in the current state.
- With probability $1 - \epsilon$ the artificial student chooses one of the actions with the current highest Q-value.

In mathematical terms this looks as follows. The policy $\pi(a|s_t)$ calculates the probability that a certain action is chosen in state s at time t as previously introduced in Chapter 3.6. Let A^* be defined as the set of the actions which have the highest value for state s_t or $A^* = \arg\max_{a \in \mathbf{A}} Q(s_t, a)$ For all $a \in \mathbf{A}(s_t)$:

$$\pi(a|s_t) = \begin{cases} \frac{1-\epsilon}{|A^*|} + \frac{\epsilon}{|\mathcal{A}(s_t)|} & \text{if} \quad a \in A^* \\ \frac{\epsilon}{|\mathcal{A}(s_t)|} & \text{if} \quad a \notin A^* \end{cases}$$

After an episode is concluded the Q-values of the state-action pairs visited in the episode are updated. We use as a reward for a certain state-action pair the discounted sum of the rewards as defined in equation 3.16. Then the Q-value is updated to the average of all these discounted sum of rewards we have seen per state-action pair, with the discount factor chosen as $\gamma = 0.9$.

4.2.2 Sarsa and Q-learning

The other two algorithms we have implemented are Sarsa and Q-learning, which are TD learning. Both of these use the same policy, e.g. the way of choosing actions, as the previously discussed Epsilon Soft algorithm. The difference lies in the way the Q-values are updated. As seen in the previous chapter, the Q-values of Epsilon Soft are updated after every episode. With Sarsa and Q-learning the Q-values are updated after every action taken.

Using Sarsa[18, p.130], to calculate the new Q-value we need to also look at the next state-action pair that is visited, which is the bootstrapping aspect of the TD style algorithms. To find this action-state pair we again use the Epsilon Soft policy to choose the next action. Then the new Q-value is calculated by the following formula in state s doing action a, where α is a previously chosen variable, R is the reward given for that specific action, s' is the state visited after s and a' the action taken in that state.

$$Q(s,a) \Leftarrow Q(s,a) + \alpha(R + \gamma \cdot Q(s',a') - Q(s,a)) \tag{4.1}$$

This action a' is decided using the policy π and the algorithm at the same time also updates π. Thus Sarsa is an on-policy algorithm.

Q-learning [18, p.131] is very similar to Sarsa. Instead of using the policy in the next visited state s' to find the next action a', and use that Q-value for bootstrapping, we use the maximal Q-value in that specific state s':

$$Q(s,a) \Leftarrow Q(s,a) + \alpha(R + \gamma \cdot \max_{a \in \mathbf{A}} Q(s',a) - Q(s,a)). \tag{4.2}$$

Here the policy π is not used in generating the Q-values and thus Q-Learning is an off-policy method.

We use $\alpha = 0.7$ as the default value, we have not further examined the impact it has on the workings of the algorithms. This is in general a high α and thus the following state-action pair has a high impact on the value of the current state-action pair.

Chapter 5

Implementation

5.1 Introduction

In this chapter we give a short overview on Oasis, the program we created, and the adaptations we have made to the theory earlier discussed. We have created Oasis in Python and we communicate with Coq through a library called Sertop in SerAPI [3].

For a much more detailed overview of all the classes and methods we created see McCarrens thesis [14].

As a first step we implemented the Markov Decision Process described in Chapter 3. We have made adaptations to certain aspects, in particular the state space. These changes are described below.

5.2 Parts of Oasis

SerAPI Instance

In this part all the communication possible with Sertop/SerAPI, and thus with Coq, is defined. Herewith we can send statements to Coq and receive its feedback. These statements can be tactics or queries. A query for example can be used to ask for the proof context, which is a string containing the goals.

Student

In this part we created the artificial student. It uses SerAPI Instance to send its actions to Coq and then receives back the reward and new state. It incorporates everything to be able to do an episode and later on to check if the optimal actions actually give a complete proof.

It also contains the policy we have implemented. Since all three algorithms (Epsilon Soft, Sarsa and Q-Learning) use the same policy we have decided to put the policy, the epsilon greedy way to choose actions, in this section of the program, instead of with the algorithms.

State Space

This part controls everything which has to do with the state space. The agent can use this to see if a state has already been visited previously and it remembers the reward value pairs. The different state spaces are properly introduced and explained in Chapter 6.

We have defined our state a little differently then introduced in Chapter 3. Since our program cannot yet understand completely what the goal means, a state will initially be a string containing this information. We have improved on the information the program can get from this string (Chapter 6). The user can choose what type of state space is to be used.

Action Space

This section of Oasis contains all the tactics we have defined ourselves and an overview of the implemented ones.

As explained before the action space is dependent on the state we are currently in. Everything to do with creating this individual action space per state is implemented here.

We have also implemented another form of the action space, where the student has access to a 'simplify' action, which incorporates all the actions that do not need a specific hypothesis to work on. In this report this action space will not be used, but for more information see McCarrens report [14, chapter 5].

An aspect of our action space is the ability to remove actions that give an error out of the action space of a specific state. This ensures that a action space is created for each state with only the applicable actions. And from this again follows that if there is no way to complete a proof from this state, the student can notice this and abandon this episode. In our program this ability is a variable and can be chosen to be active or not. If it is not active a user should be aware that the student can get stuck for eternity, so a way out should be provided.

Reward Space

The reward space contains our reward functions.

The first reward function, named terminating states, gives a positive reward of $+1$ to the artificial student if a state-action pair results in a complete proof. If the episode has to be abandoned due to the student getting stuck in the proof it gets a reward of -1.

$$R_{term}(t) = \begin{cases} +1 & \text{if} \quad s_t = [\Gamma_0 \vdash \emptyset] \\ -1 & \text{if} \quad \text{Action space of a state is empty and } s_t \neq [\Gamma_0 \vdash \emptyset] \\ 0 & \text{else} \end{cases}$$

The second reward function, named standard, gives a positive reward of $+1$ if a state-action pair results in a different state. Thus if an action changes the state we are in. It gives a negative reward of -1 if the state-action pair does not result in a state changed.

$$R_{stand}(t) = \begin{cases} -1 & \text{if} \quad s_t = s_{t-1} \\ +1 & \text{else} \end{cases}$$

The third reward function, named standard & qed, combines the two previously explained reward functions, which gives a large positive reward $+100$ for completing a proof, but also still gives $+/-1$ for changing states or states staying the same.

$$R_{s\&q}(t) = \begin{cases} +100 & \text{if} \quad s_t = [\Gamma_0 \vdash \emptyset] \\ -1 & \text{if} \quad s_t = s_{t-1} \\ +1 & \text{else} \end{cases}$$

Algorithms

Here we have implemented the way to update the Q-values, according to the algorithms Epsilon Soft, Q-Learning and Sarsa introduced and explained in Chapter 4. It also contains all the methods needed to update our Q-matrix.

Chapter 6

State Spaces

6.1 Introduction

We have implemented three different types of state spaces in Oasis. In this chapter we will first look into the elements that play a key in seeing if states are equivalent. Next we will look at the three types of state spaces and explain these further.

In Chapter 3 we have given a theoretical definition of our state space. Oasis is not able to understand the content of the goal, so we changed the idea of the state. For our new states we used as the basis the pretty printed string of the goal that Coq can provide us with. As an example this looks like $'\backslash nH : A\backslash nH0 : B\backslash nA, B : Prop\backslash n$ ========== $\backslash nB'$

which corresponds to the state in Figure 6.1.

```
1 subgoal
A, B : Prop
H : A
H0 : B
______________________________________(1/1)
B
```

Figure 6.1: The state corresponding to the pretty printed string

It depends on the state space type chosen what the state actually looks like and how we see if they are equivalent. For example the state $[H : A \wedge B \vdash A]$ is equivalent to $[H : P \wedge Q \vdash P]$ for a human student, but Oasis won't be able to see this if the state is just a string and equivalence is checked by string comparison.

6.2 Elements for States to Be Equivalent

We have a few elements that we have considered as important for states to be equivalent, that we will further explain here.

Proposition names

The names given to propositions in a lemma should not change the state. Whether a proposition is named A or P or P123523 should not change the contents of the lemma itself. Thus we would prefer that proposition names are arbitrary.

Hypotheses names and order

The names of hypothesis after introducing them are chosen by Coq, but the names of the hypothesis should not impact if a state is different. Also the order of the hypotheses in the local environment should not matter.

6.3 State Spaces

We implemented three state space options, in which we elucidate what the state looks like and how we define equivalence between these states.

6.3.1 Simple

The Simple state space is - as the name suggest, the most simple way of looking at the state - defined as the string containing the goal given by Coq.

A state is thus only equivalent if the strings are completely equal. None of the elements of equivalence are covered by this state. When doing a few episodes with the same lemma this state is sufficient. However, as soon as multiple lemmas need to be proven by the same student, this state space is not adequate anymore.

6.3.2 Name

The Name state space takes the string and uses string parsing to gather the information of the hypothesis names, proposition names and the relations between them. It creates binary trees containing this information. Then we try to find mappings between this information between the binary trees already in the state space and the one constructed from the current state. If these functions exist we can conclude equivalence.

For more in depth explanation see McCarrens work [14, chapter 3].

6.3.3 Match

In the Match state space we used a feature in Coq called 'match goal'. This method takes a given pattern and tries to match it to the proof context we are working with [9]. The patterns have the form of $[H_1 = \ldots, \ldots, H_n = \ldots \vdash G]$. Thus it contains the active goal. If the matching between two states are positive we say the states are equivalent.

In this matching the proposition and hypothesis names and hypothesis orders are arbitrary. The states are thus defined in the form that they can be matched with each other using the 'match goal' method. This is created from the string, used in the simple state space. It is changed using regular string manipulation.

So when the artificial student changes a state during an episode, we check if there is an equivalent state which we already have visited. We do this by creating a 'match goal' statement with all visited states, whose string has the same length as the state the student is currently in, taking into account lengths of proposition names. Then this 'match goal' statement returns either the equivalent previously visited state and thus we know which state we are in, or it will return that this is a new state.

As an example let's look at this example of a possible state:

$$[H : A \wedge B \vdash A] \tag{6.1}$$

The state now would be defined as:

$$H : ?A \wedge ?B \vdash ?A \tag{6.2}$$

If you use this state in a match goal statement it would match with any that are similar. The question marks in front of the propositions communicate to Coq that we do not care what the name is. As long as the ones with a similar placeholder are still named the same. For example it matches with states such as $[H : P \wedge Q \vdash P]$, but it does not match with $[H : P \wedge Q \vdash Q]$.

There are still a few shaky details in this state space. The 'match goal' method is prone to match to a simpler goal than the actual state. For example if we try a 'match goal' statement of the form $[\vdash ?A]$, it will match with pretty much any state instead of the one that is actually equivalent in reality. This would mean there are multiple states to be matched with, and we have no way yet to see which is the correct one. We have tried to reduce this happening by adding that states have to be the same length, taking into account that proposition names length can vary.

Another problem that is bound to happen if we work with extremely large lemmas. If a state is very long, Coq does not give the entire local environment in the string. This leads to (...) being added in a string, which this state space cannot handle. For now this will raise an error, and thus this state space cannot be used for these.

Chapter 7

Results

7.1 Introduction

In this chapter we explain the experiments we have done. The goal of these experiments are to compare different aspects of our program, for example the different algorithms and state spaces.

For our results we use a subset of lemmas from the Intuitionistic Logic Theorem Proving (ILTP) library. This library is created to benchmark theorem provers for intuitionistic propositional logic. It also contains non-theorems, which automated theorem provers should be able to find out that they are impossible to solve. But Oasis isn't able to do that, so because of this we use the subset of theorems which can be proven. For a more in depth look at this database see McCarrens report [14, chapter 7].

This library contains classes of lemmas. Each class contains a certain type of lemma, where each lemma has a level. A higher level corresponds to a more difficult lemma of that certain type.

We will use the ability to remove actions that give errors from an action space, thus ensuring that the program will never get into an infinite loop. This results in abandoned episodes. We still count these episodes when evaluating the effectiveness of an algorithm.

7.2 Comparison of the algorithms

In our first experiment we compared our three algorithms, Epsilon Soft, Sarsa and Q-Learning. We compared them using one lemma and letting the artificial student practice it for 15 episodes and repeat this for 50 times. For these 15 episodes we looked both at the average amount of actions each algorithm needs and the amount of episodes which end in a completed proof.

We used the following conditions:

Conditions for Comparing Algorithms	
Lemma	$\forall A, B, C, D : \text{Prop}, (((C \wedge (B \wedge A)) \vee (C \to D) \vee (B \to D) \vee (A \to D)) \to D) \to D$ (number 43)
Algorithm	Epsilon Soft, Sarsa and Q-Learning
Epsilon	0.4
State Space	Simple
Reward Function	Terminating States (5.2)
Episodes	15
Average	500

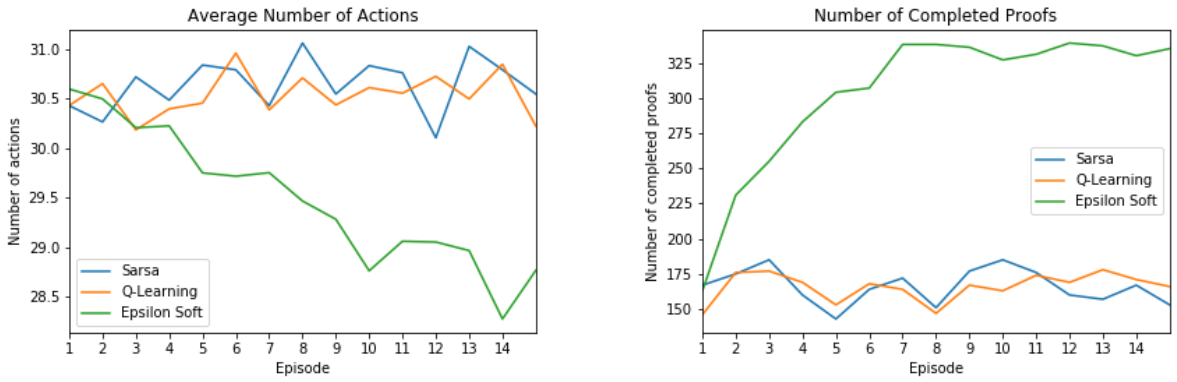

Figure 7.1: Comparison of Epsilon Soft, Sarsa and Q-Learning for one lemma

The result of this is seen in Figure 7.1.

This experiment implies that Epsilon Soft works the best, both in least amount of actions and the most completed proofs. There is a very clear curve where things improve.

There hardly seems to be any learning going on with Sarsa and Q-Learning, who interestingly are very similar. Both the average number of actions and the number of completed proofs are pretty constant.

7.3 Comparison of Possible Epsilons

In our second experiment we compared different values for the choice of epsilon.

For the algorithm we chose to only look at Epsilon Soft, since this algorithm has the best learning curve in early episodes.

We use the following conditions:

Conditions for Comparing Epsilons	
Lemma	$\forall A, B, C : \text{Prop} : (((A \wedge B) \vee (A \to C) \vee (B \to C)) \to C) \to C$ (number 42)
Algorithm	Epsilon Soft
Epsilon	$0.1, ..., 0.5$
State Space	Simple
Reward Function	Terminating States (5.2)
Episodes	10
Average	500

Again we have look both at the average amount of actions needed per episode and the amount of completed proofs.

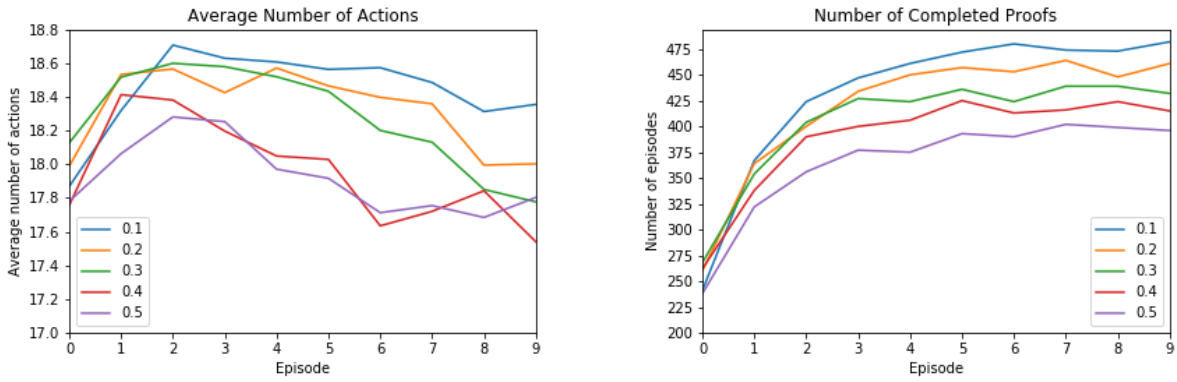

Figure 7.2: Different epsilons for Epsilon Soft

In Figure 7.2 it seems that lower epsilon implies that one needs more actions for an episode, but one does end more episodes in a completed proof. Higher epsilon does the opposite.

7.4 Comparison of State Spaces

In order to compare the possible state spaces (Simple, Match, Name) we conducted two experiments.

Experiment 1

The first experiment has to do with the classes of lemmas explained in the introduction of this chapter. Within a class of lemmas we expected lemmas to be very similar, for example similar states.

We look again at the amount of episodes ending in a proof and the average of the amount of actions need to end an episode. But we also looked at the average size of the state space at the end of a run through of an entire class. We hoped this would show if there are actually equivalent spaces in the class.

Conditions for Comparing State Spaces (1)	
Lemma	Class number 41 to 46
Algorithm	Epsilon Soft
Epsilon	0.4
State Space	Simple, Match or Name
Reward Function	Terminating States (5.2)
Episodes	1
Average	75

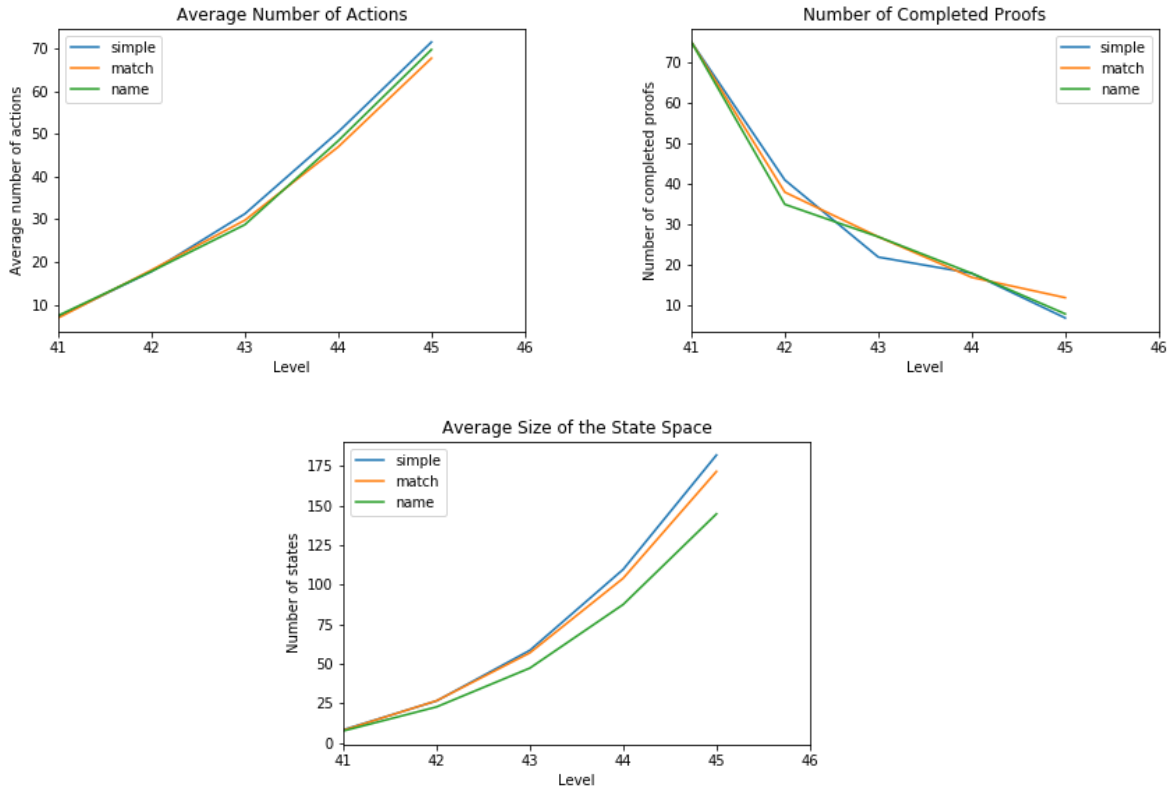

Figure 7.3: Comparing the different state spaces for one class of lemmas

The results can be seen in Figure 7.3.

We can either conclude from this that our equivalence does not work as hoped, or that while working through a class of lemma there are not a lot of equivalent states.

Experiment 2

The second experiment is about the elements of equivalence explained in Chapter 6, in particular the one concerning the different proposition names.

For this we used one and the same lemma several times, but we changed one or more of the proposition names in order to not change the structure of the lemma.

We looked at the average number of actions per episode, the amount of completed proofs in total and most importantly the average size of the state space per episode.

Conditions for Comparing State Spaces (2)	
Lemma	$\forall A, B, C, D, E : \text{Prop}, (((C \wedge (E \wedge (B \wedge A))) \vee (C \to D) \vee (E \to D) \vee (B \to D) \vee (A \to D)) \to D) \to D$ (number 44, different proposition names)
Algorithm	Epsilon Soft
Epsilon	0.4
State Space	Simple, Match or Name
Reward Function	Terminating States (5.2)
Episodes	1
Average	75

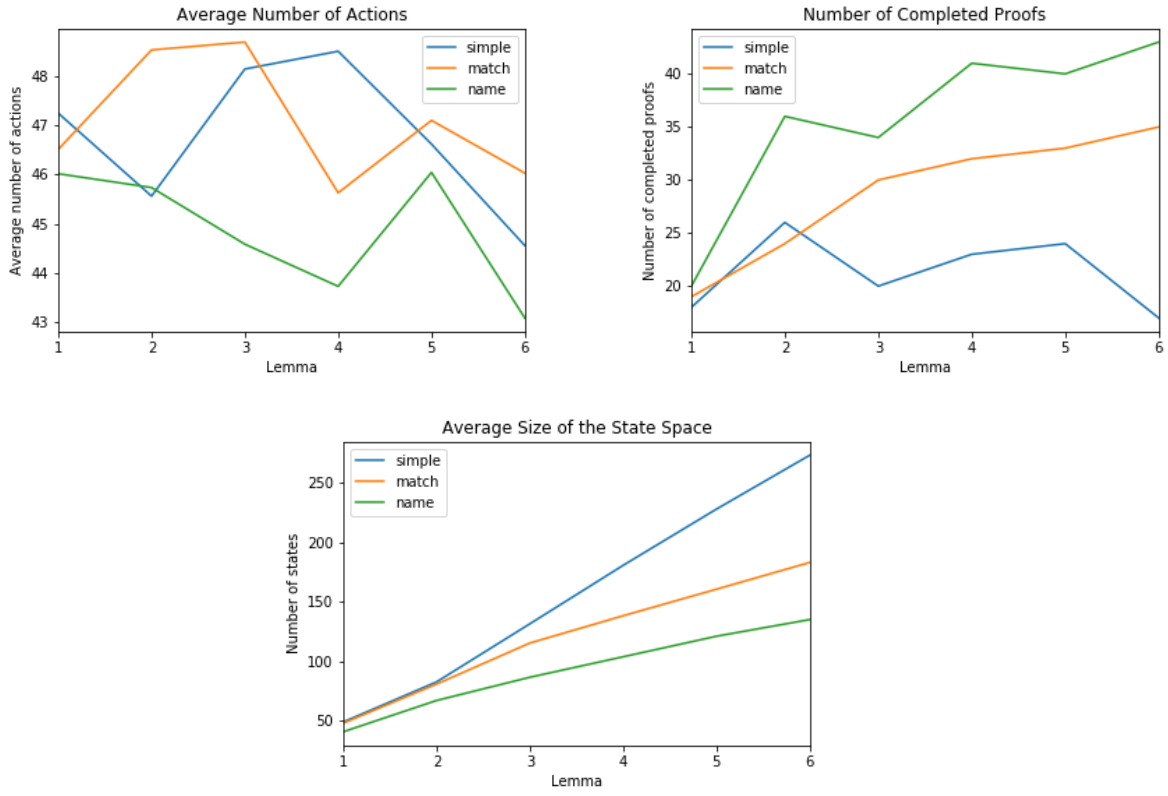

Figure 7.4: Comparing the different state spaces using the same lemma with changing proposition names

The results can be found in Figure 7.4.

These results imply that both the Name and the Match state space find equivalent states.

7.5 Comparison of Reward Functions

For this experiment we wanted to see if our different reward functions actually impacted the way the artificial student learns. Since Epsilon Soft seems to work the best, we used this algorithm to see if the reward functions make any impact.

Conditions for Comparing Reward Functions	
Lemma	$\forall A, B, C, D : \text{Prop}, (((C \wedge (B \wedge A)) \vee (C \to D) \vee (B \to D) \vee (A \to D)) \to D) \to D$ (number 43)
Algorithm	Epsilon Soft
Epsilon	0.4
State Space	Simple
Reward Function	Terminating States, Standard and Standard & qed (5.2)
Episodes	10
Average	100

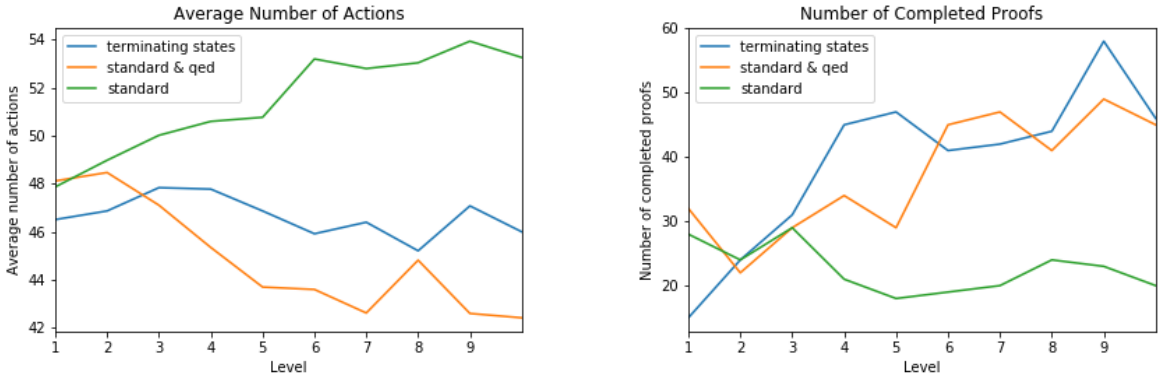

Figure 7.5: Comparing the different reward functions

from Figure 7.5 we can see that the Standard reward function has a higher average number of actions for most episodes and less completed proofs compared to the other two. Standard Qed seems to be the winner, but there is not a big difference with the Terminating States.

Chapter 8

Discussion

Python vs. OCaml

At the beginning of the project we were conflicted about which programming language to choose for our project. In the end we chose to work with Python, due to us having more experience with it and more resources being available concerning reinforcement learning. However, we quickly learned that the communication with Coq, through queries with Serapi/Sertop, takes way too long to be used in extensive testing and real life applications.

We would therefore recommend for future research to build directly onto Coq, and not to use an intermediate program to communicate.

Validity of Results

The problem of runtime, due to slow communication with Serapi/Sertop, also forms an obstacle for making conclusions based on our results. We are not able to compare our algorithms for example for a 100 episodes, due to Sertop shutting down after a certain amount of queries.

Thus our results are interesting, but the conclusions are not definite. To be able to get more information we would need to change the whole communication of our program.

State Spaces

We have implemented three types of state spaces, all with their own strengths and weaknesses.

The Simple state space is very fast, since the equivalence of states is only based on a simple string comparison. However, it is not very strong since it cannot find equivalent states between different lemmas, for example with different proposition names.

The Name state space is very strong in its finding of equivalent states. The problem here is its long runtime.

The Match state space is able to find equivalent states, with different proposition names and hypothesis names. Due to some changes, for example only doing a 'match goal' statement with similar lengths states, the runtime has been greatly reduced. But due to our sparse knowledge to how the string given by Coq changes for certain lemmas, bugs still prop up now and then. This means this state space is definitely not foolproof.

We think it would be better to be able to communicate with Coq directly and thus get more complete information about the local environment and goal and create the goal from there.

Reward Functions and Other Variables

We have only done a little experimentation with reward functions. We don't know the impact of which reward functions we choose and the size of the rewards.

Apart from an experiment to do with epsilon, we have not spent much time figuring out the impact of certain variables, namely α in Q-Learning and Sarsa and the discount factor γ in all three algorithms. These variables might have a large impact on the way the student learns.

All this warrants more research.

Chapter 9

Conclusions

We have created an artificial student that uses three algorithms Epsilon Soft, Sarsa and Q-Learning to learn to prove lemmas based on intuitionistic propositional logic.

The theoretical basis for our student was defined by how the interactive theorem prover Coq works, the Markov Decision Process and the three algorithms.

The best working algorithm we have implemented is Epsilon Soft. This is a Monte-Carlo approach which uses a greedy epsilon policy to choose actions and the average of discounted reward sums as its state-action values.

In our results we can see that the epsilon does have an impact on the learning achievement. A lower epsilon causes more episodes to end in a completed proof, a higher epsilon causes a lower average number of actions needed per episode. The best working state space representation is the Name state space, based on the binary trees. The Match state space, based on 'match goal' tactics, seems to have some potential.

We have experimented with three reward functions. The two, which give a positive reward for a completed proof, work better than the one based only with rewards only for changing states.

As a basis for future research, this artificial student can be first stepping stone to creating an artificial teacher who can actually help students in real life situations.

Bibliography

[1] Official Website of Coq. 1, 3

[2] Github of Coq. 3

[3] Github of SerAPI. 13

[4] Github of Waterproof. 1

[5] HOL Interactive Theorem Prover. 1

[6] CoqIDE. 3

[7] The E Theorem Prover, official website. 1

[8] Richard Bellman. The Theory of Dynamic Programming. 1954. 11

[9] Yves Bertot and Pierre Cateran. *Interactive Theorem Proving and Program Development, Coq'Art: The Calculus of Inductive Constructions*. 2004. 3, 4, 5, 16

[10] T.P.J. Beurskens. Computer programs for analysis Eindhoven University of Technology Computer Programs for Analysis Department of Mathematics and Computer Science. 2019. 1

[11] Roy Dyckhoff. Contraction-Free Sequent Calculi for Intuitionistic Logic. *The Journal of Symbolic Logic*, 57(3):795–807, 2010. 7

[12] Brian Groenke. Learning to reason. 2018. 1

[13] Mitsuru Kusumoto, Keisuke Yahata, and Masahiro Sakai. Automated Theorem Proving in Intuitionistic Propositional Logic. 1

[14] Sean McCarren. A Student-Teacher Reinforcement Learning Model for Automated Theorem Proving. 2020. 1, 7, 13, 14, 16, 17

[15] Grigori Mints. *A Short Introduction to Intuitionistic Logic*. 2000. 1

[16] Joan Moschovakis. Intuitionistic Logic. 1

[17] Rob Nederpelt and Herman Geuvers. *Type Thoery and Formal Proof: An Introduction*. 2014. 3

[18] Andrew G. Sutton, Richard S. Barto. *Reinforcement Learning — An Introduction*. Number 2. 2018. 1, 11, 12

[19] Josef Urban Ramana Kumar Michael Norrish Thibault Gauthier, Cezary Kaliszyk. TacticToe: Learning to prove with tactics. 2018. 1