Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics
Department of Applied Informatics

Reinforcement Learning with Abstraction

Dissertation thesis – shortened internet version

Michal Malý

Study programme: Informatics
Field of study: 9.2.1 Informatics
Supervisor: doc. Ing. Igor Farkaš, PhD.

Bratislava, 2012

Abstract

Reinforcement learning is a modern method of learning. It helps to solve many problems which comprise some notion of a short-term or long-term reward. It was successfully applied in the fields like robot control, elevator scheduling, problems in telecommunication, and chess. However, there are tasks involving the concept of reward, that we are not able to successfully solve. The existing theory provides only a few clues. The goal of this thesis is to try to solve the problems of only partially observable Markov processes or those that do not have the Markov property. We focus on tasks where a possibility to derive a world model brings an advantage.

First, we explain basic concepts of reinforcement learning, together with some other concepts, which are necessary for comprehension of the following text. We show limitations of basic methods of reinforcement learning. Then we present our own framework, which enables the agent to derive a world model using abstraction and use the model for decision making. Next, we present a prototypical implementation and demonstrate it on practical examples. Our method, reinforcement learning with abstraction (RLA) was able to solve a classical maze, a “letter” maze (a maze, where agent sees cells only as arbitrary letters assigned to them), three dimensional maze, maze with teleports, and was also able to solve a “protocol discovery” problem.

We have tested the performance on classical maze problems with varying size. Our method performed better than UDM method [McCallum, 1992]. It performed better in terms of steps necessary to discover the environment even with generic method, which were allowed full observation. When deprived of full observation, generic methods are not able to solve the problem. Thus, for some partial observation tasks, RLA can be one of few methods available.

We conclude the work with potential contribution and possible improvements.

Keywords: reinforcement learning, abstraction, world model, agent, grammar induction, automaton inference, model inference, Markov decision process, partial observability

Contents

List of Figures

Foreword

“Thou shalt not make a machine in the likeness of a man’s mind” is a chief commandment from the O. C. Bible – a religious book in a fictional world from the book Dune by Frank Herbert. In this world intelligent machines threatened humans and eventually were all destroyed in a “jihad”. The reconstruction of thinking machines was forbidden forever.

However, we do not live in this fictional universe. We hope that intelligent machines could be useful and in fact, we are using a number of quasi-intelligent machines. The whole field of Artificial Intelligence is to some extent concerned with the question how to create machines as intelligent as possible.

Although this work does not attempt to mimic human mind, our brains and minds are, and always have been, interesting, inspiring, and fascinating. One of the inspiration is the ability to create abstract concepts, which led us to considerations about minimum description length principle. The other inspiration is the ability to act and to learn from the experience, which leads us to reinforcement learning. Combining reinforcement learning with (at this time, very basic) ability to abstract seems to be a promising way how to solve simple tasks in partially observable worlds.

Introduction and overview

In this work, we rely on two concepts: reinforcement learning and abstraction. The reinforcement learning is a well-established paradigm. In Chapter 1 we explain its concepts, examples, and methods. We also mention standard problems and challenges for reinforcement learning. In Chapter 2 we explain basic concepts from the theory of formal languages. These concepts are necessary to describe the symbolic operations of the agent.

One of the challenges in the field of reinforcement learning is the problem of partially observable world (Chapter 3), which is the main theme of this work. How can the agent create a world model from partial observations? Here we connect with the second concept – abstraction. This notion is not so well-established, the interpretations of this concepts vary. We have tried to narrow its definition in order to be able to work with it and create an implementation based on this approach .

The goal of this work – to create a reinforcement learning method with the ability of abstraction – is presented and motivated from the theoretical point of view and also from the view of biological plausibility. The concept of abstraction is formalized and methods for its creation are introduced. From these, grammatical induction is a prominent one. It is an elegant and powerful method. However, its drawback is the computational cost (or even intractability). The necessity of the computational requirements is debated. The leading principle here is that it is better to solve the problem correctly at a great computational cost, than to solve the problem incorrectly or not to solve it at all.

The general framework for solving problems in partially observable world – RL with Abstraction Framework is presented in Chapter 4. A concept of meta-planning is also introduced here. Chapter 5 contains a prototypical implementation and in Chapter 6 we present the results of its testing and comparison to other approaches. The work concludes with Chapter 7 which recapitulates the results and presents possibilities for future work.

The work also contains two appendices – Appendix A contains an illustration of the model development. Appendix B contains a documentation for the source code of our method.

We would like to make a small language note here. We have chosen to use the animate pronoun “he” for the rational agent we are speaking about, although the agent is not alive. We have two reasons for this: First, we think it will help the reader to distinguish between the “animate” agent and the “inanimate” environment. Second, we are used to refer to the agent in this way similarly as the sailors are used to proudly call their ship “she”.

Chapter 1

Reinforcement learning

1.1 Introduction

Reinforcement learning is a mode of learning inspired by behaviorism. First notions of reinforcement were introduced by Skinner [1938]. Reinforcement learning lies in the fact that the learning agent acts on the basis of inputs (observations) from the environment, but the feedback – numerical evaluation signal (reward or punishment) – can be arbitrarily delayed. Therefore, he never sees the correct action in a given situation. Also, the fact whether the action was correct can only be inferred indirectly from an arbitrarily delayed reward signal. The agent's goal is to maximize this reward in the long term.

"Reinforcement learning is learning what to do – how to map situations to actions – so as to maximize a numerical reward signal. The learner is not told which actions to take, as in supervised forms of machine learning, but instead must discover which actions yield the largest reward by trying them. In the most interesting and challenging cases, actions may affect not only the immediate reward but also the next situation and, through that, all subsequent rewards. These two characteristics – trial-and-error search and delayed reward – are the two most important distinguishing features of reinforcement learning " [Sutton and Barto, 1998].

An important feature of reinforcement learning paradigm is that the agent has an active impact on the environment via his actions. Although, in general, several methods are used, any method that meets the described characteristics could be considered a reinforcement learning method.

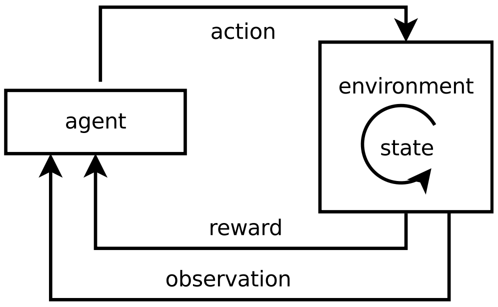


Figure 1.1: Model of reinforcement learning: agent interacts with the environment

1.2 Definition of the reinforcement learning problem

1.2.1 Markov property

The Markov property, or Markov assumption (originally formulated by Markov [1906]), is a useful concept in defining a "memoryless" stochastic process, i.e. that the conditional probability distribution of the next state depends only upon the present state. Knowledge of the history of the process does not add any new information. Formally:

Definition 1 (Markov property)**.** Let $\{X(t), t \geq 0\}$ be a time continuous stochastic process which assumes non-negative integer values. The process is called a discrete Markov process if for every $n \geq 0$, time points $0 \leq t_0 < t_1 < \ldots < t_n < t_{n+1}$ and states $i_0, i_1, \ldots, i_{n+1}$ it holds that

$$\begin{aligned} P(X(t_{n+1}) = i_{n+1} \mid X(t_n) = i_n, X(t_{n-1}) = i_{n-1}, ..., X(t_0) = i_0) = \\ = P(X(t_{n+1}) = i_{n+1} \mid X(t_n = i_n)). \end{aligned} \tag{1.1}$$

In the reinforcement learning problem the agent interacts with the environment (see Figure 1.1). In time intervals $t = 0, 1, 2, \ldots$ the agent receives information – observed state s_t and chooses, according to his strategy, an action a_t from a pre-defined set $\mathcal{A}$. As a result of this action, the state of environment changes. The agent again receives the next observation s_{t+1}, and receives also a numeric reward r_{t+1}[1] (which may be zero). This is

[1]The notation for reward varies in the literature. Some authors choose to denote the reward r_t to emphasize that the reward is given as a response to the action a_t. It is possible to use the notation r_t, in this case, the series of observations, actions and rewards is $s_0, a_0, r_0, s_1, a_1, r_1, s_2, \ldots$. It is also possible to use the notation r_{t+1} as we do, in this case the series of observations, actions and rewards

repeated. The numeric reward is computed according to the agent's performance. Thus, the reward can be the result of a long-term interaction and is not only an immediate result of the preceding action.

Usually the environment is seen as a Markovian process. The transitions between states can be stochastic. This illustrates the fact, that agent's actions may not be perfect or successful. It can also happen, that the agent has only limited access to the environment state, for example if his sensors are inaccurate. The access to the environment state can also be limited by the nature of the task: if the agent plays poker, he does not see into the cards stack.

The agent moves and interacts with the environment during some time. The task can end after a limited, pre-specified time, or after fulfilling a goal, or can continue for an unlimited amount of time. The goal of the agent is to maximize the reward in this span of time. This long-term return (beginning at time t) is usually formalized as the sum

$$\mathcal{R}(t) = \sum_{i=t}^{T} r_i \cdot \gamma^{i-t}, \tag{1.2}$$

where T is the total time of the task and $0 < \gamma \leq 1$ is the so called "discount factor". The discount factor specifies, how valuable are rewards in a distant time. If the total time T is pre-specified and finite (an episodic task), the case $\gamma = 1$ is allowed – here all rewards are considered equally valuable. In the other case, the inequality must be strict so it is guaranteed that the sum will not reach an infinite value.

1.3 Value-function methods of reinforcement learning

The methods for solving the reinforcement learning problem are based on the following: The agent internally maintains an estimated valuation (assignment of values) of states and expected outcome of actions. According to a chosen policy he decides which action to perform.

- The policy $\pi(s, a)$ denotes the probability that the agent takes the action a in the state s. It usually depends on the value of s and a (for example, the policy may be "take the action with the maximum value", called *greedy*). It may be deterministic, in case the notation $\pi(s) = a$ is used for the action taken in the state s.

is $s_0, a_0, r_1, s_1, a_1, r_2, s_2, \ldots$. As the agent receives the resulting state and the reward simultaneously, swapping them makes no difference, neither does shifting the index by one.

- The state-value function $V^\pi(s)$ denotes the expected (discounted) reward, if he occurs in state s and follows the policy π.
- The action-value function $Q^\pi(s, a)$ denotes the expected (discounted) reward, if he occurs in state s, takes the action a and then follows the policy π.

The action-value function is useful when the agent wants to take an action different from what will be appropriate according to the policy. For example, he does not want to execute the action with the best value, but an unknown action in order to explore the environment. Thanks to the separation of the action-value function and state-value function this "improvisation" will not influence the evaluation of the policy.

1.4 Bellman equation

Richard Bellman has formulated a necessary condition for optimal solution of problems solved by dynamic programming.

Theorem 1 (Principle of Optimality)**.** *[Bellman, 1957] An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.*

This principle is used to split the problem to small parts, recursively going from the first step to the next. In the problem formalized by a Markov Decision Process (MDP) it holds:

$$V^\pi(s) = E_\pi\{\mathcal{R}_t | s_t = s\} = E_\pi\{r_t + \gamma \sum_{s'} P(s'|s, \pi(s)) \cdot V^\pi(s') | s_t = s\} \tag{1.3}$$

This means that a value of a state following some policy is equal to immediate reward and discounted future rewards obtained when following this policy.

For an optimal valuation and optimal policy it holds, that:

$$V^*(s) = \max_a E\{r_t + \gamma V^*(s_{t+1}) | s_t = s, a_t = a\} \tag{1.4}$$

```
double epsilon = 1e-3; //stopping criterion for difference
double delta; //actual difference

foreach (s in States)
    V(s)=0;
do {
    delta = 0;
    foreach (s in States)
    {
        VoldValue = V(s);
        sum = 0;
        foreach (a in Actions)
        {
            foreach (s2 in States)
                sum+=Policy(s,a) *  Probability(s,a,s2) * Reward(s,a,s2) +
                    gamma * V(s2);
        }
        V(s)=sum;
        delta=max(delta,abs(V(s) - vOldValue));
    }
} while (delta > epsilon);
```

Figure 1.2: The algorithm for dynamic programming

1.5 Algorithms for reinforcement learning

1.5.1 Dynamic programming

Dynamic programming approach is based on a direct iteration of states and actions values (Fig. 1.2). We can afford this if we are not concerned about the computational time, and if we have the complete world model available (for example, a full Markov decision process is known, including the transition probabilities).

The equation for updates of the value function in dynamic programming is derived from Bellman equation (1.3):

$$\begin{aligned} V_{k+1}(s) &= E_\pi\{r_{t+1} + \gamma V_k(s_{t+1}) | s_t = s\} = \\ &= \sum_a \pi(s,a) \sum_{s'} P(s_{t+1} = s' | s_t = s, a_t = a) \cdot E\{r_t | s_t = s, a_t = a, s_t + 1 = s'\} \\ &\quad + \gamma V_k(s') \end{aligned} \tag{1.5}$$

Dynamic programming is computationally intensive, however, it has been proven to converge to the optimum [Sutton and Barto, 1998]. This is not guaranteed for other strategies. As we can see, the program uses the so called "in place" value updates, instead of updating the whole vector of values at once (using a temporary copy). Both approaches are possible, the "in-place" approach appears to be faster, as explained by Sutton and Barto [1998]: "To write a sequential computer program to implement iterative policy evaluation, as given by (1.5), you would have to use two arrays, one for the old values, and one for the new values. This way, the new values can be computed one by one from the old values without the old values being changed. Of course it is easier to use one array and update the values 'in place,' that is, with each new backed-up value immediately overwriting the old one. Then, depending on the order in which the states are backed up, sometimes new values are used instead of old ones on the right-hand side of (1.5). This slightly different algorithm also converges to V^π; in fact, it usually converges faster than the two-array version, as you might expect, since it uses new data as soon as they are available. We think of the backups as being done in a sweep through the state space. For the in-place algorithm, the order in which states are backed up during the sweep has a significant influence on the rate of convergence. We usually have the in-place version in mind when we think of DP algorithms" [Sutton and Barto, 1998].

Other approaches, such as solving the respective linear equations, are possible. However, they seem to be slow and are not used in practice. "Other ways to find Q^h include online, sample-based techniques similar to Q-learning, and directly solving the linear system of equations. (...) The main reason for which policy iteration algorithms are attractive is that the Bellman equation for Q^h is linear in the Q-values. This makes policy evaluation easier to solve than the Bellman optimality equation (1.4), which is highly nonlinear due to the maximization in the right-hand side" [Babuška and Groen, 2010].

1.5.2 Monte Carlo

Monte Carlo method (first used for pole balancing by Michie and Chambers [1968]) is based on experience. The agent learns by averaging the samples from the on-line interaction or from the simulation. Monte Carlo does not need a given Markov model nor does create it. It can be used to solve episodic tasks only – it requires the episode to end. After each episode, the reward is used for updating the estimation of state-action pair value: the estimation $Q^\pi(s, a)$ is computed as the average of the returns that have followed visits to the state s in which the action a was selected. After a sufficient number of episodes the average is a good estimation of state-action value function Q^π. This step – computing an estimation of Q^π for a given policy π – is called policy evaluation.

After we have evaluated the policy π, we can modify it by making the policy greedy with respect to the current value function. We update π as follows:

$$\pi(s) = \arg\max_a Q(s,a).$$

This step is called policy improvement. The steps for policy evaluation and policy improvement are repeated. For Monte Carlo it is usual to execute policy improvement after each episode, i.e. evaluation and improvement alternate on an episode-by-episode basis.

1.5.3 Temporal difference learning

Temporal difference learning improves the Monte Carlo method. While Monte Carlo waits with an update until the result of the action is known, TD-learning uses the expected value (Fig. 1.3). This is similar to the dynamic programming method. TD-learning does not create a world model as well. Let us suppose that at time t the agent visited state s_t. To update state-value function, Monte Carlo needs to know $\mathcal{R}(t)$, the actual return following time t (defined in Eq. 1.2), which is known only after the episode ends. TD-learning instead approximates $\mathcal{R}(t)$ using the immediate reward r_t, and using the current value of $V(s_{t+1})$ as an estimation of future returns.

$$\mathcal{R}(t) = \sum_{i=t}^{T} r_i \cdot \gamma^{i-t} = r_t + \gamma \sum_{i=t+1}^{T} r_i \cdot \gamma^{i-(t+1)} = r_t + \gamma V(s_{t+1}) \tag{1.6}$$

To incrementally update $V(s)$ in each step, we choose a parameter α. The update can be then be given as

$$V(s_t) \leftarrow V(s_t) + \alpha[\mathcal{R}_t - V(s_t)] = V(s_t) + \alpha[r_t + \gamma V(s_{t+1}) - V(s_t)] \tag{1.7}$$

1.5.4 Q-learning

The Q-learning, proposed by Watkins [1989], uses a similar mechanism to learn the state-action function Q instead of the value-function V. The update is given by

```
const double alpha;//learning rate

foreach (s in States)
    V(s)=0;//anything

foreach (episode in Episodes)
{
    State s = episode.startState();
    for(t in Time)
    {
        a = getAction();//according to the policy
        ExecuteAction(a);
        r = getReward();
        s2 = getState();
        V(s)=V(s) + alpha * (r + gamma*V(s2) - V(s));
        s = s2;
        if (episode.end(s))
             break;
    }
}
```

Figure 1.3: TD-learning algorithm

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{1.8}$$

The learned action-value function Q directly approximates optimal Q^*, independent of the agent's policy π. Sutton and Barto [1998] note, that this dramatically simplifies the analysis of the algorithm, and that it was one of the most important breakthroughs in reinforcement learning.

1.5.5 Q(λ)-learning

Q(λ)-learning (also suggested by Watkins [1989]) uses so called *eligibility traces*[2]. The trace represents how much is a state or action eligible for learning changes. When a reinforcement event – a difference between expected and obtained reward – occurs, only the values for eligible states and actions are modified. Naturally, only states and actions that were executed are eligible, and are eligible proportionally to the recency of their occurrence. Also, Q-learning uses the $\max_a Q(s_{t+1}, a)$ term, which can be different from the actual term for the action executed, in case that an exploratory step, and not the *greedy* one, was performed. In this case the causal chain must be also interrupted – previous actions and states are not "responsible" for the future outcome. Formally, eligibility trace $e_t(s, a)$ has the following value:

$$e_t(s,a) = \begin{cases} 1, & \text{if } s = s_t \text{and } a = a_t \\ 0, & \text{otherwise} \end{cases} + \begin{cases} \gamma\lambda e_{t-1}(s,a), & \text{if } \max_a Q_{t-1}(s_t,a_t) = Q_{t-1}(s_t,a) \\ 0, & \text{otherwise} \end{cases} \tag{1.9}$$

The update of Q is then defined by

$$Q_{t+1}(s_t, a_t) \leftarrow Q_t(s_t, a_t) + \alpha e_t(s,a)[r_{t+1} + \gamma \max_a Q_t(s_{t+1}, a) - Q_t(s_t, a_t)] \tag{1.10}$$

1.5.6 SARSA

The problem with Q-learning is, that the update is computed always using the greedy policy (the $\max_a Q(s_{t+1}, a)$ term) without taking into aspect the policy which is the agent

[2] The concept of eligibility traces can be applied also to TD-learning and SARSA, we refer the reader to Sutton and Barto [1998] for details.

using for exploration. On the other hand, using always the *greedy* policy slows down the learning rate at the beginning, because other actions with lower predicted value may in fact be better.

The SARSA [Rummery and Niranjan, 1994] uses instead the state-action value associated with the action chosen by the current policy:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{1.11}$$

This update is done for every quintuple $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$. The letters in the quintuple have given the algorithm its name SARSA.

1.5.7 Actor-Critic methods

Advanced methods separate the action selection from the action evaluation. When choosing a random action we divert from the policy, which can lead to a different result than we might get if we follow the policy. Thus, the result of the random action should not be counted into evaluation of the given state. Instead of this, we will evaluate, if the reward has risen or fallen in comparison with the expectation. According to this error, the execution of the action in this state will be reinforced or weakened (Fig. 1.4).

1.5.8 Continuous methods

We can use function approximators, such as neural networks, to build the Actor/Critic mechanism. The neural network can generalize the experiences by making a good approximation. An additional benefit is that a neural network can operate on continuous input states and action space. This makes them a good candidate where the state space is continuous. (In order to use a discrete method we would have to artificially discretize the inputs.)

1.6 Human reward system

The concept of RL developed in the machine learning community has biological relevance, because the system of rewards functions also in animals, including humans. Thus, we briefly present also this overview.

The human reward system [Routtenberg, 1978] comprises ventral tegmental area, nucleus accumbens, amygdala, medial septum (clusters of neurons near septum pellucidum,

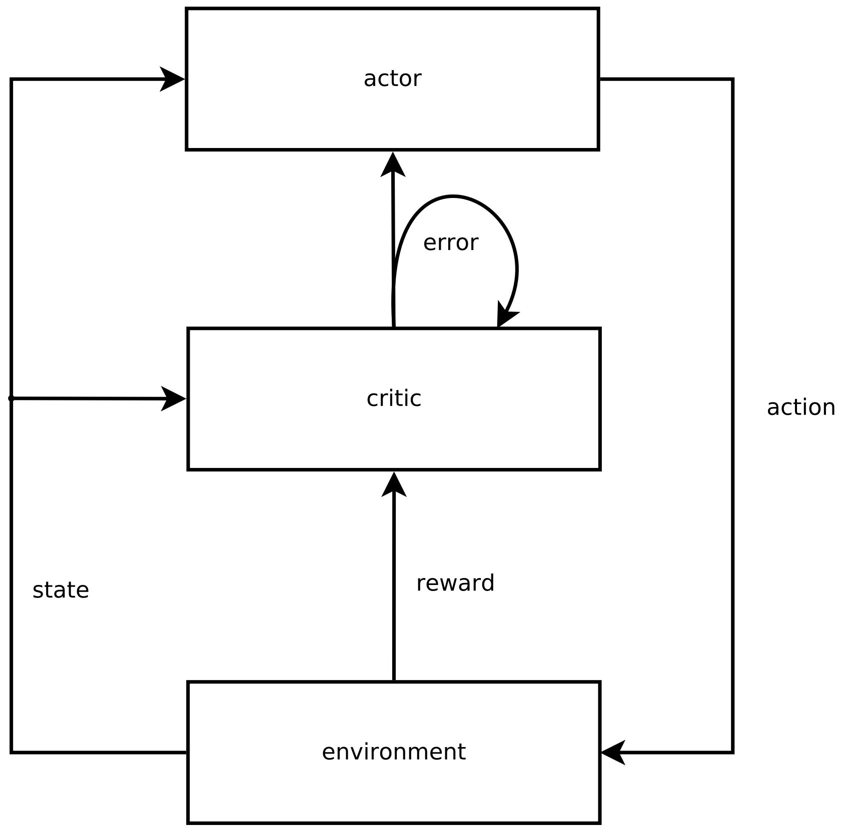


Figure 1.4: Actor-Critic

see Figure 1.5), and possibly hippocampus. The reward system controls our behavior by providing pleasure or unpleasant feelings. Rewards can stem from actions necessary for survival, such as obtaining food or sexual activities, or from situations which are somehow associated with them but do not provide direct survival, e.g. touch, money, art, and so on.

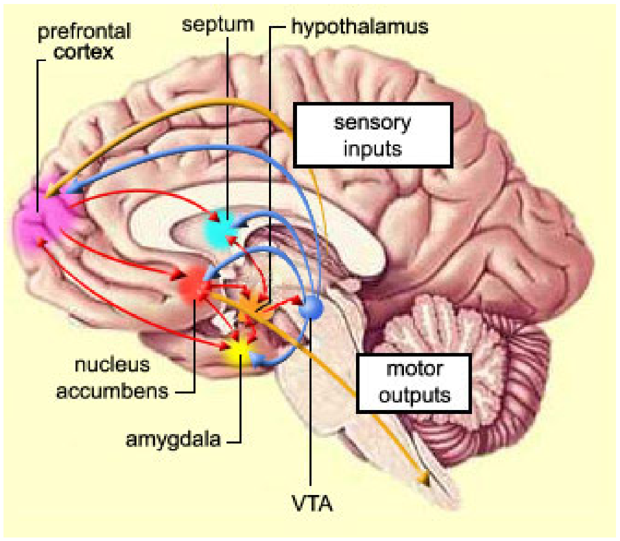


Figure 1.5: An illustration of human reward circuit [Dubuc, 2012], showing ventral tegmental area (VTA), medial septum, nucleus accumbens, and amygdala. These centers are tightly connected to hypothalamus, which is responsible for various regulatory functions, and to prefrontal cortex, which carries out executive functions.

We usually try to maximize our happiness, seeking experiences which are enjoying. The use of drugs can lead to pleasing experience, thus a person easily becomes addictive. The addiction is stronger, as the experience is stronger, more “pleasant” (hard drugs such as cocaine). Since our motivational system works in a similar way, we can say, that we are “addicted” to happiness.

Dopamine is a neuromodulator, which has many important functions in animal and human brains. Its roles include ”behavior and cognition, voluntary movement, motivation, punishment and reward, sexual gratification, sleep, mood, attention, working memory, learning, and aggression” [Couppis and Kennedy, 2008]. Some researchers consider the role of dopamine to be motivation and desire instead of pleasure (“want” versus “like”).

Dopamine may function analogically to temporal difference learning, where reward error is used as a teaching signal [Redgrave and Gurney, 2006]. When we get a greater reward than expected, this increases firing of neurons activated by dopamine. This leads to greater motivation to repeat the behavior. If the reward is lower than expected, neurons fire less and our motivation decreases [Arias-Carrión and Pöppel, 2007].

It is interesting, that in insects, dopamine's role is reversed. Here, releasing dopamine provides a punishment.

Other neuromodulator besides dopamine, which may contribute to the attention and mood is serotonin, which is also released when a positive event occurs.

It is also possible to induce direct electrical impulse to the septal area, what causes a rewarding stimulation. In the experiment by Olds and Milner [1954], the stimulation was linked to lever pressing. The rats pressed the lever as many times as possible, until exhaustion. After waking up, they began to press again.

1.7 Biological plausibility of RL

As mentioned in Section 1.6, the dopamine neurons appear to function in a similar way as the error function in temporal difference learning. In TD learning, the error function is the difference between estimated and actual reward. Used in the equation it drives the change necessary to accurately reflect the stimulus value and adapt to the environment. In an experiment [Schultz, 1998] the monkeys were trained to associate a stimulus with a liquid reward. If the liquid reward occurred according to the prediction due to the stimulus, dopamine neurons did not fire. If the reward occurred although no conditioned stimulus was present, a positive error occurred and dopamine neurons were activated. If the stimulus occurred, but the reward was missing, a negative error was reflected by depression of the dopamine neurons.

An alternative algorithm to TD learning, based heavily on experimental data from conditioning, was proposed, called *The primary value learned value (PVLV) model* [Hazy et al., 2010].

Dynamic treating regime in medicine

Dynamic treating regime is a treatment method based on optimizing a concrete individual patient's medical outcome. It is the application of methods analogous to reinforcement learning. The "reward" can be influenced by multiple goals, such as symptoms, time, cost, risks etc.

1.8 Standard challenges for reinforcement learning

Reinforcement learning is used to solving various problems. Because of its generality, it meets various challenges which have to be understood and resolved in order to use the method to solve the problem.

1.8.1 Curse of dimensionality

This difficulty occurs when a problem is represented exactly in all real-world details. Usually, all possible actions and all possible states form a huge state space; this means the agent in the search of optimum must explore the whole vast space.

The solution may be to work with a less-detailed representation (with some parameters neglected, actions constrained, and so on) or to use a function approximation (such as a neural network) which generalizes the value function. Then, the agent has to explore only a smaller fraction of the world.

1.8.2 Temporal credit assignment problem

> The Universe has as many different centers as there are living beings in it. Each of us is a center of the Universe, and that Universe is shattered when they hiss at you: “You are under arrest.” If you are arrested, can anything else remain unshattered by this cataclysm? But the darkened mind is incapable of embracing these displacements in our universe, and both the most sophisticated and the veriest simpleton among us, drawing on all life’s experience, can gasp out only: “Me? What for?” And this is a question, which, though repeated millions and millions of times before, has yet to receive an answer. *Alexandr Solzhenitsyn: The Gulag Archipelago*

The agent in an unknown environment faces a similar, yet probably a less dramatic problem of assigning the rewards or punishments to his previous actions. To which action should the agent be thankful for the reward? Which action caused the punishment? Which action(s) should be assigned a credit for the current good state?

The reward can be arbitrarily delayed. The learning signal from the action outcome becomes weaker with time, so it is necessary to perform more iterations to influence and reinforce the action.

1.8.3 Partial observability problem

Basic reinforcement algorithms require a fully observable world. Therefore they cannot be used when the agent cannot observe the state directly. The formalism of Partial observable Markov decision problems (POMDP) is used to describe approaches which are able to solve this class of the problems. Existing algorithms try to include an ad-hoc belief state to supplement missing information. In general, solving POMDP solving can be intractable and finding good special cases is a hot research topic [Sutton and Barto, 1998].

1.8.4 Non-stationary environments

Many RL algorithms converge slowly. If a fast changing, dynamic environment is presented, they fail. Sometimes this cannot be solved at all: In order to learn, something must be stable to be extracted by the learning algorithm.

1.8.5 Credit structuring problem

In order to run a RL agent, one has to design a reward mechanism. A correctly set reward may greatly influence the time necessary to find the solution.

However, one should be careful that the reward does not lead the agent in a trivial way. Let us suppose we put the agent into a maze and the reward is provided according to the distance between the agent and the target. The agent has a simple task of following the gradient. In fact, in order to compute the reward we have already "solved" the problem (using a typical symbolic algorithm such as Dijkstra) and we just provide the result of the computation to the agent.

Of course, in real-world situation the symbolic algorithm would be used instead of RL. However, from a theoretical point of view, one should be aware that the reward can influence the solution and the result of a theoretical experiment can be misleading.

1.8.6 Exploration-exploitation dilemma

After some time, the agent finds a way to gain reward through a set of actions. However, a part of the world might still be unexplored. This leaves the possibility of larger rewards but possibly also greater punishments. The exploration-exploitation dilemma reflects the problem whether the agent should use the existing information and execute only rewarding actions, or try to find a more rewarding behavior. In the former case, the agent might be stuck in a “local maximum”, in the latter case, he can spend energy uselessly and miss the known rewarding opportunities.

1.9 Unsolved problems of reinforcement learning

As mentioned in section 1.8.3, general POMDP can be intractable. As put by Sutton and Barto [1998], “if we are not willing to assume a complete model of a POMDP’s dynamics, then existing theory seems to offer little guidance.”

The goal of this work is to contribute in this field and therefore this problem is presented in Chapter 3 in more detail.

Chapter 2

Formal languages

To address the challenge of exploring POMDP world and creating its model, we have decided to use the framework of formal languages. This enables us to describe necessary symbolic manipulations used by our method.

A formal language is a set of finite words (i.e. which are of a finite length) over some alphabet. Instead of the term "word" sometimes the term "string" is used. Depending on the context and application, the alphabet can consist of letters, symbols, or tokens. A formal language is a useful tool to describe a problem of some kind. For example, the decision whether a number is a prime, is transformed to a decision whether a word representing the number belongs to the formal language containing all prime numbers.

2.1 Definitions

Definition 2 (Alphabet)**.** An alphabet is a finite, nonempty set of symbols. [Hopcroft et al., 1979]

Definition 3 (Word)**.** A word[1] is *a finite sequence of symbols chosen from some alphabet* [Hopcroft et al., 1979]. An empty word is denoted by the symbol ε.

Definition 4 (Powers of an Alphabet)**.** We define Σ^k to be the set of words of length k, each of whose symbols is in Σ. Let us also define $\Sigma^+ = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \ldots$ and $\Sigma^* = \Sigma^+ \cup \varepsilon$ [Hopcroft et al., 1979].

[1]Some authors, e.g. Hopcroft et al. [1979], prefer the term *string*.

Definition 5 (Language)**.** A set of words all of which are chosen from some Σ^*, where Σ is a particular alphabet, is called a *language*. If Σ is an alphabet, and $L \subseteq \Sigma^*$, then L is a *language* over Σ [Hopcroft et al., 1979].

The theory of formal languages studies mostly mathematical – syntactical and structural properties of formal languages. A formal language can be specified using

- mathematical notation – set specification (e.g. $L = \{a^n \mid n \text{ is a prime}\}$)
- a (formal) grammar – the set of rules describing, how it is possible to generate words
- an automaton (a theoretical machine), which for a given word decides, whether it belongs to the language – it is "accepted"
- a regular expression, which matches words of the language

The relation between these specification formalisms is an important part of the theory of formal languages. In this chapter we will introduce some of these formalisms and some problems.

2.2 Operations on words and languages

For convenience, we will introduce some operations on words and (almost) analogical operations on languages.

Definition 6 (Concatenation of Words)**.** Let x and y be words. Then $x.y$ (sometimes denoted as xy) means word concatenation, i.e. if x is a sequence of symbols $a_0, a_1, \ldots, a_m$ and $y = b_0, b_1, \ldots, b_n$, then $x.y = a_0, a_1, \ldots, a_m, b_0, b_1, \ldots, b_n$.

Definition 7 (Exponentiation of Words)**.** The word exponentiation is defined as $x^0 = \varepsilon,\ x^{(i+1)} = x.x^i$.

Definition 8 (Iteration of Words)**.** The iteration is defined by

$$x^* = \cup_{i=0}^{\infty}\{x^i\}.$$

For languages, analogous operations can be defined:

Definition 9. Let L_1 and L_2 be two languages. Then $L_1.L_2$ (sometimes denoted as L_2L_2) is the language consisting of all words of the form vw where v is a word from L_1 and w is a word from L_2.

Definition 10. The exponentiation is defined by $L^0 = \{\varepsilon\}, L^{(i+1)} = L.L^i$.

Definition 11 (Kleene iteration and the + operator)**.** The Kleene iteration is defined by

$$L^* = \cup_{i=0}^{\infty} L^i$$

In addition, let us define

$$L^+ = \cup_{i=1}^{\infty} L^i$$

In addition to these operations it is possible to use the standard set operations $\cup, \cap, \subset$ and others. The negation $\neg L$ is defined with respect to all possible words of the given alphabet. These operators could also be used for alphabets.

2.3 Formal grammar

A formal grammar is a set of rules which describe how to form strings in a formal language. Usually a formal definition is given, which consists of: a finite set of non-terminal symbols, a finite set of terminal symbols (alphabet), a finite set of production rules, and a starting symbol; in the formal notation it is a quadruple (N, T, P, S).

Definition 12. A grammar is a quadruple (N, T, P, σ), where N is a nonempty set of nonterminals, T is a nonempty set of terminals, P is a set of rules, $\sigma \in N$ is a starting nonterminal. The rules of P have to be of the form $(N \cup T)^* N (N \cup T)^* \to (N \cup T)^*$.

Definition 13. The relation $\Rightarrow_G$ of one-step derivation in G is defined by

$$x \Rightarrow_G y \quad \Longleftrightarrow_{\text{def}} \quad \exists u, v, p, q \in (T \cup N)^* : x = u.p.v \wedge p \to q \in P \wedge y = u.q.v$$

The relation $\Rightarrow_G^*$ is the reflexive transitive closure of the relation $\Rightarrow_G$.

Definition 14. (Language generated by the grammar) The language generated by the grammar G is denoted $L(G) = \{w \mid \sigma \Rightarrow_G^* w\}$.

2.4 Grammars of different power

The form of the rule can be further restricted. According to the level of restriction, a hierarchy of these grammar classes can be built. "Hierarchy" means that each class contains the preceding classes.

regular grammar (type 3) allows only rules of the form $N \to T^*N$ or $N \to T^*$, i.e. the nonterminal can be at most one and must be at the end of the rule "body" and the "head" is restricted to one nonterminal.

context-free grammar (type 2) allows only rules of the form $N \to (N \cup T)^*$, i.e. the head of the rule must be only one nonterminal, without the "context" (what gave the name to this class).

context-sensitive grammar (type 1) allows only rules of the form $x\alpha y \to x\beta y$, where $x, y \in (N \cup T)^*, \alpha \in N, \beta \in (N \cup T)+$, i.e. the "head" of the rule can specify context, which must be preserved. Moreover, the nonterminal β must not be erased and σ must not occur on the right side. As a special exception, the rule $\sigma \to \varepsilon$ is permitted[2].

unrestricted grammar (type 0) allows rules of any form, i.e. $(N \cup T)^* N (N \cup T)^* \to (N \cup T)^*$.

2.5 Chomsky hierarchy

An interesting finding by Chomsky [1956] is that the hierarchy of grammar classes, which seems arbitrary at the first glance, corresponds to the hierarchy of automaton classes. The class of regular languages (languages generated by a regular grammar) contains all finite languages (finite sets of words). A regular language can be recognized by a finite state automaton. The context-free languages class corresponds to the class of languages accepted by non-deterministic push-down automaton. A context-sensitive language can be generated by a context-sensitive grammar or accepted by a linear bounded automaton. The class of all languages generated by unrestricted grammar corresponds to the languages recognized

[2]This enables the grammar to generate an empty word ε. This technicality makes context-sensitive languages a superclass of context-free languages; without this addition only a context-free languages not containing ε would be contained.

by a Turing machine; or in other words, to the class of recursively enumerable languages. For the respective definitions of the machines (finite-state automaton, nondeterministic push-down automaton, linear bounded automaton, Turing machine) refer to Hopcroft and Ullman [2001].

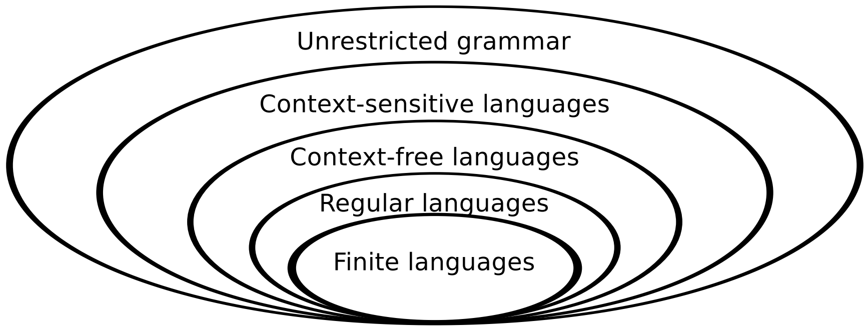


Figure 2.1: Chomsky hierarchy of languages

It is necessary to understand the diagram (Fig. 2.1) as a depiction of a series of theorems: every set is proper; it is possible to prove that the lower class language can be expressed by the means of the upper class, and that there exist languages in the upper class that cannot be expressed by the means of the lower class.

In the Chomsky hierarchy of grammars (see Fig. 2.1), regular grammars are in a lowest position. They are strictly weaker than classes of grammars higher in this position.

2.6 Turing machines and computability

The notion of an algorithm is strongly related to the Turing machines [Turing, 1937, 1938] – abstract computing machines. Compared to other computational processes (lambda calculus, recursive functions, and others), Turing machines operate in a very "mechanical" manner. Independent of the computational process used to describe algorithms, the theory of computability and undecidable problems shows us what we are or what we are not able to accomplish through programming. The limitation here is not the large amount time necessary to solve the problem, but the principal limitations of algorithms. However, one should always think about possible modification of the problem formulation and about possible heuristic solutions [3] This is also the way we will take when tackling uncomputable problems in Chapter 4.

[3]Take, for example, antivirus software. In its exact formulation, the problem of malware detection is undecidable. However, antivirus companies are quite successful in selling their products.

The Turing machine (TM) consists of a finite-state head and an infinite tape divided into cells. Each cell on the tape holds one of a finite number of symbols. The tape head is placed at one position on the tape, "above" a cell. The TM makes moves based on its current state and the tape symbol at the cell scanned by the tape head. In one move, it changes the state, overwrites the scanned cell with some tape symbol, and moves the head one cell left or right.

The following formal notation is according to Hopcroft et al. [1979].

2.6.1 Formal notation

Turing machine is a 7-tuple $M = (Q, \Sigma, \Gamma, \delta, q_o, B, F)$ where

- Q is the finite set of states,
- Σ is the finite set of input symbols,
- Γ is the complete set of tape symbols; $\Sigma \subseteq \Gamma$
- δ is the transition function. The arguments of $\delta(q, X)$ are a state q and a tape symbol X. The value of $\delta(q, X)$, if it is defined, is a triple (p, Y, D), where:
  1. $p \in Q$ is the next state
  2. $Y \in \Gamma$ is the symbol written in the cell being scanned, replacing whatever symbol was there.
  3. D is a direction in which the head moves, either "left" or "right".
- $q_0 \in Q$ is the start state, a member of Q, in which the finite control is found initially.
- $B \in \Gamma$ is the blank symbol. $B \notin \Sigma$ – it is not an input symbol. The blank symbol appears initially in all but the finite number of initial cells that hold input symbols.
- $F \subset Q$ is the set of final or accepting states.

We describe moves of a Turing machine by the $\vdash_M$ notation. Let $M = (Q, \Sigma, \Gamma, \delta, q_o, B, F)$ be a Turing machine. Define $\vdash_M$ as follows:

When the TM M is understood, we shall use just $\vdash$ to reflect moves. We also use $\vdash_M^*$, or just $\vdash^*$ to represent zero, one, or more moves of the TM M.

Suppose the next move is to the left: $\delta(q, X_i) = (p, Y, \text{left})$. Then

$$X_1X_2\cdots X_{i-1}qX_iX_{i+1}\cdots X_n \vdash_M X_1X_2\cdots X_{i-2}pX_{i-1}YX_{i+1}\cdots X_n$$

with the exceptions of $i = 1$, where

$$qX_1X_2\cdots X_n \vdash_M pBYX_2\cdots X_n$$

and the case of $i = n$ and $Y = B$, where

$$X_1X_2\cdots X_{n-1}qX_n \vdash_M X_1X_2X_{n-2}pX_{n-1}.$$

An analogic pattern appears for the moves to the right.

2.6.2 Halting

Turing machine accepts its input whenever[4] it reaches an accepting state, i.e.

$$L(M) = \{w \| \exists u, v \in \Gamma*, q' \in F : q_0w \vdash_M^* uq'v\}.$$

2.6.3 Undecidability

It can be demonstrated, that there exist problems that could not be solved by a computer. An example of such a problem can be the so-called diagonalisation language [Hopcroft et al., 1979]. To define it, we need to introduce the concept of codes for Turing machines. To represent a Turing machine we first assign integers to the states, tape symbols, and directions ("left" and "right"). We then encode rules of the transition function δ. Using a suitable separator, we can then concatenate codes for all rules to form a binary string. (Refer to Hopcroft et al. [1979] for technical details.) After we have done this, we can use the notation M_i, the i-th Turing machine. If i is not a valid code for Turing machine, we assume $L(M_i) = \emptyset$.

Definition 15 (Diagonalization language)**.** The language L_d is the set of strings w_i such that w_i is not in $L(M_i)$.

Theorem 2. *L_d is not a recursively enumerable language. That is, there is no Turing machine that accepts L_d.*

[4]There is also another notion of "acceptance" that is commonly used for Turing machines: acceptance by halting. We say a TM halts if it enters a state q, scanning a tape symbol X, and there is no move in this situation; i.e., $\delta(q, X)$ is undefined.

Proof. Suppose for some TM M, it holds that $L(M) = L_d$ Since L_d is a language over alphabet $\{0, 1\}$, M would be in the list of Turing machines we have constructed, since it includes all TMs with input alphabet $\{0, 1\}$. Thus, there is at least one code for M, say i; so we have $M = M_i$. Now let us investigate if $w_i \in L_d$.

1. If $w_i \in L_d$, then M_i accepts w_i. But then, by definition of L_d, the word w_i is not in L_d, because L_d contains only those w_j such that M_j does not accept w_j.
2. Similarly, if $w_i \notin L_d$, then M_i does not accept w_i. Thus, by definition of L_d, w_i is in L_d.

Since it is not true that $w_i \in L_d$ and neither it is true that $w_i \notin L_d$, there is a contradiction of our assumption that M exists. Thus, L_d is not a recursively enumerable language. □

The problems that can be solved by a Turing machine are divided into two classes:

- problems that have an algorithm (there is a Turing machine that halts whether or not it accepts the input)
- problems that are only solved by Turing machines that accept the input (and halt), but otherwise may run forever on inputs they do not accept.

The latter class of problems is undecidable.

2.7 Grammar induction

Grammatical induction (also called grammar inference) is a process of creating a formal grammar which produces a given (formal) language. Grammar inference is a special case of inductive learning, in which the goal is to create a formal grammar from positive and negative examples of the words of a language. Positive examples should belong to the language generated by the grammar, while negative should not be found in it.

There is always an infinite number of grammars satisfying this criterion; we usually look for the "simplest" grammar (for example a grammar with a minimal number of rules[5]). This corresponds to the minimum description length principle [Rissanen, 1978]. However, as the Kolomogorov complexity is incomputable, usually an approximation suffices. In the following text, we present some of the algorithms for grammar induction.

[5]In general, there can be more than one grammar which satisfies the minimality condition.

i	x_i^+	$\mathcal{P}$	$\mathcal{P}$ produces $\mathcal{D}^-$?
1	a	$S \to A$ $A \to a$	No
2	aaa	$S \to A$ $A \to a$ $A \to aA$	No
3	$aaab$	$S \to A$ $A \to a$ $A \to aA$ $A \to ab$	Yes: $ab \in \mathcal{D}^-$
3	$aaab$	$S \to A$ $A \to a$ $A \to aA$ $A \to aab$	No
4	aab	$S \to A$ $A \to a$ $A \to aA$ $A \to aab$	No

Table 2.1: An illustration of the progress of the trial and error algorithm.

2.7.1 Basic trial and error algorithm

The basic algorithm can be described as follows. The input is a set of positive examples $\mathcal{D}^+$, set of negative examples $\mathcal{D}^-$, and a specification of grammar type (1, 2, 3). The grammar type restricts the forms of the candidate rewrite rules. An initial grammar G_0 is guessed, G_0 is usually as simple as possible. The algorithm gradually expands the set of production rules as needed. Positive training sentences $x+$ are selected from $\mathcal{D}^+$ one by one. If $x+$ cannot be parsed by the grammar, then new rewrite rules are proposed for P. A new rule is accepted if and only if it is used for a successful parse of $x+$ and does not allow any negative samples to be parsed [Duda et al., 2001].

Example 1. (from Duda et al. [2001])

Consider inferring a grammar G from the following positive and negative examples: $\mathcal{D}^+ = \{a, aaa, aaab, aab\}$, and $\mathcal{D}^- = \{ab, abc, abb, aabb\}$. Clearly the alphabet of G is $\mathcal{A} = \{a, b\}$. We posit a single internal symbol for G_0, and the simplest rewrite rule $\mathcal{P} = S \to A$.

The Table 2.1 shows the progress of the algorithm. The first positive instance, a, demands a rewrite rule $A \to a$. This rule does not allow any sentences in $\mathcal{D}^-$ to be derived, and thus it is accepted for $\mathcal{P}$. When $i = 3$, the proposed rule $A \to ab$ indeed allows x_3^+ to be derived, but the rule is rejected because it also derives a sentence in $\mathcal{D}^-$. Instead, the next proposed rule, $A \to aab$ is accepted. The final grammar inferred has four rewrite rules shown at the bottom of the table.

2.7.2 Sequitur algorithm

The Sequitur algorithm [Nevill-Manning and Witten, 1997] infers a context-free grammar from a sequence of symbols. It replaces repeated string (of two symbols) occurrence with a grammatical rule that generates the string. In the new rule, a new nonterminal symbol is used. This process is repeated recursively, forming a hierarchical structure. At the end, each nonterminal used only once is replaced by its expansion. It can be used for lossless data compression.

Example 2. Take the decimal expansion of the fraction 22/7 to 48 decimal places: "3.142857142857142857142857142857142857142857142857". The Sequitur will generate the following grammar:

$S \to 3.\alpha\alpha$
$\alpha \to \beta\beta$
$\beta \to \gamma\gamma$
$\gamma \to 142857$

We can see that the algorithm successfully identified the repeating pattern "142857". However, it does not generalize – it produces the exact 48-decimal expansion, it does not try to approximate.

For example, having a 1536-digit expansion would need one starting nonterminal and 8 additional nonterminals ($1536 = 2^8 * 6$). A more effective way could be to encode the infinite expansion of 22/7, requiring only 2 nonterminals, and then limit the output in an ad-hoc way to 1536 digits. However, the ad-hoc limitation must be either outside the context-free grammar, or a grammar of higher power (e.g. context-sensitive) could be used.

2.7.3 Myhill-Nerode equivalence

In the previous work [Malý, 2011] we developed an algorithm to create a regular deterministic grammar by using a Myhill-Nerode equivalence and used it for inferring simple morphological properties of Slovak words. In principle, the algorithm could be applied to any inflecting language. A stemming dictionary can be created from a simple list of words of the language, without previous knowledge of the language and without specific rules for the language.

Principle of operation

Nerode [1958] proved the relation R defined on words u, v from the alphabet Σ by

$$u \; R \; v \iff_{def} \forall x \in \Sigma^*(ux \in L \Leftrightarrow vx \in L)$$

is a relation of equivalence. If the language L is regular, there is a finite number of equivalence classes. Our algorithm uses the relation R to find the equivalence classes and to infer the automaton. When applied to a sufficiently large and (at least to some extent) complete dictionary, one can observe repeating occurrence of the morphological suffixes. These occurrences turn out to be the frequently used states of the automaton. The suffixes generated by often-used states can be marked differently as the stem of the word. The whole process can be described as follows:

1. Create a list of all prefixes of all words.
2. For each prefix, create a set of suffixes that can be attached to it so we get a word in the dictionary.
3. Prefixes with the same set of suffixes belong to the same equivalence class.
4. For each equivalence class, assign a nonterminal. For each equivalence class and a suffix, assign a grammatical rule.
5. Reduce each nonterminal, which contains only one rule, by this rule in every occurrence of the nonterminal.
6. Count the number of uses of each nonterminal.
7. Mark the nonterminals with the count greater than or equal to the given threshold.
8. Generate the words from the grammar using recursion. If you reach a marked terminal, create a new group and add there all words from all children calls.
9. Output the groups.

rules for the nonterminal	#uses
$0 \rightarrow \ldots \mid mesta\ 56 \mid meste \mid mesteck\ 61 \mid mesto\ 48 \mid mestsk\ 455 \mid \ldots$	(start)
$3 \rightarrow i \mid \varepsilon$	111
$15 \rightarrow ho \mid j \mid \varepsilon$	179
$48 \rightarrow m \mid \varepsilon$	101
$49 \rightarrow ch \mid m \mid \varepsilon$	111
$56 \rightarrow ch \mid m\ 3 \mid \varepsilon$	121
$61 \rightarrow a \mid o \mid u$	7
$83 \rightarrow m \mid u$	35
$455 \rightarrow a \mid e\ 15 \mid i \mid o\ 83 \mid u \mid y\ 49$	2

Table 2.2: Excerpt of the resulting grammar

Results

We have run the algorithm on a dictionary of Slovak words occurring in the Slovak National Corpus [Jazykovedný ústav Ľ. Štúra SAV, 2009]. Table 2.2 presents an excerpt of the resulting grammar related to the words beginning with *"mest"*. Numbers denote nonterminals. Each nonterminal (except the starting nonterminal 0) is assigned a count of uses.

Limitations

Our algorithm has a similar disadvantage as mentioned in the example for the Sequitur algorithm: it does not try to generalize the input data.

Possible extension for infinite languages

The input in the described algorithm is always finite. For some purposes, this is enough. However, we can pose a question, if a similar algorithm could be created for an infinite language. The following steps would be necessary to accomplish this:

1. Define the input format of the language (finite automaton, grammar, regular expression).
2. If necessary, convert the input to a convenient internal representation (in the following text to an automaton).
3. It is not possible to enumerate sets of prefixes and suffixes – they can be infinite.

4. The representation of prefixes using equivalence sets. One set will represent all prefixes, which end in the same automaton state. (The number of sets is equal to the number of states of the automaton)

5. Comparison of the suffix sets using an temporary automaton. The comparison is converted to the problem of language difference. The automaton, accepting set of suffixes is the same as the original automaton, except its start state is the state, in which the prefix ends. The language difference is created using a standard automata construction, creating a temporary automaton. Then the temporary automaton is tested for the existence of a reachable end state.

Chapter 3

Reinforcement learning in a partially observable world

3.1 Introduction

Using reinforcement learning in non-Markovian worlds is a difficult task. As put in Sutton and Barto [1998], "if we are not willing to assume a complete model of a POMDP's dynamics, then existing theory seems to offer little guidance."

These limitations have been known for a long time. In Lin and Mitchell [1992], this problem was illustrated in the following task: "Consider a packing task which involves 4 steps: open a box, put a gift into it, close it, and seal it. An agent driven only by its current visual percepts cannot accomplish this task, because when facing a closed box the agent does not know if the gift is already in the box and therefore cannot decide whether to seal or open the box."

The authors also analyze three connectionist memory architectures, which extract attributes from the history, and so complete the agent's state information.

3.1.1 Partially observable Markov decision process

Definition 16 (Partially observable Markov decision process). *Partially observable Markov decision process* is a 6-tuple (S, A, O, T, Ω, R), where S is a set of states, A is a set of actions, O is a set of observations, T is the transition function, Ω is the observation function, R is the reward function.

In each step the agent executes the action a which causes the environment to change the (unobservable) state from s to s' with probability $T(s, a, s')$ and the agent observes o with probability $\Omega(a, s')$ and receives a reward $R(s, a)$.

Algorithms for solving POMDPs usually suppose a known model structure with unknown transitions probabilities. The probabilities and the states are derived during the search.

3.1.2 Perceptual aliasing

The problem of perceptual aliasing – the fact that in different states of environment the agent can have the same perception – was observed already by Whitehead and Ballard [1991], who proposed the *Lion* algorithm (see below).

3.2 Existing solutions and approaches for POMDP

3.2.1 The *Lion* algorithm

The *Lion* algorithm [Whitehead and Ballard, 1991] tries to maintain an internal state which is consistent with the observations. Where the confusion of the estimated and the real state occurs as the result of incomplete observation, *Lion* solves this unexpected situation by identifying the action, which leads to the inconsistent state. The algorithm resets the valuation of the action to zero and so prevents executing it and enables the agent to select another action.

3.2.2 State splitting: Utile distinction memory

An interesting work of McCallum [1993] is based on a statistical state splitting. The key is the utility of the state: if the world satisfies the Markov property, the rewards in a perceived state must be similar. If they are different, there must be a significant difference and splitting the state will help the agent to better predict rewards.

We consider this method interesting also from a biological point of view. Although not presented in the original paper, an analogy to a human (or animal) brain can be drawn. The human brain reacts differently in a situation where predicted reward is different from the actual reward (see section 1.7). It is possible that the brain makes the necessary steps to achieve a similar “state split” in order to capture the new situation.

3.3 Approaches to the space explosion problem

Unlike the algorithms in the previous section, the following algorithms attempt to solve a “dual” problem: how to cope with too-fine grained state space, i.e. having a very precise and detailed observation. A large state space means that it would take a long time until the agent visits all states. These algorithms try to generalize the experience and extrapolate it to the unvisited spaces.

3.3.1 G-algorithm

Chapman and Kaelbling [1991] proposed G-algorithm, which divides the world recursively to finer parts, and creates a tree structure for the action-value function. At the beginning it supposes that all bits from the input vector are irrelevant. The entire table is thus collapsed into a single block. The algorithm then collects Q-values and statistical evidence for the relevance of individual bits within this block. When it discovers that a bit is relevant, it splits the state space, with respect to the relevant bit. This is repeated – the blocks can in turn be split again.

3.3.2 State aggregation

In work by Singh et al. [1995] an approach was proposed based on soft aggregation of states. Each state can belong to several clusters with some probability. The agent can observe individual states, but can update the value function of the whole cluster only. The value of the cluster is then applied to its states according to their probabilities of belonging to this cluster. A heuristic algorithm is used to find good clustering probabilities for a fixed number of clusters.

3.3.3 Hybrid Probabilistic Logic Programs

An approach combining reinforcement learning and Answer set programming (ASP)[1] logic programming was proposed by Saad [2010]. A special action language was devised to represent agent’s observations and actions. The logical program is then solved to yield valid probable trajectory in the world. However, this method requires that we know the model of the environment (the states are not completely known).

[1] Answer set programming paradigm uses stable model semantics for problem solving. See e.g. Baral [2003] for details.

3.4 Other general approaches: Cognitive architectures

A general approach is to create a complex system which is capable of solving general problems. This system is usually called a cognitive architecture.

3.4.1 What is a cognitive architecture?

A cognitive architecture is a prototype for intelligent agents, which attempts to simulate the behavior of a cognitive system (usually that of a human), or to act intelligently. Cognitive architectures attempt to model the internal properties of the cognitive system and not only to mimic the external behavior [Langley et al., 2009].

A good cognitive architecture should incorporate learning, generalization, concepts of creation and knowledge representation.

Our framework (described in the Chapter 4) does not have this ambition, however, we consider it interesting and useful do describe some of the cognitive architectures.

3.4.2 What is a rational agent?

Russell et al. [2010] mention that "an agent's choice of an action at any given instant can depend on the entire percept sequence observed to date" and that "for each possible percept sequence, a rational agent should select an action that is expected to maximize its performance measure, given the evidence provided by the percept sequence and whatever built-in knowledge the agent has".

Vernon et al. [2007] outline different paradigms for cognitive systems (see Table 3.1).

Task nonspecificity

According to Weng [2012], an agent is task nonspecific if: "1) during the programming phase, its programmer is not given the set of tasks that the agent will end up learning; 2) during the learning phase, the agent incrementally learns various task execution skills from interactions with the environment using its sensors and effectors; and 3) during the task execution phase, at any time the agent autonomously figures out what tasks should be executed from the cues available from the environment."

Characteristic	**Cognitivist**	**Emergent**
Computational Operation	Syntactic manipulation of symbols	Concurrent self-organization of a network
Representational Framework	Patterns of symbol tokens	Global system states
Semantic Grounding	Percept-symbol association	Skill construction
Temporal Constraints	Not entrained	Synchronous real-time entrainment
Inter-agent epistemology	Agent-independent	Agent-dependent
Embodiment	Not implied	Cognition implies embodiment
Perception	Abstract symbolic representations	Response to perturbation
Action	Causal consequence of symbol manipulation	Perturbation of the environment by the system
Anticipation	Procedural or probabilistic reasoning typically using a priori models	Self-effected traverse of perception-action state space
Adaptation	Learn new knowledge	Develop new dynamics
Motivation	Resolve impasse	Increase space of interaction
Relevance of Autonomy	Not necessarily implied	Cognition implies autonomy

Table 3.1: A comparison of cognitivist and emergent paradigms of cognition [Vernon et al., 2007].

3.4.3 Overview of some cognitive architectures

We have chosen a few cognitive architectures which we consider most interesting from the computational or cognitive point of view.

CLARION

CLARION (Connectionist Learning with Adaptive Rule Induction ON-line) created by the Sun [2003, 2007] focuses on interaction between implicit and explicit processes and incorporates action-centered and non-action centered subsystems.

It comprises a number of separate subsystems: action control subsystem; general knowledge subsystem; motivational subsystem, which provides necessary motivation for perception, action and cognition; and metacognitive subsystem, which monitors and manages other subsystems. Each subsystem is dual with respect to implicit and explicit representations.

COGITOID

Cogitoid [Wiedermann, 1998b; Beran and Wiedermann, 2002] models "mental events" on a level of concepts formation, and excitatory and inhibitory links between them, abstracting from biological details. Concepts are represented as lattices[2] – sets of elementary properties of an object or an event. "Related concepts are connected using associations. During the computation, first the input concepts are activated and eventually those concepts are linked to the active concepts by excitatory associations" [Wiedermann, 1998a]. Active concepts are then sent to the output and the cycle is repeated again.

The author maintains that the Cogitoid, if connected *to the "similar inputs and acting on similar peripheries as the human brain, could form the behavior similar to that of the human mind. It could also explain the concept of "self", mental flow, introspection, problems of free will, language learning, speech generation and emergence of consciousness.*

One of works implementing and testing the properties of Cogitoid by Ačová [2002] states that the main disadvantage is the great computational complexity – cubical with respect to the large concept universe (usually the complete lattice, being $2^{\text{number of features}}$).

ACT-R

The ACT-R architecture [Anderson, 1996; Anderson et al., 1997] supposes, that human knowledge can be divided to two groups: declarative and procedural. Declarative knowledge is represented in a form of vector representation of the properties. These vectors are accessible via the interface for the modules. The modules are of two types, perceptual-motoric and memory. Perceptual-motoric modules provide connection with the outer world. The memory modules save declarative facts ("Bratislava is the capital of Slovakia") or procedural knowledge (how to make a step).

AIXI

The AIXI model [Hutter, 2007] is the most ambitious one. It uses Solomonoff's theory of universal induction and the author formally proved the general optimality. AIXI is uncomputable, but the author has proposed a restricted version AIXI^{tl} bounded by time t and length l. However, it was still unclear, whether the AIXI can be practically feasible.

Recently, a Monte Carlo approximation of AIXI model was proposed in Veness et al. [2009] and refined in Veness et al. [2010]. This approximation might be computationally

[2] The model of Cogitoid has been evolving, the most recent version of the model uses the lattices.

feasible.

3.5 Implications for autonomous rational agents

Vernon et al. [2007] conclude their report with the following design principles for systems that are capable of development, based also on studies Krichmar and Edelman [2006, 2005]; Krichmar and Reeke [2005]:

1. The architecture should address the dynamics of the neural element in different regions of the brain, the structure of these regions, and especially the connectivity and interaction between these regions

2. The system should be able to effect perceptual categorization: i.e. to organize unlabeled sensory signals of all modalities into categories without a priori knowledge or external instruction. In effect, this means that the system should be autonomous and, as noted by Weng [2004], a developmental system should be a model generator, rather than a model fitter (e.g. see Olsson et al. [2006]).

3. The system should have a physical instantiation, i.e. it should be embodied, so that it is tightly coupled with its own morphology and so that it can explore its environment

4. The system should engage in some behavioral task and, consequently, it should have some minimal set of innate behaviors or reflexes in order to explore and survive in its initial environmental niche. From this minimum set, the system can learn and adapt so that it improves its behavior over time.

5. The system should have a means to adapt. This implies the presence of a value system (i.e. a set of motivations that guide or govern its development). These should be nonspecific modulatory signals that bias the dynamics of the system so that the global needs of the system are satisfied: in effect, so that its autonomy is preserved or enhanced. Such value systems might possibly be modelled on the value system of the brain: dopaminergic, cholinergic, and noradrenergic systems signalling, on the basis of sensory stimuli, reward prediction, uncertainty, and novelty.

6. The brain-based devices should lend themselves to comparison with biological systems.

Our system aspires to satisfy four of these six principles (all except the first and the last one). Let us discuss this in more detail with respect to principles 2-5:

2. Our agent has no a priori knowledge and creates a world model from scratch. The survey [Vernon et al., 2007] also mentions Weng [2004] in support of the claim that the the system should be a model generator, rather than a model fitter.
3. Although simulated, a robotic agent can be physically constructed so it can explore a real environment.
4. The space of actions and observations is specified, according to the environment, at the compilation time. We have implemented also a simple exploration preference. Other reflexes can be readily implemented.
5. The reinforcement learning framework provides an excellent way how to implement a value system. The system should have means for adaptation. As mentioned before, it has a close relation to the dopaminergic signalling system in the human reward system.

Chapter 4

A framework for reinforcement learning with abstraction

In this chapter we present our proposal of a framework for reinforcement learning, which is capable of generalization, based on principles outlined in the previous chapter. In the first half of this chapter we will present theoretical arguments for this construction, which guided us in the framework design. Then we will present the framework and discuss possible instantiations of its underlying components.

4.1 Theoretical principles for a rational agent construction

4.1.1 Sensory preprocessing and filtering, sensory uncertainty

In any physical agent, sensors are not perfect. Various factors contribute to the noise which influences the outcome of the sensor measurement. The sensory input can be preprocessed and filtered to some extent. The main algorithmic core of the agent can be saved from many technical details of preprocessing, for example, the algorithm can directly get coordinates of some detected object from the graphic module which specializes at the low-level detection. Then, the main algorithm does not have to include graphic processing. Nevertheless, if the actions in environment are stochastic, it is necessary to incorporate the notion of uncertainty to the core algorithm, because the uncertainty cannot be removed by technical means.

4.1.2 Kolmogorov complexity

Definition 17. Let M be a Turing machine. If running M with input w produces the output x, then the concatenation of the machine code and the input string $\langle M \rangle \cdot w$ is the description of the string x. If this description has minimal length, then the length of this description is called the Kolmogorov complexity of string x:

$$K(x) = \min_{M(w)=x} |\langle M \rangle \cdot w|$$

The Kolmogorov complexity does not depend on a specific computation mechanism, up to a constant:

Theorem 3. *If K_1 and K_2 are the complexity functions relative to description languages L_1 and L_2, then there is a constant c such that*

$$\forall s \ |K_1(s) - K_2(s)| \leq c.$$

The Kolmogorov complexity is a uncomputable function.

Incompressible strings

From the Dirichlet principle, incompressible strings s such that $K(s) \geq |s| - c$ exist. These correspond to the intuitive concept of "randomness".

4.1.3 Hutter Prize: Compressing Wikipedia

The Hutter Prize has been awarded by the scientist Marcus Hutter since 2006. The prize is awarded to a scientist who submits a compression and decompression program for a specific version of English Wikipedia text dump. The sum of the sizes of compressed file and decompressor must be less than 99% of the previous compression record, yielding 500 euro for each 1%. The decompression must run less than 10 hours on a 2 GHz Pentium 4 requiring at most 1 GB memory.

The organizers hope to motivate research in the field of data compression, because they are convinced that the data compression and artificial intelligence are connected, and that the compression of the natural language (of which the dump consists) is equivalent to passing the Turing test: Understanding the text obviously can help its compression.

4.1.4 Algorithmic (Solomonoff) probability

Algorithmic probability is based on the following two principles:

- Epicurus' principle: keep all hypotheses that are consistent with the data
- Occam's razor: entities should not be multiplied beyond necessity

We may reformulate the Occam's razor in more formal terms using Universal Turing machines.

$$m(x) \; := \sum_{p \,:\, U(p)=x} 2^{-|p|}$$

The sum goes over all halting programs p for which a Universal Turing machine U outputs the string x.

One can see that the maximum term in the summation is the term for the minimal program $2^{-K(x)}$. Levin proved, that the other direction also holds: $-\log\, m(x) \; = \; K(x) + O(1)$ and thus $m(x) = \Theta(2^{-K(x)})$.

Example 3. How to determine the Kolmogorov complexity of a language, accepted by a final automaton? A Turing machine, which has the head equivalent with the automaton – with the addition of head movement to the right after each step – accepts the same language. Thus, the Kolmogorov complexity is equal to or less than the description length of this Turing machine.

It can be lower, though: Let us suppose that we have a finite automaton accepting all prime numbers smaller than 10^{100}; this is a finite set. This set cannot be encoded more efficiently, than by enumerating all numbers – the automata will be similar to a so-called "trie" – prefix tree. However, it is possible to write a short program, which accepts these prime numbers. If we wanted to write it in Turing machine (instead of using some high-level programming language), we would need to implement mathematic operations (which can be a little cumbersome). Even then, this program would be much smaller than the description length of the automaton.

This example shows that a stronger formalism can yield a better result from the description length point of view.

4.1.5 Minimum description length principle

The minimum description length (MDL) principle was introduced by Rissanen [1978]. Unlike Kolmogorov complexity, it restricts the set of allowed codes in order to make it possible (computable) to find the shortest codelength of the data. It also chooses a code that is reasonably efficient with respect to the data available.

The philosophy is based on two-part encoding: First, we encode the hypothesis H, using $L(H)$ bits. Second, we encode the data D using the assumption of hypothesis H, using $L(D|H)$ bits. The best model is the one which minimizes the sum

$$L(H) + L(D|H). \tag{4.1}$$

One can view H as the useful information, and $L(D|H)$ as an "accidental" information – for example, noise. If we decide to code only H without encoding the "corrections" necessary to fully describe D, we have the lossy compression. The MDL approach is that we take the hypothesis for which the sum is the lowest, but that does not guarantee, that there is no other hypothesis which could be a better explanation – we are only unable to find with available resources. For example, take a stream of cryptographically generated sequence. It appears random, therefore, any practically feasible approach would consider it as noise. However, if we are somehow able to guess (or are given) the cryptographic key, we may switch to a better hypothesis which fully explains the stream as not a random, but deterministic and predictable stream of data.

For the purpose of this work, we assume that there is no noise in observation and no stochastic results of actions. Therefore we can take a simpler approach, requiring that the model fully and completely explain the accumulated experience.

4.2 Inductive bias

A typical task of machine learning is to find a regularity from the supplied data, to find out how the data relate to one another and to predict future data, even in situations that previously have not been observed.

It is necessary to make further assumptions about the data. Otherwise, the unobserved situations could be arbitrary. The choice of which prior knowledge is specified is called the inductive bias, first coined by Mitchell [1980]. The designer decides about the constraints which are used during learning. Sometimes the bias results directly from the used algorithm or method, sometimes it is even not formalized (it is defined only by functioning of the algorithm).

Example 4. Suppose the series $1, 2, 3, \ldots$. The next following number can be guessed only if an assumption is made. Aside from the obvious answer 4, we can also claim that the next number is 47 because the sequence are denominators of continued fraction convergents to $\sqrt{267}$ (cf. the sequence A041501 in Sloane [2011]).

In fact, if we have a sequence of n numbers, we can choose an arbitrary $(n+1)$-th number and say that the numbers are values of a polynomial of a degree at most $n+2$.

This example shows that an assumption about the data must be made. And usually the most logical assumption is that a simpler solution is better.

4.2.1 Components of abstraction

In this section we would like to introduce the term "abstraction". Let us consider some information about the world. This information describes our experience and observations. Is it possible to derive from these data a useful world model? Certainly, to be useful, the model has to be better than a simple collection of data. It can be better in two ways: First, it can be smaller than this collection, so we save the memory and/or computational resources when using the model instead of the collection. Second, it can extrapolate the data to provide us with the information not explicitly contained in the collection. This estimate information is valuable, although it can be imperfect or sometimes wrong. If the action we guess based on the model is on average still better than a blind action in unknown territory, we have gained an advantage.

What makes the model "good", what makes it satisfy these properties? We formulate the principles for the abstraction:

1. finding patterns in existing data (e.g. repetition)
2. extrapolating to unknown data
3. robustness: correction of errors in existing data

The first principle is satisfied if the provided model is smaller than the data. This is also the case for all grammar induction algorithms presented in section 2.7. The second principle is more difficult and it is usually present and expected in some well-known architectures like neural networks [1]. There must always be present an inductive bias in some form, which drives the extrapolation. For example, if the only requirement would be to generate a grammar which produces all (positive) examples, a simple grammar yielding Σ^* language suffices. If the requirement is to produce only positive examples, only the first principle is effective and no generalization occurs. This was the case with Sequitur algorithm and Myhill-Nerode equivalence algorithms.

In a real-world scenario, usually negative examples occur (e.g. punishments or lower rewards). If only positive examples are present, language learning without additional constraints is impossible (this is a well-known Gold problem, Gold et al. [1967]). However, if additional constraints are specified (forming an inductive bias), it is possible to make generalizations also from the set of only positive examples.

Example 5 (Learning from positive examples)**.** In a previous work [Valentín, 2010] we investigated the possibility of creating a learning firewall using a neural network. The network was presented only with positive packets (a "legitimate traffic") and had to learn to filter the "illegitimate traffic".

We made an assumption that the set of negative examples $\mathcal{N}$ can be approximated by a complement of the set of positive examples. It holds that $\mathcal{N} \subseteq \mathcal{P}^C$. Assuming $\mathcal{N} = \mathcal{P}^C$ gives the network a "bad point" for accepting a packet not present in the positive set. Not accepting a positive example gives also a "bad point". Thus the network generalization is held on a useful level– when trying to maintain the best score, overfull generalization is penalized by bad points from the complementary packets. A low generalization is penalized by bad performance on a validation set. The trade-off between validation set performance and "bad points score" was implicit due to chosen network architecture.

As can be seen from the example, sometimes it is necessary to cope with noisy or missing data (packets which are legitimate but did not occur in the input set, and if generated, they could be presented to the network as negative examples earning it "bad points"). Either the method is inherently meant to approximate noisy data – this is the case of neural networks – or an exception must be formed within the architecture (e.g. a special rule for preventing one word from being in the language). Formation of that explicit rule makes the model greater in the description length. Thus if we want to allow noisy data in a symbolic mechanism like grammar induction, we have to introduce some trade-off parametrization

[1]Neural networks have an implicit inductive bias, caused by the architecture and by the training procedure – minimizing the error on the validation set ("early stopping").

between “bad points” and the description length[2].

4.3 Parameters and their biological analogues

We think that parameters, which have to be set (such as error/generalization trade-off or the exploration/exploitation ratio), are analogical to biological parameters of our brains (and our natures). Some of us are more exploratory, some of us prefer known, less risky ways. Many people sacrifice an immediate reward to gain future advantage, some just do not bother with planning and enjoy the life. Some people see patterns everywhere while others are more conservative with their inductions.[3]

Similarly, it is possible to set up agents with different preferences. An evolutionary or a similar mechanism could be used to select the best choice for a given task.

4.4 The world model

4.4.1 Markov decision process

Markov decision process is a mathematical structure suited to describe an environment where an action executed in some state causes transition to another state or states, where the transition is stochastic – the resulting state depends on a probability given by the transition function. The agent also receives a corresponding reward. The transition does not depend on history – the change of the environment is (up to the randomness) fully assumed by the current state.

Formally, if S is the set of states, A the set of actions, transition function $P(a, s, s')$ is the probability of transition from the state s to the state s' while executing the action a, and $R(a, s, s')$ is the reward received by the agent.

[2]In fact, the exact value of the error-correction/description-length parameter is not relevant in a long run. If the ratio is set to 1:1, an exception would be tried once. If the ratio is set to 10:1, an exception would be tried 10 times until accepted as a settled fact. When the performance is possibly infinite, a constant number of bad attempts plays no role. The trade-off may also be set as a ratio of total attempts (or energy spent), here we are able to set a performance not worse than $(1 - \epsilon)$ of the optimal one.

[3]Suppose you are in a hurry of submitting an article or dissertation and your computer crashes. You are trying many times to turn it on, everything is unsuccessful. Then, suddenly, the computer boots. You remember that you were holding the button for five seconds and nervously knocked three times on the chassis. The computer goes on for a while. But an hour later, you need to turn it on again. What will you try? Holding the button for 5 seconds and knocking three times? (Maybe there is a cold solder joint...) Or will you first try something less ridiculous? How long will you try something else until you try the magical combination? And will you try it again (and how many times?) if it does not work for the first time?

In the case that the agent can only partially observe the state s, comes into consideration the probability $\Omega(o, s', a)$ of getting the observation $o \in O$, if a transition to the state s' occurs.

4.4.2 Likelihood of the model

Definition 18 (Likelihood of the model)**.** Let be given a series of observations, actions and rewards $P = o_1, a_1, o_2, r_2, a_2, o_3, r_3, a_3, \ldots, a_{n-1}, o_n, r_n$, where o_i is an observation, r_i is a reward, and a_i is an action at time i. Let $M = (S, A, O, T, \Omega, R)$ be a partially observable Markov decision process. The likelihood of the model is

$$B(M) = \max_{s \in S^n} \prod_{t=1}^{n-1} P(a_t, s_t, s_{t+1}) \cdot \Omega(o_{t+1}, s_{t+1}, a_t)$$

From all possible models we could theoretically choose the most likely model (if we disregard the computational intractability). However, this model might be too complicated. In accordance with Occams's razor principle, it might be better to choose a less likely, but a simpler model. Here we would have to specify the trade-off between the simplicity and the likelihood.

In practical applications, this formulation of the problem is too wide. In the next section, we limit ourselves to a simpler case, where the transitions and observations are deterministic.

4.5 Motivation and biological plausibility

Our framework consists of three components: reinforcement learning, abstraction, and (optionally) metaplanning. Why have we chosen these components? Why do we think they will be a contribution? Why can they succeed in tasks better than other architectures?

In Chapter 1, we described reinforcement learning and showed the biological and theoretical motivation for it. From the theoretical point of view, reinforcement learning can be understood as a general approach to solve a generic problem. "Reinforcement learning encompasses all of artificial intelligence: An agent is placed in an environment and must learn to behave successfully therein" [Russell et al., 2010].

In Chapter 2, we explained the philosophy behind the minimum description length principle. This principle sets out a general constraint to world modelling. However, we have also mentioned the problem of uncomputability. Thus the constraint has to be relaxed: a pseudo-minimal model is also acceptable.

Having a few possible (pseudo-minimal) models of the world, how should an agent proceed? If a decision has to be made immediately, the agent should proceed according to the best (minimal in description length) model. However, if there is a time for exploration, the agent could first go and investigate the possibility of other possible models. Spending little energy and gaining some information can yield an advantage. In what way and to what extent should the agent explore the possibilities? This is the question of meta-planning. Here we again advocate the Minimal Description Length principle, but in a relaxed way: The best model should be considered most likely, but the second (third-, and so on) model should also be investigated. Of course, the best way to decide between them is to explore the differences between them, to see, if there is a reason to consider the other model in comparison with the best model. This could be compared to a scientific method: The scientists try to create an experiment which could distinguish between two possible worlds, which could falsify the hypothesis.

4.6 Computational and interaction complexity

It has been proved, that the dynamic programming will reach an optimal result [Sutton and Barto, 1998]. The computational complexity is not very good, however, once the computation ends, the agent performs optimally. In contrast with this, the Monte Carlo or TD-learning methods do not perform time-consuming computations, but they use their interaction with the environment to estimate the value of the states and actions. This means that not every interaction in the real environment is optimal.

When we try to design a new method for reinforcement learning, we think it is a good strategy to sacrifice any computational resources to ensure the real performance of the agent. Of course, this can mean the agent can be terribly slow. However, the cost of the interaction with the real environment can be large in comparison with simulated computations. We can buy a better computer or try to improve the algorithm, but we cannot change the speed of the real environment. In other words, it is better to solve the task in a longer time than not to solve it at all.

4.7 Framework description

The general framework (see Fig. 4.1) can be laid out in a few steps, which are repeated in a cycle:

1. Perception comes from sensors and is preprocessed and filtered.
2. Sensory input is saved into the history.
3. Immediately after each sensory input, or at pre-defined time-steps (depending on computational requirements) the abstraction module starts.
4. Abstraction module prepares MDL model(s) of the world.
5. The metaplanning module takes models and computes differences, assigns an internal reward to the states which are able to differentiate between models ("systematic exploration").
6. Reinforcement module takes (the best) MDL world model and decides (via dynamic programming) the next step (taking the internal reward of some states into account).
7. The action is sent to the motor output.

4.8 Components of the framework

Sensors

Sensors for the agent can be specified as necessary. The domain – possible values of observations – has to be specified for each sensor. If the perception is missing, a *null* value can be used. The sensory input can be continuous, however, necessary adjustment for the abstraction module must be made, or the input can be discretized in the preprocessing module.

Sensory preprocessing

All technicalities of the sensory processing should be made here, such as the calibration or graphical processing.

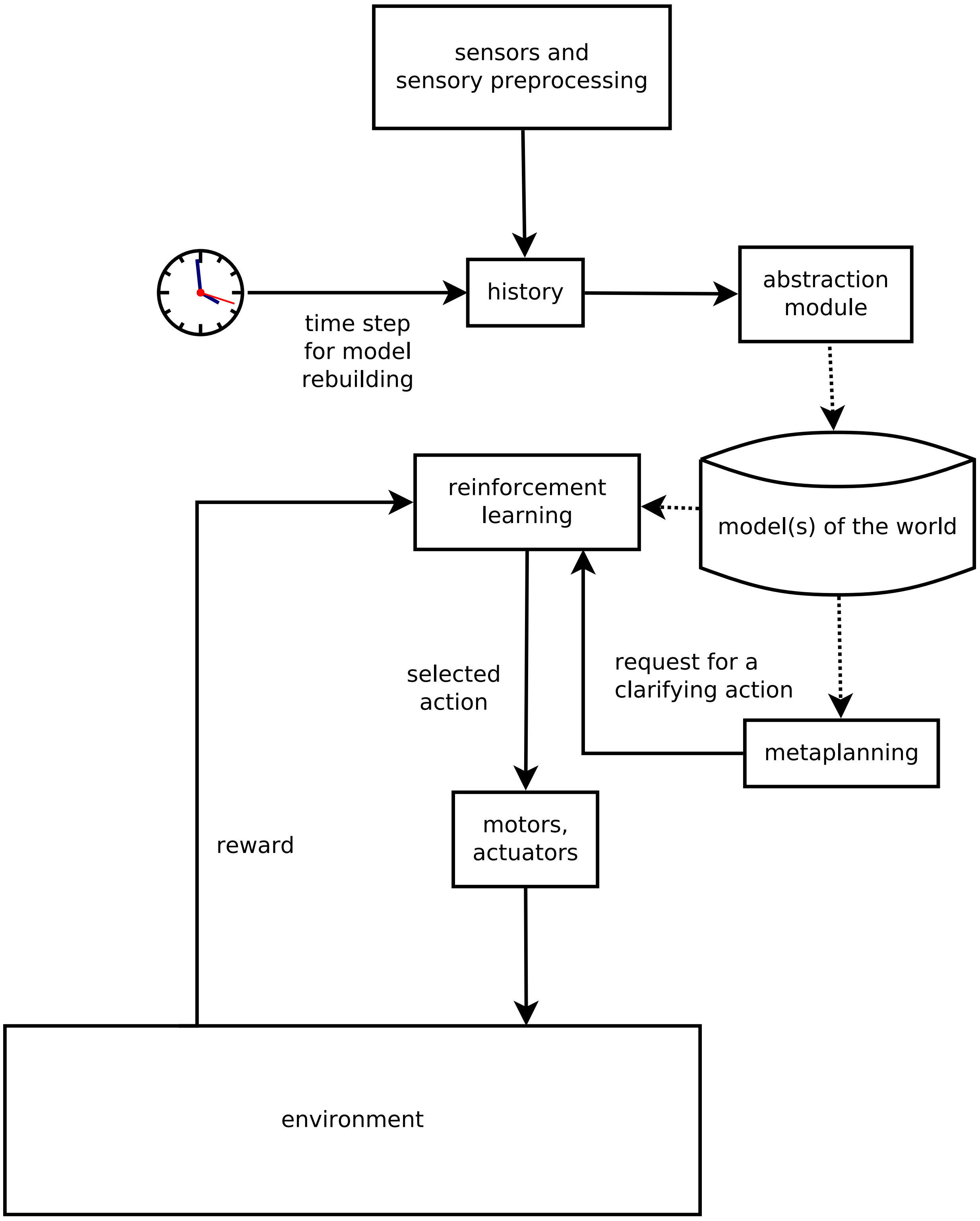


Figure 4.1: Components of the proposed framework and their connections

History

This module saves all actions, observations, and rewards together with time. The whole history as a sequence is used by the abstraction module.

Abstraction module

This is the most important module. It takes the history and tries to induce a world model which can explain the agent's experience at best. For this purpose, we suggest to use grammar induction or automaton inference.

The expected result of this module is a world model – to be technically precise, a machine representing this model. If the meta-planning module is used, the result is a set of the models, rated by their score (description length + error score).

If it is necessary to incorporate stochastic knowledge, a probabilistic automaton [Rabin, 1963] could be produced. However, there is no feasible algorithm available which would generate a probabilistic automaton directly. An option is to generate a non-deterministic automaton, to convert it to a probabilistic automaton, and recompute the probabilities using one of the known algorithms [Ron et al., 1994].

The generated model does not have to be optimal, in case of a strong formalism it is not even possible. Also a pseudo-minimal (best found) model is useful. The module could use some heuristic to generate the model, such as those described in Section 3.2. It could also use the previous model and try to modify it to fit the new knowledge (similar to the state-splitting algorithms).

Reinforcement learning module

This module uses the precomputed world model to infer the next best action. This can be accomplished for example by simple dynamic programming algorithm described in Section 1.5.1.

If the time-tick for re-generation of world model is longer, this module can make ad-hoc modifications into the existing model.

Metaplanning module

This module takes the set of proposed models from the abstraction module and tries to navigate the agent so as to decide, which of the modules is the correct one. This enables a systematic exploration of the environment. The module computes, which sequence of actions provides a difference between the models.

If the models are finite-state automata, it is possible to find this sequence in a systematic way. For automata M_1 and M_2 it is possible to find the language which is the difference between the two generated languages: $L_{\text{diff}} = L(M_1) - L(M_2)$. This can be done by creating an automaton accepting the difference (for the construction, see Theorem 4.10 in Hopcroft and Ullman [2001]) and finding a word accepted by the automaton. The word represents the sequence which has to be taken. The metaplanning module can influence the action values in order to execute the sequence, or make it more preferable.

If a stronger formalism is used, the difference cannot be computed (it is uncomputable for CF languages). However, the metaplanning module can make a look-ahead few steps in order to see if a difference occurs. If in n steps the predictions by machines M_1 and M_2 differ, these n steps can be taken in order to decide between M_1 and M_2. Again, the metaplanning module can make these steps more preferable in order to encourage the exploration.

4.9 Expected properties of the framework

An agent constructed by the framework will explore the world in a systematic way and optimize its actions to earn the reward. During the exploration it can create a world model, which can supply more knowledge than the knowledge explicitly contained in the observations. For example, if a sequence of actions must be executed in order to receive the reward, the execution of the first few actions may not have any effect in the environment. However, the internal state of the derived model records these changes and effectively predicts the necessary actions and the reward.

Of course, the agent's first steps are random and unsystematic. However, as the time proceeds and the agent explores the environment, the world model should become closer to the reality. If a sufficiently strong formalism is used, the agent may derive its own concepts, which have not been explicitly formulated.

In doubts, the agent can employ two or more models, and take the adequate actions in order to resolve his doubts. This enables the agent to systematically examine the world.

Chapter 5

Implementation

In this chapter, we present our prototypical implementation of the Reinforcement Learning with Abstraction (RLA) framework presented in Chapter 4. We present chosen instantiations of the underlying components and motivations behind the decisions. We will also present a prototypical task, on which we were testing our implementation. The prototypical task – the maze problem – follows the general properties outlined in the previous chapter, however, it is somewhat simpler. The reader must be cautioned that the maze problem is meant to be an instantiation of a generic problem; therefore it is not possible to solve it using the standard maze traversal methods or positioning methods like SLAM (Simultaneous localization and mapping, Stachniss et al. [2011]), although it can seem so at the first glance. A detailed description and discussion of the task is presented in the following two sections.

5.1 Goal of the implementation

Almost all existing approaches of reinforcement learning are based on an implicit definition of the state space. The space is usually defined by the range of possible observations. It can be discrete or continuous.

For example, in tic-tac-toe game the agent can observe the board and the markings placed there. All possible board layouts create the state space. Some layouts (such as four X marks and only one O mark) may turn out to be unreachable during training, or they can be already removed from the state space during the agent design.

What if the observation cannot cover the whole space of possibilities given by the environment? For example, if we consider the problem of navigation in the maze: If the coordinates (x, y) are observable, they give the full account of agent's location in the maze and they allow the learning methods to compute the state values and action values in the respective states (locations). However, if the agent can observe only the surrounding (say, his location and 4 neighboring cells) and does not know in advance the maze dimensions, how should be the state space defined? There can be a number of cells with the same configuration of neighboring cells and walls, but the agent gets the same observation in all of them.

The goal is to create an agent, which is able to create a world model from his observations and actions. This model allows the possibility to distinguish states with the same observation.

Moreover, this approach is also useful when the observation space equals the state space, but it is possible to find interesting and useful links between the states. We do not know these links in advance or are not willing to enter this knowledge to the agent because it could be a laborious task.

5.2 The maze problem

Suppose we have a labyrinth, that the agent does not know in advance. How to traverse this maze and create its map?

Of course, there exist standard approaches to solve this task, with the assumption that it is possible to place a marker into the environment in order to remember that we have visited a concrete cell (Trémaux algorithm, Tarry's traversal algorithm, and so on).

In our case it is not possible to place markers nor is it possible to have the coordinates. Also we do not want to give the agent an explicit information that he is in a maze of a specific type. We demand that he fulfills this task without this information, what will also demonstrate his ability to solve even more general problems. In other words, we want the same algorithm to be able to traverse the maze, even if the maze contains teleports or one-way doors. The only information available to the agent is the observation from the predefined set O and the agent can take actions from the set A. The observation can provide information about the agent's surrounding. The actions or observations are atomic – they do not carry any type of information such as coordinates or direction. We can use the same agent in a 2D or 3D maze, if we change the size of the sets A and O appropriately.

Theorem 4 (Zero-marker maze traversal)**.** *It is impossible to traverse a maze, if the agent* *cannot place markers into the maze [Dudek et al., 1991].*

Proof. The impossibility follows from the fact that two regular graphs with the same degree cannot be distinguished for the agent (For example a triangle and a square, a graph with three or four vertices connected into a cycle, respectively. Each vortex has two edges, so it is impossible to distinguish them if the observation does not contain additional information.) □

Notwithstanding this result, it makes sense to solve the maze problem from the reinforcement learning point of view. If two word models are possible (i.e. both models fit the agent's experience), and we are not able to distinguish them, it is wise to suppose the simplest model. If the observations begin to disagree with our expectations, we are able to select another model.

5.3 Prototypical problem: Finding the model of the maze

If we assume a simpler case that the environment is deterministic, we can separate world models according to whether they satisfy the observation/action sequence, i.e. according to whether the above-defined likelihood of the model is $B(M) = 1$ or 0.

From the models which satisfy this condition, we would like to select the simplest model (with the smallest number of states). This is our selection of the *inductive bias* (Section 4.2), which is necessary to overcome the indeterminacy.

We argue that the best method is the grammatical induction (or automaton inference, depending on the formalism, ranging from finite-state automaton, through the push-down automaton, to the Turing machine). If we choose the finite-state automaton (a regular deterministic grammar), the link to the Markovian model is obvious: the states of the automaton (the grammar nonterminals) represent the states of the Markov process. The actions in the Markovian process are the input symbols (terminals) for the automaton (or grammar). The detailed approach is outlined in the following section.

For a stronger formalism than finite automata it is necessary to look at the resulting automaton (say, a Turing machine) as an oraculum, which responds to all questions regarding implicitly represented Markov model, i.e. to what state leads the action, what observation and reward the agent gets in a specific state, and so on. The measured complexity is the complexity of the machine (description length), not of the Markov model –

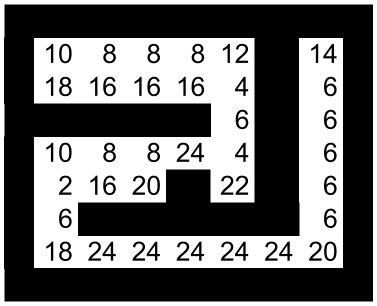


Figure 5.1: The maze. The numbers represent the observations received by the agent.

it is not necessary to represent the Markov model explicitly. The arguments and possible advantages and disadvantages of a stronger formalism are presented in the discussion.

5.3.1 A simple maze instance

The maze can look like the one in Fig. 5.1. The perception of the agent is limited to 4-bit information about the cell and four neighboring cells to the left, right, up, down (whether the cell contains a wall or is free) from the least important bit to the most important one.

5.4 Algorithm

The goal is to find (the minimal) finite automaton (Markovian process) which would satisfy the observation/action sequence $P = o_1, a_1, o_2, r_2, a_2, o_3, r_3, a_3, \ldots, a_{n-1}, o_n, r_n$.

5.4.1 Model inference via automaton/grammar inference

How is it possible to find a model using the mechanism of formal languages? Let us suppose we have already implemented a method for grammar inference. The following example illustrates, how this mechanism could be modified to model inference.

Example 6 (Model vs. automaton)**.** The prefix subsequences $a_1, \ldots, a_k$ where $k \leq n$, which lead to a pre-selected observation o (i.e. it holds $o = o_k$) are considered to be the words of the language L_o. A regular grammar (finite automaton) is created for the language L_o. By a small modification it is possible to create automata for each of the languages L_o (for all possible observations $o \in O$), where the automata are equivalent, up to the position of the accepting states. Edges and states of the automata represent directly the states and transitions in the Markov model, positions of the accepting states in the automaton accepting the language L_o represent those states in the Markov model, which provide the observation o.

5.4.2 Unsuccessful generalization using Myhill-Nerode equivalence

We tried to use the method of grammatical induction based on Myhill-Nerode equivalence, which we created (described briefly in Section 2.7.3 and in more detail in Malý [2011]). Unfortunately, this method turned to be unsatisfactory. The reason is that this method produces a grammar/minimal automaton accepting only the given set of words. This automaton has too many states. The inability to accept other words diminishes the generalization ability (see Section 4.2.1).

5.4.3 Solution using the SAT solver

We decided to find the minimal model by formalizing the constraints for the model using logical formulas, and letting a SAT solver to solve these formulas. A SAT solver is a program, which decides whether there exists a satisfying valuation for a logical formula, and if there is such a valuation, finds the values for the variables.

Formalization

Let us suppose that the model has j states numbered $0, 1, \ldots, j-1$ and, without loss of generality, it starts in the state number 0. Let the number of actions be $|A| = k$, the number of the possible observations $|O| = l$.

Let actions and observations be numbered from 0 to k and l, respectively. For the sake of simplicity, we disregard the rewards r.

The fact that in the state s the agent gets the observation o is written by the predicate $obs(o, s)$. The fact that the action a transfers the environment from the state s to the state s' is denoted by the predicate $tr(s, a, s')$. The fact that in time t the environment is in the state s is denoted by $pos(t, s)$.

We have the following constraints. For the predicate obs:

In one state we have at most one observation: $\forall s \in \mathcal{S} : \forall o, o' \in \mathcal{O}, o \neq o' : \neg obs(o, s) \vee \neg obs(o', s)$

In one state we have at least one observation: $\forall s \in \mathcal{S} : \bigvee_{o \in \mathcal{O}} obs(o, s)$

Similarly, for the predicate tr it holds:

$\forall s, s', s'' \in \mathcal{S}, a \in \mathcal{A}, s' \neq s'' : \neg tr(s, a, s') \vee \neg tr(s, a, s'')$

$\forall s \in \mathcal{S}, a \in \mathcal{A} : \bigvee_{s' \in \mathcal{S}} tr(s, a, s')$

And for the predicate pos:

$\forall t, 0 \leq t \leq T, s, s' \in \mathcal{S} : \neg pos(t, s) \vee \neg pos(t, s')$

$\forall t, 0 \leq t \leq T : \bigvee_{s \in \mathcal{S}} pos(t, s)$

The constraints that the model satisfies the observations are:

$\forall 0 \leq t \leq T, s \in \mathcal{S} : obs(s, o_t) \vee \neg pos(t, s)$

$\forall 0 \leq t \leq T - 1, s, s' \in \mathcal{S} : tr(s, a_t, s') \vee \neg pos(t, s) \vee \neg pos(t + 1, s')$

The assumption that the automaton begins in the state 0 is given by stating that $pos(0, 0)$.

Search for the minimal model

In the formalization above, we assumed that we know the number of states (j). However, we do not know this number in advance. Therefore, we must invoke the SAT solver multiple times using different values. This can be accomplished for example by a binary search for the minimal j (cutting the interval in halves, until an unsatisfiable result for $j - 1$ and a satisfiable result for j is found). However, during experiments we realized that

1. The SAT solver usually finished in a short time (usually less than 0.5 second) for satisfiable instances and in a long time (sometimes $>$ 2 min.) for unsatisfiable instances.

2. The number of states needed for the automaton is monotonic (a new visited cell may increase the number of states necessary to represent the world, but it cannot decrease).

3. As mentioned in Section 4.8, the abstraction module does not have to find strictly the minimal model. Also a pseudo-minimal model is acceptable.

Therefore, we decided to use an iterative solution, beginning with the number of necessary states set to the value 1, and increasing this value for each unsuccessful or too long SAT solver run (probably unsatisfiable formula).

Unknown actions and unknown states

The model found by the described algorithm does not incorporate the uncertainty – it does not mark unknown actions, and contains only visited states. Unexecuted actions are unbound in the transformed logical formula and it is up to the SAT solver to decide how they will be assigned.

However, this is easily resolved by a simple iteration over the history. The respective transition for each executed action is marked as known. For each unknown action, a (new) unknown state is created (see Fig. 5.2 for illustration). The method is able to distinguish between two triples of states having the same observation (three cells marked with the number 8 and three cells marked by 16 in top left part of the maze, compare 5.2 with 5.1).

5.4.4 Algorithm overview

The algorithm overview is given in Fig. 5.3. Basically, the agent runs in a "perceive-learn-act" loop. In the "perceive" and "act" parts the agent interacts with the environment. The "learn" part consists of recording the experience, creating a world model, and computing the best action according to the newly created model. Creation of the world model consists of a loop (the inner loop), in which the the SAT solver is iteratively invoked in attempt to find a solution of a formula, containing a transcript of agent's experience transformed into logical propositions.

5.4.5 Policy

The reinforcement module is independent from the abstraction module. This allows a free choice of abstraction method or the policy for reinforcement learning. We tested two policies. First, the agent was run with a random policy. The purpose was mainly to test the program and to test the performance of the abstraction module.

After this first experiment, we have tried a greedy policy which favorized the unknown actions to encourage the maze exploration.

5.5 Results and discussion

We have run the implemented agent in the maze. The agent gradually extended his world model, adding necessary states and keeping the model at the minimal (or pseudo-minimal) level. The development during the first 23 time steps is shown in Appendix A.

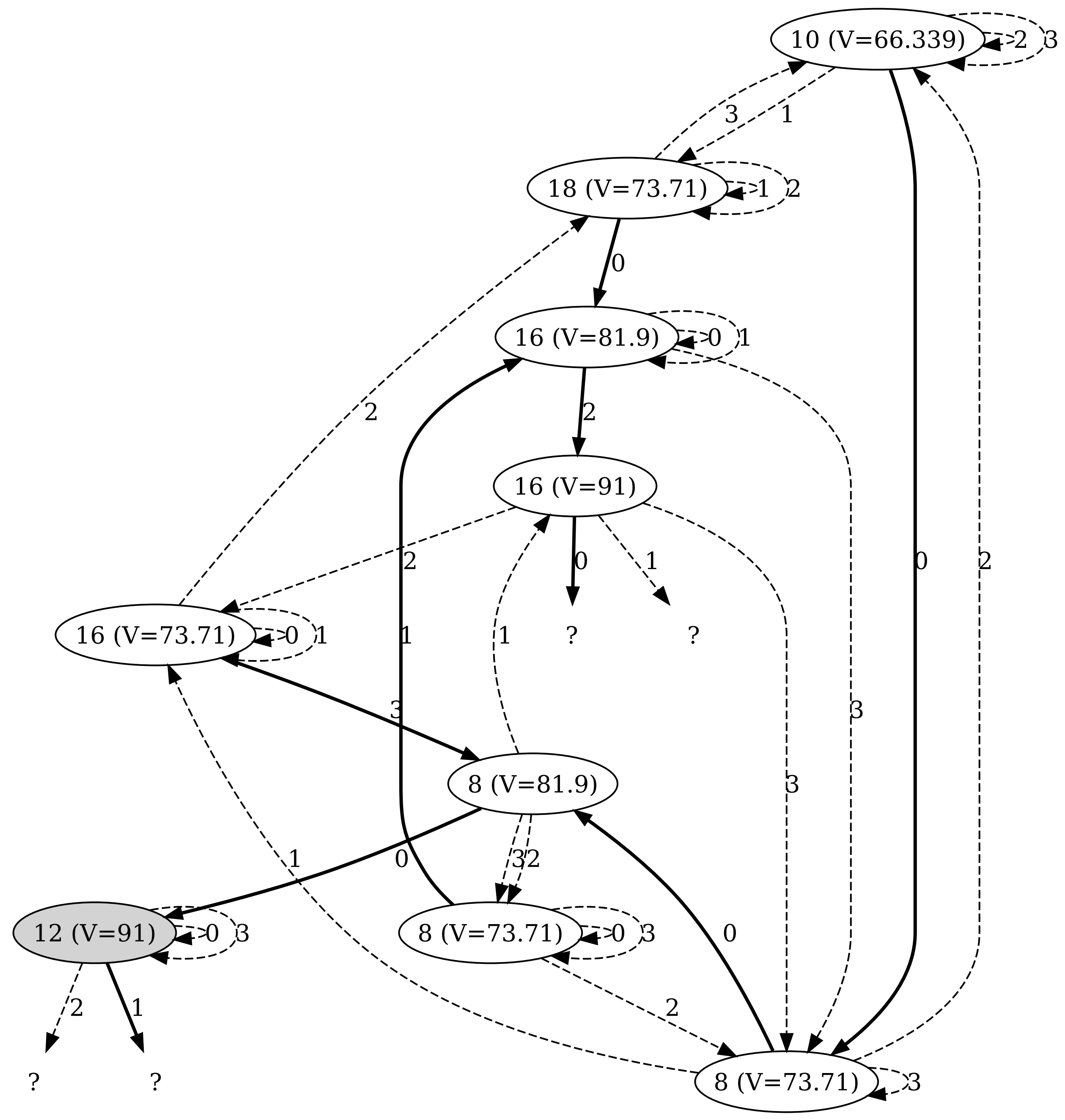


Figure 5.2: An example of generated model (the left top part of the maze).

```
number_of_states = 1;
while(true)
{
        observation = getObservation();
    reward = getReward();
    history.append(observation, reward);

    while(true)
    {
            formula=transformToCNFFormula(number_of_states,history);
        if ( invokeSATSolver(formula,timelimit) == SAT)
                break;
        else //UNSAT or INDET (timelimit exceeded)
                number_of_states++;
    }

    model = extractSATSolution();

    solveDynamicProgramming(model);

    //initial state is 0 by definition (without loss of generality)
    model.initialize(0);
    for(h in history)
           model.advance(h);//emulate action to get to the next state

    current_state = model.getCurrentState();

    if(debug)
    {
           model.draw();
           model.mark(current_state);
    }

    action = bestAction(model.state(current_state));

    executeAction(action);
}
```

Figure 5.3: Overview of the implementation

The performance of the implementation depends primarily on the SAT solver times, with the time to solve satisfiable instances being less than 0.01 second for 12 states and increasing to about 60 seconds for 30 states.

The main goal was to focus more on the prototype testing and not the performance. Further performance improvements are presented in Section 5.7.

5.5.1 Separate observations for different modalities

In this implementation, we have presented the agent a simple number. The agent now has no prior possibility to know that 2 and 12 have both a wall on the left. It would be possible to split the observational domain into multiple subdomains, i.e. in this case having four 1-bit numbers (observations o_1, o_2, o_3, o_4), so the agent could derive a rule based on a separate part of the information obtained from the environment (without having to use a modular arithmetic or another means). However this requires a stronger formalism and brings no advantage when only the FSA inference is used.

5.6 Flexibility of the algorithm

The framework and the algorithm for reinforcement learning with abstraction (RLA) were proposed so as to be flexible, creating the world model as necessary. Therefore, the algorithm can be used without change for example of a task of traversing a maze with cells labeled with letters – the agent sees a letter on the current cell (see Fig. 5.4), and does not have any other information. The performance was similar as mentioned above. The only modification is to change the number of possible observations to match the number of letters.

Teleports and one-way doors could also be added arbitrarily.

5.6.1 3D-maze traversal

We have used the algorithm also for traversing a three dimensional maze. The only requirement necessary was to modify the number of actions. An example of the traversed maze is in Figure 5.1.

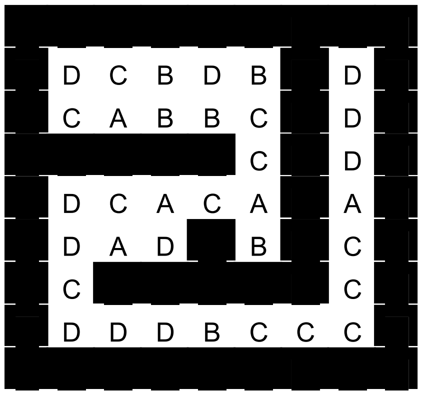


Figure 5.4: A “letter” maze. The letters represent the observations received by the agent.

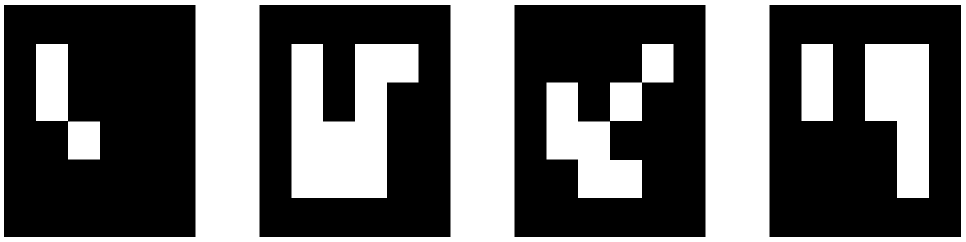


Table 5.1: An example of 3D-maze (27 free cells) traversed by the RLA agent in 199 steps/486 seconds.

5.6.2 Protocol discovery

The principle of operation of RLA is not limited to a maze traversal. It can be used for a variety of real-world tasks, characterized by a limited (partial) observation of the state and a finite-state nature of the environment. A simple example might be kind of “hacker’s” game – a protocol discovery.

Let us suppose that we do not know the exact protocol for sending emails (Simple Mail Transfer Protocol, SMTP), but we know the way how to connect the agent to the SMTP server and let it communicate arbitrary messages. This can be done for example by connecting the output of the agent to an internet socket. Let us also suppose that we are able to detect if the mail was successfully sent, for example, using an external mail client.

The RLA agent is then let to learn the protocol, trying different actions, which evoke different responses from the server. Most of the responses usually generate an error message, but some will lead to a different response and cause a change in an internal state of the server. This change is dictated by the protocol. An example of a simple SMTP communication is shown in Figure 5.5.

We have tested our RLA agent in a simple SMTP environment. We have modified the agent so it can send lines like “HELO”, required by the protocol. [1] The agent was able to learn the SMTP protocol in 113 steps.

5.7 Inclusion of the old model

The speed of the model search can be easily improved. In many cases the agent does not explore any new cell. In this case, the model can remain the same, under the assumption it was correct. Visiting already visited cells should usually not bring any surprise to the agent. Thus, before a new search, the old model can be tested if the latest experience does not contradict it. If it does not, the old model can remain and be considered a valid world model in the current step.

Another situation occurs when the agent explores a new cell. If the old model was correct, it predicted the existence of the new cell. Thus the only modification necessary is

[1] In principle, the agent could be modified to send arbitrary strings. In this case, because of a great space of possible strings, it would be practical not to record every such string as a separate action, but to record only a few past actions or only the actions which lead to a different response. It should be also noted, that some strings would require a high number of attempts if generated by a simple chance. The design of the strings generation is highly protocol-dependent and is more of a “hacking” nature, which we consider to be out of scope of this work.

```
220 server ESMTP ready
HELO server
220 I am glad to meet you
MAIL FROM:<agent@rla.sk>
250 OK
RCPT TO:<user@rla.sk>
250 OK
DATA
354 End data with <CR><LF>.<CR><LF>
From: agent@rla.sk
To: user@rla.sk
Subject: first mail
Dear user,
this is my first mail.

Sincerely,
Your RLA agent
.
250 OK: queued as 71
```

Figure 5.5: An example of SMTP communication. Text in **bold** indicates the response of the server, requests of the client are in *italics*.

to add the new cell as a new state to the old model. This new model (the old model plus one state) becomes a valid model for the current step.

Combining these two tests can save substantial time. Instead of a completely new search from scratch, we first test the old model. Then we try to find a new model, but use the old model as an additional information. This greatly restricts the search space for the SAT solver since almost all variables are fixed, and the new model is found almost immediately.

Only if these two approaches fail, the agent attempts to search for a completely new model.

5.7.1 Experiments with partial use of the old model

We have also experimented with the idea of using parts from the old model as a basis for the new search. In graph terms, we can imagine this as computing a model of one part of the world (for example, an old experience) and a model of a second part (new experience) separately, and then joining the models together. At some point, we need to link states from the first model to the states of the second model. It is necessary to find all points where the two models join together. If one of the cells (states) where in reality the two parts of the world join remains unlinked, at the next visit of this cell/state the agent is forced to create a separate model for the experience beyond this point. This would mean that a third model is created, in which we can find an unlinked state, and this situation repeats again and again.

Thus, the two models must be connected perfectly. We decided to combine the process of computing the model for the second part and the process of linking together: we search for a big (joined) model, where the model for the first part is kept as bound variables, and the model for the second part (new, free variables) will link correctly to it. The big model then encompasses the whole experience. The computation time depends mainly on the number of free variables since the old variables are fixed and do not need to be computed. Therefore, the computation time depends on the size of the "new" part plus some additional (but relatively small) number of variables which describe the relations between the "old" and "new" states.

However, this approach was not very successful. The problem is, which part of the old model to use. We have tried to define a threshold for inclusion based on the count and recency of the respective experience. For each state and each edge, we count how many times we visited the corresponding cell, or executed the corresponding action. We also record how recently this happened. This was inspired by the idea of computing models for two (almost) independent parts of the maze separately and then joining these models into one model. However, the results were mixed. For example, in most of the cases inclusion of the old model caused unsatisfiability due to some fundamental "flaw" in the model, or the model has to be restricted (with a high threshold) so much that it did not bring a substantial improvement of time. Moreover, we did not find any useful heuristic to set a static threshold or to compute it from the run variables by a simple formula. Thus the program has to search for the "right" threshold, increasing gradually from zero to maximum, beginning at the original model, then trimming some of the old states and edges, ending with a completely new model. This search itself takes a certain amount of time. At some point the sum of the time spent for the "threshold" models exceeds the time necessary for computing a completely new model. To spend the time in search for a completely new model seems a better investment in this case.

When we decided to stop the process at some point and took the result of this "threshold" approach as a final result, we underwent a risk that the agent would not find a model for a given number of states, even if such a model existed. A small number of such failures should not be a problem – using "pseudo-minimal" model is perfectly in line with philosophy of MDL. However, the failures seemed to be repeated one after another, because the flaw, which occurred in some of the earlier models, was transferred to the following models. We were thus forced to switch from the "threshold approach" to a "complete" search, using a brand new model at some point. That required introducing another parameter – a threshold for when to switchover, what further complicates the heuristic approach. In theory, it could be possible to find a combination of right parameters for selecting which thresholds to try, at what time to interrupt them, and when to switch to "complete" search. But we were not able to find such a good combination, we rather decided to use a more transparent approach, trying only the whole old model, and after a failure immediately to switch and search for a new model from scratch. The heuristic approach was thus greatly simplified – only an appropriate time limit for for the new model search has to be guessed.

5.8 Further experiments with mazes

After initial experiments we found out, that the system was in principle able to derive a world model. However, this was done at some computational cost. This can be reduced by adding ad-hoc rules about the environment, which are known in advance. One can see these rules as a replacement of a good meta-abstraction module: they could be in principle derived by the agent itself, only if he is provided with a good, strong abstraction mechanism.[2] We have decided to use the following rules:

1. It is not possible to move against a wall.
2. When there is not a wall, a movement in that direction takes the agent to a different cell.
3. When it is possible to move from one state to another, then a movement in the opposite direction returns the agent back to the first state.

These rules limit the generality of the agent. However, they greatly improve the speed. For example, the first rule is broken in games like Sokoban, and the third rule is not true in environments with one-way doors.

[2]The inclusion of local knowledge was recommended also by M. Forišek.

Chapter 6

Performance testing and comparison to other approaches

After optimizations presented in Section 5.7, we have a faster implementation of the framework. We would like to answer two questions: What is the speed of the framework? This is necessary to assess whether the implementation is useful and what size of problems it can solve, whether the solution is viable when applied to practical problems. The other but related question is how our approach compares to other approaches. There are many kinds of practical problems. Would it be beneficial to use our framework for this or that particular problem? Or should one use a different method? In what kind of problems is our method superior and where is it not?

6.1 Methodology

We have compared other algorithms for POMDP and three generic (MDP, fully observable world) algorithms with our RLA method. The test were performed on a computer with AMD Athlon$^{\text{TM}}$64 Processor 3500+ and 1GB RAM. For generic algorithms, we used the PyBrain library Schaul et al. [2010], which contains an implementation of many RL methods. We modified the examples supplied with the library to the purpose of our test. These programs are available on the CD medium.

6.1.1 Algorithms for POMDP

We compared our RLA with algorithms outlined in Section 3.2. For each of these, we compared experimental results of the authors with the performance of RLA. As each of the papers uses a different approach and a different test case, so we do in our comparison, running RLA on the same input as did the author(s) of the respective paper.

Lion algorithm

In Lion algorithm (described in Section 3.2.1), when the agent finds an inconsistent state (a state where the obtained reward is different from the expected one) it tries to avoid that state by setting the utility function to zero. Therefore, the algorithm is able to be successful only in a world which contains a path from goal that consists of only unaliased states[1]. This is, however, not true in most of the mazes and therefore the algorithm would be not successful.

Utile Distinction Memory

McCallum [1992] writes that for a maze depicted in Figure 6.1, "UDM consistently learned the optimal policy for this maze in five trials of 500 steps each", i.e. it took 2500 steps. The author does not state the CPU time necessary to perform the trials.

In our experiment, RLA explored the same environment in 35 steps and it took 3.8 seconds.

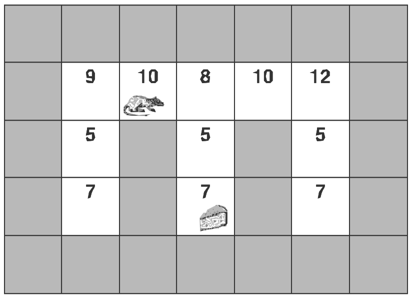


Figure 6.1: The maze used for testing in McCallum [1992]. In this paper, the agent's perception is defined in a similar manner that we described in Section 5.3.1. It is a bit vector of length four whose bits specify whether or not there is a wall to the agent's immediate north, east, south, west. The numbers in the squares are the agent's observations.

[1]To be precise, it could possibly find a way containing aliased states by accident – if the agent is lucky when entering these states for the first time.

6.1.2 Generic algorithms

We have selected three generic RL methods: Q, Q(λ), SARSA (see Sections 1.5.4, 1.5.5, 1.5.6), which were run together with our RLA implementation on a set of mazes. Each method was stopped after it had successfully explored the maze and created a model of the maze. In the case of RLA, the model is explicit.

It should be noted that the generic algorithms were not designed to run in a partially observable world. Therefore, they have been provided a fully observable world – in this sense the test is not "fair", because RLA obtains only a partial observation, however, we think it is a useful comparison. We see that even in this "unfair" test RLA performs quite well. How generic algorithms behave in a partially observable world can be seen in the next subsection.

In the case of the other three generic methods, the exploration is considered to be finished when the value-function does not change significantly in any state. We consider a change of the value-function in a state to be significant if the change leads to a different action to be chosen in that state. In other words, we compute the optimal action in every state, i.e. the action which (presumably) leads to the goal. In fact, this enables us to visualize the implicit model – the computed table of of the value-function – and make it explicit to see the direction in which the agent moves.

▨	▨	▨	▨	▨	▨	▨	▨	▨
▨	→	↓	↓	→	↓	▨	→	▨
▨	→	→	→	→	↓	▨	↑	▨
▨	▨	▨	▨	▨	↓	▨	↑	▨
▨	↓	↓	←	←	←	▨	↑	▨
▨	↓	←	←	▨	↑	▨	↑	▨
▨	↓	▨	▨	▨	▨	▨	↑	▨
▨	→	→	→	→	→	→	↑	▨
▨	▨	▨	▨	▨	▨	▨	▨	▨

▨	▨	▨	▨	▨	▨	▨	▨	▨
▨	0.63	0.67	0.65	0.65	0.70	▨	0.97	▨
▨	0.65	0.69	0.70	0.73	0.76	▨	0.85	▨
▨	▨	▨	▨	▨	0.80	▨	0.94	▨
▨	0.81	0.82	0.82	0.82	0.75	▨	1.00	▨
▨	0.82	0.82	0.76	▨	0.77	▨	0.95	▨
▨	0.82	▨	▨	▨	▨	▨	0.92	▨
▨	0.83	0.84	0.86	0.87	0.88	0.90	0.91	▨
▨	▨	▨	▨	▨	▨	▨	▨	▨

Table 6.1: An example of a model learned by SARSA. At the left side, the arrows indicate the direction of the best action for each cell. On the right, the utilities of the states are shown.

For each run, we track the number of steps in the environment (environment time, Fig. 6.2 and 6.4) and the real time (CPU time needed for the computation, Fig. 6.3 and 6.5). If one method has both times lower, it is clearly superior. If the real time is high, but the environment time is low, we can consider it as a good trade-off. As we explained before:

for some tasks it is better to invest in a CPU-intensive computation rather than to let the agent (for example, a robot) make a number of costly steps in the real environment.

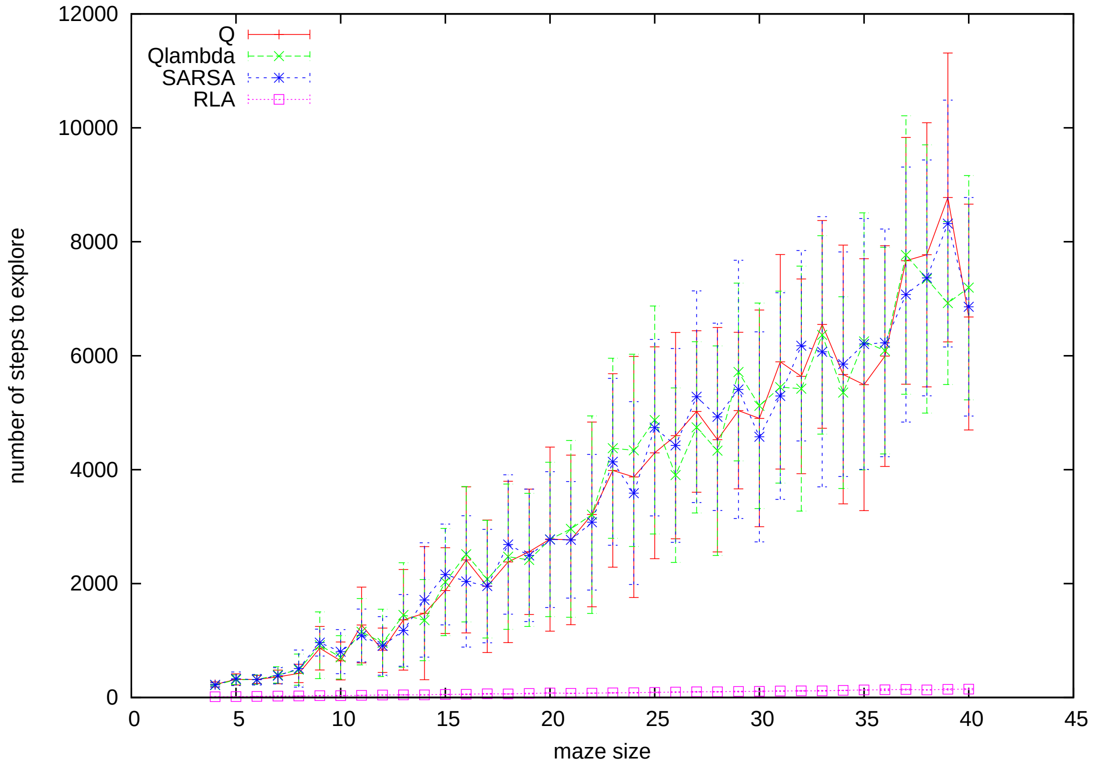


Figure 6.2: The number of steps in environment necessary for SARSA,Q, and Q(λ) to fully explore the world, depending on the size of the maze (the number of free cells). The vertical bars show the standard deviation from 10 runs. We have also added RLA for comparison (a rescaled view of RLA is available in Fig. 6.4).

6.1.3 Discussion of results

We can see that the RLA in comparison with other POMDP algorithms performs significantly better. In comparison with generic algorithms, when these are granted full observation, RLA performs better in terms of the number of steps necessary to explore the environment.

	Q				**Q(λ)**				**SARSA**				**RLA**			
size	**steps**	**dev**	**time**	**dev**	**steps**	**dev**	**time**	**dev**	**steps**	**dev**	**time**	**dev**	**steps**	**dev**	**time**	**dev**
4	228	47	1.2	0.0	218	11	1.2	0.0	223	35	1.2	0.0	13	0	0.3	0.0
5	318	97	1.2	0.0	312	83	1.2	0.0	328	120	1.2	0.1	17	1	0.5	0.0
6	318	70	1.2	0.0	314	83	1.2	0.0	312	93	1.2	0.0	20	1	0.7	0.0
7	360	119	1.2	0.1	395	144	1.3	0.1	382	142	1.3	0.1	24	2	1.1	0.1
8	422	162	1.3	0.1	489	274	1.3	0.1	508	326	1.3	0.1	27	1	1.4	0.1
9	865	383	1.5	0.1	918	586	1.5	0.2	963	237	1.5	0.1	32	3	2.2	0.1
10	642	333	1.4	0.1	705	381	1.4	0.2	804	386	1.4	0.2	33	2	4.5	0.2
11	1272	664	1.6	0.3	1154	582	1.6	0.2	1089	464	1.6	0.2	39	3	7.1	0.4
12	829	390	1.5	0.2	958	591	1.5	0.2	905	515	1.5	0.2	43	3	9.6	0.4
13	1363	884	1.7	0.4	1449	916	1.7	0.4	1178	630	1.6	0.3	47	5	13.0	0.4
14	1479	1168	1.8	0.5	1357	712	1.7	0.3	1713	1004	1.9	0.5	47	1	16.1	0.7
15	1877	753	2.0	0.3	2027	941	2.0	0.4	2160	885	2.1	0.4	53	4	20.6	0.8
16	2415	1282	2.2	0.6	2516	1188	2.2	0.5	2038	1153	2.0	0.5	56	6	25.8	1.5
17	1952	1163	2.0	0.5	2077	1033	2.0	0.5	1957	996	2.0	0.4	64	7	32.3	1.7
18	2380	1416	2.2	0.6	2472	1275	2.2	0.6	2686	1223	2.3	0.6	62	5	35.8	1.3
19	2558	1100	2.3	0.5	2416	1168	2.2	0.5	2494	1163	2.2	0.5	70	7	44.6	2.3
20	2780	1615	2.4	0.8	2775	1355	2.4	0.7	2772	1193	2.4	0.6	78	11	54.1	4.8
21	2766	1487	2.4	0.7	2959	1552	2.5	0.7	2767	1023	2.4	0.5	73	2	57.7	2.6
22	3213	1621	2.7	0.8	3210	1733	2.7	0.9	3077	1191	2.6	0.6	76	3	66.7	2.3
23	3986	1699	3.1	0.8	4375	1581	3.2	0.8	4138	1465	3.1	0.7	82	5	81.8	3.5
24	3872	2117	3.0	1.0	4339	1688	3.2	0.8	3588	1604	2.9	0.8	85	5	90.4	4.1
25	4296	1861	3.2	0.9	4871	2001	3.5	1.0	4738	1549	3.4	0.8	90	9	116.3	11.3
26	4597	1812	3.4	0.9	3903	1532	3.0	0.7	4426	1701	3.3	0.8	95	5	125.5	7.0
27	5021	1418	3.6	0.7	4742	1503	3.5	0.8	5280	1859	3.8	0.9	100	7	164.3	13.9
28	4526	1969	3.5	1.1	4332	1839	3.4	1.0	4927	1643	3.7	0.9	101	6	170.9	8.6
29	5036	1374	3.7	0.7	5714	1560	4.1	0.8	5408	2267	3.9	1.2	105	7	194.1	12.7
30	4900	1901	3.7	1.0	5119	1804	3.8	0.9	4576	1843	3.5	1.0	109	8	226.6	19.1
31	5893	1884	4.2	1.0	5448	1686	4.0	0.9	5292	1813	3.9	1.0	113	6	243.3	12.0
32	5637	1709	4.1	0.9	5422	2150	4.0	1.1	6175	1672	4.4	0.9	117	8	273.5	19.6
33	6549	1823	4.6	1.0	6365	1743	4.5	0.9	6069	2371	4.4	1.3	118	8	302.0	15.4
34	5670	2270	4.1	1.2	5352	1684	4.0	0.9	5851	1972	4.2	1.1	124	3	343.4	14.8
35	5492	2211	4.1	1.2	6252	2255	4.5	1.2	6206	2202	4.4	1.2	132	7	409.8	25.4
36	5993	1937	4.3	1.0	6089	1814	4.4	1.0	6226	1998	4.5	1.1	137	14	443.4	21.4
37	7666	2166	5.4	1.3	7768	2445	5.5	1.4	7074	2238	5.1	1.3	140	10	837.4	214.9
38	7772	2318	5.3	1.2	7347	2356	5.1	1.3	7365	2070	5.1	1.1	134	4	565.7	31.0
39	8778	2536	6.2	1.5	6924	1429	5.1	0.8	8319	2166	6.0	1.3	143	13	637.8	48.8
40	6679	1982	5.0	1.2	7194	1969	5.3	1.2	6859	1918	5.1	1.2	146	11	1035.6	398.1

Table 6.2: Time and number of steps necessary to create a model of a maze of a given size. Three full-observation algorithms Q, Q(λ), and SARSA are compared to the partial-observation RLA method. The table shows averages and standard deviations from 10 runs on 10 different mazes.

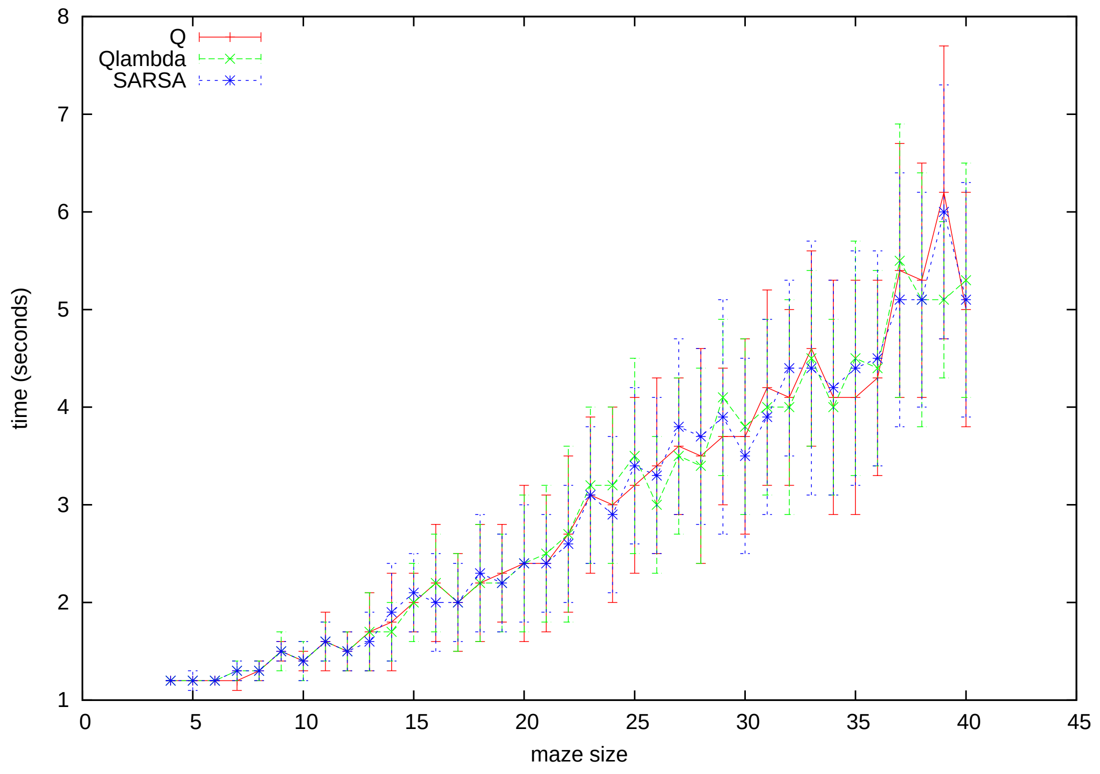


Figure 6.3: The time necessary for SARSA, Q, and Q(λ) to fully explore the world, depending on the size of the maze – the number of free cells. The vertical bars show the standard deviation from 10 runs.

The absolute (processor) time needed by RLA to explore the world grows exponentially. This is expectable, since RLA uses SAT to compute models. With currently available processor powers, an upper limit for feasible use (about 1 hour) would be about 50–60 states.

6.1.4 How do generic algorithms perform in a partially observable environment?

We have modified the code used for exploration in the previous section in such a way that the algorithm receives only a partial observation, 3×3 cells surrounding the agent. This

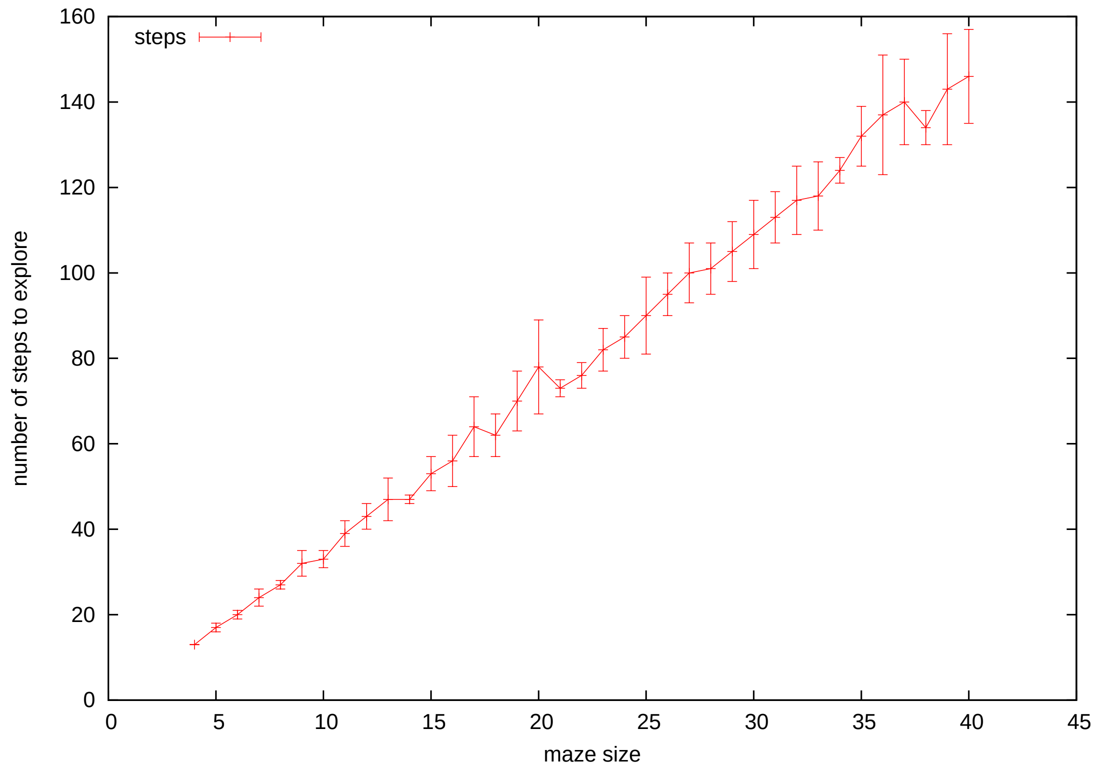


Figure 6.4: The number of steps in environment necessary for RLA to fully explore the world, depending on the size of the maze – the number of free cells. The vertical bars show the standard deviation from 10 runs.

means, that in different states the agent can obtain the same observation. The result is that none of the algorithms was able to create a world model.

These generic algorithms are designed in such a way that they in principle average the reward information received in a state. Thus, if the world contains multiple states with the same observation but of a different utilities, these utility values will be averaged. Unfortunately, the maze contains many perceptually aliased states, which are distributed quite uniformly across the maze, thus having a quite uniform distribution of their utility values. For example, the utility value in a simple maze depends on the distance to the goal. The “model” of a generic algorithm will thus consist of all states having the almost the same utility, equal to the average of utilities of all states plus some noise. The actions

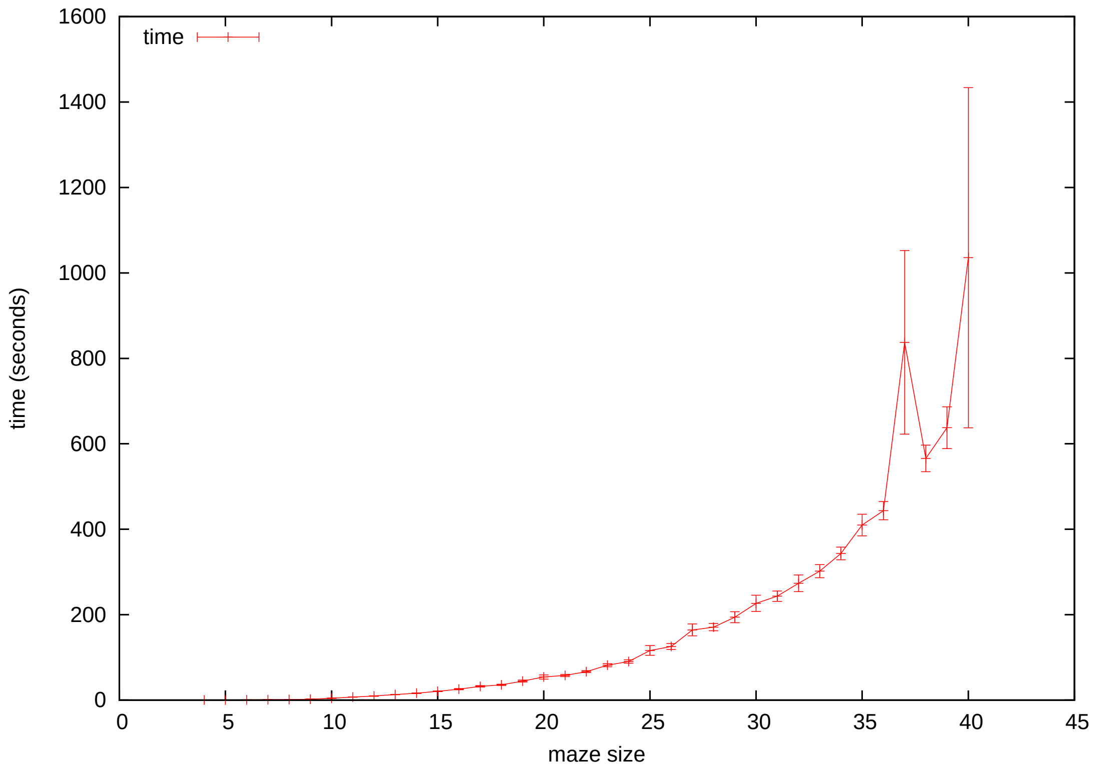


Figure 6.5: The time needed for RLA to fully explore the world, depending on the size of the maze – the number of free cells. The vertical bars show the standard deviation from 10 runs.

of an agent using such a model will be very similar to a random walk.

Chapter 7

Conclusion and future work

7.1 Results

We have described the principles, advantages, and limitations for our framework that enables the agent to create the world model. We have also presented a simple implementation, using a SAT solver for automaton inference. The implementation successfully created a world model representing the maze. The agent was able to distinguish states with the same perceptual information, unlike most of the reinforcement learning methods.

For POMDP maze tasks, the RLA method is faster than other available methods. Even in comparison to generic methods (SARSA, Q learning, Q-lambda) in fully observable MDP tasks, RLA performs significantly better in terms of number of steps in the environment without the requirement for full observation. If the problem is a POMDP, none of the generic methods is available, and the RLA method might be one of few methods capable of solving the problem.

7.2 Future work

It would be possible to use another method of grammar inference or automata inference, for example the ECGI method [Rulot et al., 1989], which produces a regular grammar, genetic algorithm [Javed et al., 2004] or other symbolic techniques [Alquezar, 1997; Rivest and Schapire, 1993]. It would also be possible to modify our algorithm using the SAT solver to find a stronger grammar (e.g. a context-free grammar), or a one- or two-counter automaton. However, the complexity (the number of clauses to solve) would be much greater. It is also necessary to remember the uncomputability limit – two counter automata are equivalent to Turing machines when a specific encoding is used.

7.2.1 Strong formalism

If it was possible to use a stronger method, we suppose that the generalization would be more interesting. For example, let us assume an empty matrix, where the agent can move in an unlimited number of steps to the left, right, up, down, and can put a mark on the cell. Obviously, as the matrix is infinite, the agent cannot have experience with each cell. The question "where will I be from the cell p if I do 3 steps right, 3 steps up, 3 steps left, and 3 steps down" has no answer within his experience. The finite-state automaton cannot provide the answer either, because each cell has to be assigned a separate state. Thus it is unable to generalize the rule that by making this movement, the agent will end up on the same cell.

However, if a stronger method could be used, for example, a two-counter automaton, it is possible to derive a simple automaton which counts the coordinates on its counters. This automaton has only a few states, thus it is more preferable to any finite-state automaton that records each cell separately.

The concept of coordinates, which could be inferred in this way, was not included in the agent's design, but the agent has derived it himself.

We do not have knowledge, whether the technique presented in Fahmy and Roos [1996] for real-time 2-counters automata is able to produce such an automaton.

7.2.2 Two-level grammars

An alternative possibility is to introduce another level of grammar induction on top of the inferred model. The inferred meta-rules could be used for deciding between similar models (of the equal or comparable sizes) and improving the generalization of the model. For example, in the current implementation with finite automaton there is no useful information deduced for the unknown actions, as they constitute free variables for the formula. However, analyzing the derived model could infer this meta-information – that the action "go right" from the state with the wall in the right will result in the same state. This rule may or may not be valid (we could arrange a maze with moving walls – a kind of Sokoban – where this is not true). The metaplanning module would have to take care about the use and verification of these meta-rules.

Two context-free grammars were proven to be Turing complete [Sintzoff, 1967]. Double-step inference of (pseudo-minimal) context-free grammars could be a more effective way how to search for a (pseudo-minimal) Turing machine at once[1]. However this is only a speculation.

7.3 Conclusion

This work is concerned with partially observable problems in reinforcement learning. This class of problems is still open and existing theory provides only partial solutions. A review of existing approaches is given together with a general theoretical background of reinforcement learning and Markov models.

The work proposes a novel framework – Reinforcement Learning with Abstraction. It also contains a prototypical implementation of the RLA method. The implementation is tested on a set of tasks. The results and comparison to another approaches are presented. For a subset of specific problems, namely partially observable problems, the RLA performs significantly better than other available approaches.

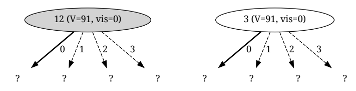
The framework in general is derived theoretically and well-founded. Partially also biological motivations are provided. The future work (Section 7.2) contains some proposals to make the implementation more feasible and/or to gain more generalization power.

[1]Search for a minimal Turing machine is a undecidable problem. A pseudo-minimal (best which we can find in a reasonable time) Turing machine would be, however, also a useful generalization of the problem.

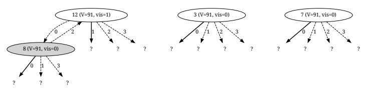
Appendix A

Illustration of model development

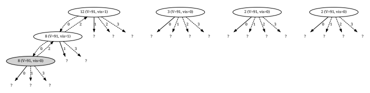
The following figures illustrate the development of agent's world model during exploration of a maze (Fig. 5.1). Ellipses represent states, each state is labelled by the observation and the value (V). The grey ellipse represents the state the agent is in, the bold arrow marks the best action. The dotted arrows mark the actions which were not yet explored (and are only "predicted" by the model).

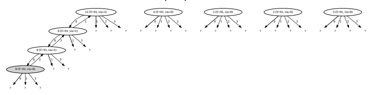


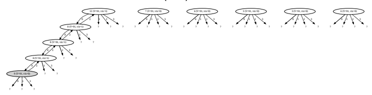
(a) 0

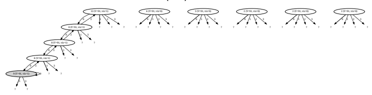


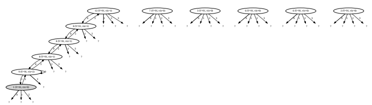
(b) 1

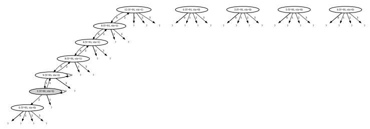

(c) 2

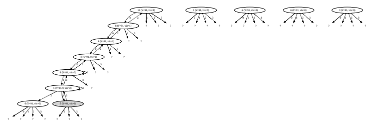

(d) 3

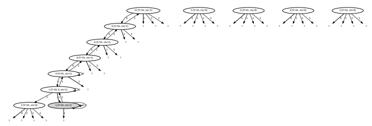

(e) 4

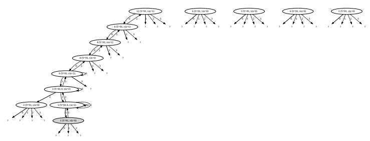

(f) 5

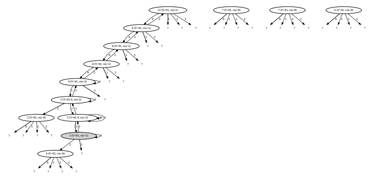

(g) 6

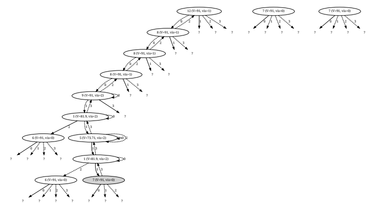

(h) 7

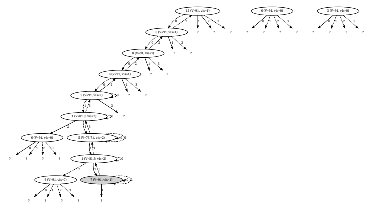

(i) 8


(j) 9


(k) 10


(l) 11


(m) 12


(n) 13

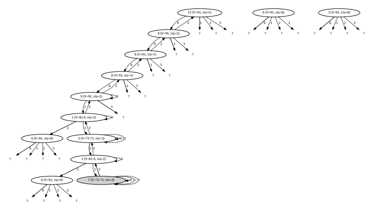

(o) 14

Figure A.1: Development of agent's model during the maze exploration. Steps 0-14.

Appendix B

Source code documentation

Removed in shortened version, can be requested by email.

Bibliography

Alquezar, R. (1997). *Symbolic and connectionist learning techniques for grammatical inference*. PhD thesis, Universitat Politecnica de Catalunya.

Anderson, J. (1996). Act: A simple theory of complex cognition. *American Psychologist*, 51(4):355.

Anderson, J., Matessa, M., and Lebiere, C. (1997). ACT-R: A theory of higher level cognition and its relation to visual attention. *Human-Computer Interaction*, 12(4):439–462.

Arias-Carrión, Ó. and Pöppel, E. (2007). Dopamine, learning, and reward-seeking behavior. *Acta Neurobiologiae Experimentalis*, 67(4):481–488.

Ačová, M. (2002). Implementácia a testovanie vlastností cogitoidu. Diplomová práca. Fakulta matematiky, fyziky a informatiky, Univerzita Komenského.

Babuška, R. and Groen, F. C. (2010). *Interactive Collaborative Information Systems*. Springer Verlag.

Baral, C. (2003). *Knowledge representation, reasoning and declarative problem solving*. Cambridge Univ Press.

Bellman, R. (1957). *Dynamic Programming*. Princeton University Press.

Beran, M. and Wiedermann, J. (2002). Kogitoid – operační systém myslícího počítače. *Vesmír*, 81(10):556–558.

Chapman, D. and Kaelbling, L. (1991). Learning from delayed reinforcement in a complex domain. In *Twelfth International Joint Conference on Artificial Intelligence*.

Chomsky, N. (1956). Three models for the description of language. *IRE Transactions on Information Theory*, 2:113–124.

Couppis, M. and Kennedy, C. (2008). The rewarding effect of aggression is reduced by nucleus accumbens dopamine receptor antagonism in mice. *Psychopharmacology*, 197(3):449–456.

Dubuc, B., e. a. (2012). The brain from top to bottom. Available online at: http://thebrain.mcgill.ca/flash/a/a_03/a_03_cr/a_03_cr_que/a_03_cr_que.html.

Duda, R., Hart, P., and Stork, D. (2001). *Pattern Classification*, volume 2. Wiley-Interscience.

Dudek, G., Jenkin, M., Milios, E., and Wilkes, D. (1991). Robotic exploration as graph construction. *IEEE Transactions on Robotics and Automation*, 7(6):859–865.

Fahmy, A. and Roos, R. (1996). Efficient learning of real time two-counter automata. In *Algorithmic Learning Theory*, pages 113–126. Springer.

Gold, E. et al. (1967). Language identification in the limit. *Information and Control*, 10(5):447–474.

Hazy, T., Frank, M., and O'Reilly, R. (2010). Neural mechanisms of acquired phasic dopamine responses in learning. *Neuroscience & Biobehavioral Reviews*, 34(5):701–720.

Hopcroft, J., Motwani, R., and Ullman, J. (1979). *Introduction to Automata Theory, Languages, and Computation*, volume 3. Addison-wesley Reading, MA.

Hopcroft, J. E. and Ullman, J. D. (2001). *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, second edition.

Hutter, M. (2007). Universal algorithmic intelligence: A mathematical top→down approach. In Goertzel, B. and Pennachin, C., editors, *Artificial General Intelligence*, Cognitive Technologies, pages 227–290. Springer, Berlin.

Javed, F., Bryant, B., Črepinšek, M., Mernik, M., and Sprague, A. (2004). Context-free grammar induction using genetic programming. In *Proceedings of the 42nd annual Southeast regional conference*, pages 404–405. ACM.

Jazykovedný ústav Ľ. Štúra SAV (2009). Frequency Statistics for the prim-4.0, Slovak National Corpus. Available at: http://korpus.juls.savba.sk/stats/prim-4.0/word-lemma/prim-4.0-juls-all-word-frequency.txt.gz.

Krichmar, J. and Edelman, G. (2005). Brain-based devices for the study of nervous systems and the development of intelligent machines. *Artificial Life*, 11(1-2):63–77.

Krichmar, J. and Edelman, G. (2006). Principles underlying the construction of brain-based devices. In *Proceedings of AISB*, volume 6, pages 37–42.

Krichmar, J. and Reeke, G. (2005). The darwin brain-based automata: Synthetic neural models and real-world devices. *Modelling in the Neurosciences: From Biological Systems to Neuromimetic Robotics*, pages 613–638.

Langley, P., Laird, J., and Rogers, S. (2009). Cognitive architectures: Research issues and challenges. *Cognitive Systems Research*, 10(2):141–160.

Lin, L. and Mitchell, T. (1992). Memory approaches to reinforcement learning in non-Markovian domains. Technical Report CMU-CS-92-138, School of Computer Science, Carnagie Mellon University, Pittsburgh, PA.

Malý, M. (2011). Natural Language Processing with Application to Slovak Language. Rigorous thesis, Faculty of Mathematics, Physics and Informatics, Comenius University, Bratislava.

Markov, A. A. (1906). Extension of the law of large numbers to quantities depending on each other. *Bulletin of the Physics-Mathematical Society of Kazan University*, 2.

McCallum, R. (1992). "first results with utile distinction memory for reinforcement learning". Technical Report 446, Department of Computer Science, University of Rochester.

McCallum, R. A. (1993). Overcoming incomplete perception with utile distinction memory. In *In Proceedings of the Tenth International Conference on Machine Learning*, pages 190–196. Morgan Kaufmann.

Michie, D. and Chambers, R. A. (1968). BOXES: An experiment in adaptive control. In Dale, E. and Michie, D., editors, *Machine Intelligence*. Oliver and Boyd, Edinburgh, UK.

Mitchell, T. M. (1980). The need for biases in learning generalizations. Technical Report CBM-TR-117, Rutgers University, Dep. Comput. Sci., New Brunswick, NJ.

Nerode, A. (1958). Linear automaton transformations. *Proceedings of the American Mathematical Society*, 9(4):541–544.

Nevill-Manning, C. and Witten, I. (1997). Identifying hierarchical structure in sequences: A linear-time algorithm. *Arxiv preprint cs/9709102*.

Olds, J. and Milner, P. (1954). Positive reinforcement produced by electrical stimulation of septal area and other regions of rat brain. *Journal of Comparative and Physiological Psychology*, 47(6):419.

Olsson, L., Nehaniv, C., and Polani, D. (2006). From unknown sensors and actuators to actions grounded in sensorimotor perceptions. *Connection Science*, 18(2):121–144.

Rabin, M. (1963). Probabilistic automata. *Information and Control*, 6(3):230–245.

Redgrave, P. and Gurney, K. (2006). The short-latency dopamine signal: a role in discovering novel actions? *Nature Reviews Neuroscience*, 7(12):967–975.

Rissanen, J. (1978). Modeling by shortest data description. *Automatica*, 14(5):465–471.

Rivest, R. and Schapire, R. (1993). Inference of finite automata using homing sequences. *Machine Learning: From Theory to Applications*, pages 51–73.

Ron, D., Singer, Y., and Tishby, N. (1994). Learning probabilistic automata with variable memory length. In *Proceedings of the Seventh Annual Conference on Computational Learning Theory*, COLT '94, pages 35–46, New York, NY, USA. ACM.

Routtenberg, A. (1978). The reward system of the brain. *Scientific American*, 239(5):154–164.

Rulot, H., Prieto, N., and Vidal, E. (1989). Learning accurate finite-state structural models of words through the ECGI algorithm. In *International Conference on Acoustics, Speech, and Signal Processing*, pages 643–646. IEEE.

Rummery, G. A. and Niranjan, M. (1994). On-line Q-learning using connectionist systems. CUED/F-INFENG/TR 166, Cambridge University Engineering Department.

Russell, S., Norvig, P., Candy, J., Malik, J., and Edwards, D. (2010). *Artificial Intelligence: A Modern Approach*. Prentice Hall.

Saad, E. (2010). Reinforcement learning in partially observable markov decision processes using hybrid probabilistic logic programs. *Arxiv preprint arXiv:1011.5951*.

Schaul, T., Bayer, J., Wierstra, D., Sun, Y., Felder, M., Sehnke, F., Rückstieß, T., and Schmidhuber, J. (2010). PyBrain. *Journal of Machine Learning Research*.

Schultz, W. (1998). Predictive reward signal of dopamine neurons. *Journal of neurophysiology*, 80(1):1–27.

Singh, S. P., Jaakkola, T., and Jordan, M. I. (1995). Reinforcement learning with soft state aggregation. In *Advances in Neural Information Processing Systems 7*, pages 361–368. MIT Press.

Sintzoff, M. (1967). Existence off Van Wijngaarden syntax for every recursively enumerable set. *Annales Sot;. Sci. Bruxelles*, 11:115–118.

Skinner, B. (1938). *The Behavior of Organisms: An Experimental Analysis*. Appleton-Century.

Sloane, N. J. A. (2011). The On-Line Encyclopedia of Integer Sequences. Available online at: http://www.research.att.com/~njas/sequences/A041501.

Stachniss, C., Frese, U., and Grisetti, G. (2011). OpenSLAM. Available at: http://www.openslam.org.

Sun, R. (2003). "a detailed specification of CLARION 5.0". Technical report, Cognitive Science Department, Rensselaer Polytechnic Institute.

Sun, R. (2007). The motivational and metacognitive control in CLARION. *Modeling integrated cognitive systems*, pages 63–75.

Sutton, R. S. and Barto, A. G. (1998). *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA.

Turing, A. M. (1937). On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265.

Turing, A. M. (1938). On Computable Numbers, with an Application to the Entscheidungsproblem. A Correction. *Proceedings of the London Mathematical Society*, s2-43(1):544–546.

Valentín, K. (2010). Automatic detection of security issues using methods from AI. Master's thesis, Faculty of Mathematics, Physics and Informatics, Comenius University, Bratislava.

Veness, J., Ng, K., Hutter, M., and Silver, D. (2010). Reinforcement learning via AIXI approximation. In *Proceedings of the Conference for the Association for the Advancement of Artificial Intelligence (AAAI)*, pages 605–611.

Veness, J., Ng, K., Hutter, M., Uther, W., and Silver, D. (2009). A Monte Carlo AIXI Approximation. *Arxiv preprint arXiv:0909.0801*.

Vernon, D., Metta, G., and Sandini, G. (2007). A survey of artificial cognitive systems: Implications for the autonomous development of mental capabilities in computational agents. *IEEE Transactions on Evolutionary Computation*, 11(2):151–180.

Watkins, C. J. C. H. (1989). *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, UK.

Weng, J. (2004). Developmental robotics: Theory and experiments. *International Journal of Humanoid Robotics*, 1(2):199–236.

Weng, J. (2012). Symbolic models and emergent models: A review. *IEEE Transactions on Autonomous Mental Development*, 4(1):29–53.

Whitehead, S. and Ballard, D. (1991). Learning to perceive and act by trial and error. *Machine Learning*, 7(1):45–83.

Wiedermann, J. (1998a). Kogitoid: Matematický model mentální činnosti. In *Informatické kolokvium Jaro 1998*. Fakulta informatiky Masarykovy univerzity.

Wiedermann, J. (1998b). Towards algorithmic explanation of mind evolution and functioning. *Mathematical Foundations of Computer Science 1998*, pages 152–166.