Radboud University Nijmegen

Faculty of Science

History-based Rewards for POMDPs

-

Thesis MSc Computing Science

Author:
Serena Rietbergen

Supervisor:
dr. Nils Jansen

Second reader:
dr. Sebastian Junges

27 September 2021

Contents

Abstract

We study the problem on how to obtain a policy which optimizes the expected reward for a POMDP which has a history-based reward function R. We introduce reward controllers (RCs), which are deterministic automata that behave like the history-based reward function where the input is the domain of R and the output is the co-domain of R. We show how to obtain RCs given a list of sequences, or a list of regular expressions, together with their respective rewards. We combine this RC together with the original POMDP to create an induced POMDP, which encodes the behavior of the RC into the state space of the POMDP. This induced POMDP will have a reward function that only depends on the current state. Since we assume that rewards are only obtained when the process has stopped, we show how to extend a POMDP with an `end` action that allows us to terminate the process. Now when extending the induced POMDP, we obtain a POMDP that has a reward function that is not dependent of the history. This allows us to use known methods to obtain a policy that maximizes the expected reward. To minimize the memory complexity of this problem, we also introduce a limit T which lets us limit our stochastic process to T steps. When the `end` action has not been used before the limit, the POMDP is forced to perform the `end` action and thereby terminating the process.

Chapter 1

Introduction

In reinforcement learning, we are often shown world models in which the agent gets a reward for certain behavior. Commonly, the goal of the agent is to maximize the expected reward, which is done by performing some strategy that will maximize this reward. These world models are usually described as Markov decision processes (MDPs), where the agent is allowed to take certain actions. The agent will then obtain a certain reward for taking actions in a given state. The strategy the agent uses provides the agent with what action they should take for a given state or given the previous states and actions. There is numerous research for obtaining these strategies for reward functions that directly give a reward for taking an action [1, 2, 3].

However, not every problem can be modeled through the means of a reward function that directly returns a reward for every step. Many real-world problems are usually modeled through a reward that is dependent of the history. Let us look at Figure 1.1, where we have a server robot (R), which main purpose is to serve the guest (G) in a grid-like room. At a certain moment, the guest will request an item (I). At this point the robot will start their journey to obtain the wanted item and then move towards the guest. We only want to give a reward to the robot if it returns the right item, but only when the item was actually requested. If the robot would bring the wrong item, or any item when the guest had not requested it, it should not obtain a reward. This behavior requires the reward function to remember if an item was requested, which item was requested and if it was delivered.

When it comes to fully observable MDPs, multiple methods have been researched [4, 5, 6] which allow us to obtain a strategy that optimizes the expected reward, given a history-based reward function. Since for real-world issues, we mostly do not have access to all information, let us also consider models which are not fully observable, namely partially observable MDPs (POMDPs). Figure 1.1 can also be used in the context of a POMDP. Instead of the robot having full observability of what item they are observing, they only observe the fact that they have encountered an item. The problem could also be simplified by making all items equal and giving a reward for when the robot brings an item to the guest only when they have ordered it. This still presents us with a history-based reward function, namely one in which we need to remember if an item was requested and if the robot has brought the item.

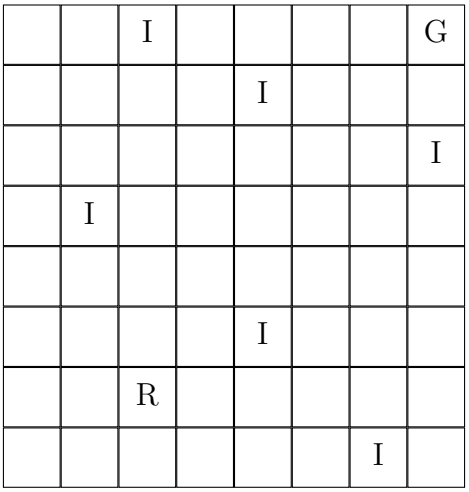


Figure 1.1: Overview for Roomba problem

For POMDPs, the task of obtaining a policy that optimizes the maximum reward (or minimum cost) for non-history-based reward functions, is in general undecidable [7]. Adding a history-based reward function would increase the complexity of the problem, since the reward for taking a certain action in a given state would not be given immediately.

We meet the challenge on how to obtain a policy for history-based rewards for POMDPs by first creating a reward controller (RC). This automaton behaves like the history-based reward function, where the input is the domain of the history-based function and the output is the co-domain. The idea is to combine the POMDP with the history-based reward function into an induced POMDP, which is done by combining the original POMDP with the RC that represents the history-based reward function. This induced POMPD allows us to encode the reward in the states themselves. Since we assume that we are only allowed to obtain the reward after we are done with the process, we extend a POMDP with an extra action to force the process to terminate, allowing us to obtain the reward. By first inducing the POMDP and then extending it, we have not lost the integrity of the history-based reward function. The final product, the extended induced POMDP, will then contain a non-history-based reward. To limit the memory complexity of this problem, we introduce a limit T. This forces the POMDP to terminate after T transitions, if the process has not already been terminated before then.

Problem Formulation

Given a POMDP with a history-based reward function, obtain a policy that maximizes the expected reward.

Contributions

In this thesis we tackle how to obtain a policy which optimizes the expected reward for a POMDP given an history-based reward function. For this we first define reward controllers, which mimic the bahvior of the history-based reward function. We present a procedure on how to obtain a RC given a number of sequences over the domain and their associated rewards. We then present a method on how to obtain a RC given a number of regular expressions over the domain together with their respective rewards.

Next we define an extended POMDP, which helps us to obtain the reward by adding an `end` action, together with a final state to the POMDP. Since we assume we only obtain the reward after we end the process, we can now use the `end` action to obtain the reward. We next define an induced POMDP, which allows us to combine the original POMDP $\mathcal{M}$ together with the RC created from the history-based reward function of $\mathcal{M}$. The reward function for this induced POMDP is now only dependent on the current state and action, which allows us to obtain a policy that optimizes the expected reward through known methods. This in turn allows us to obtain a policy for the original POMDP $\mathcal{M}$. See Figure 1.2 for an overview of how these concepts are related. Furthermore, we introduce a limit T for extended POMDPs which allows us to terminate the process after T transitions if the process has not terminated by then.

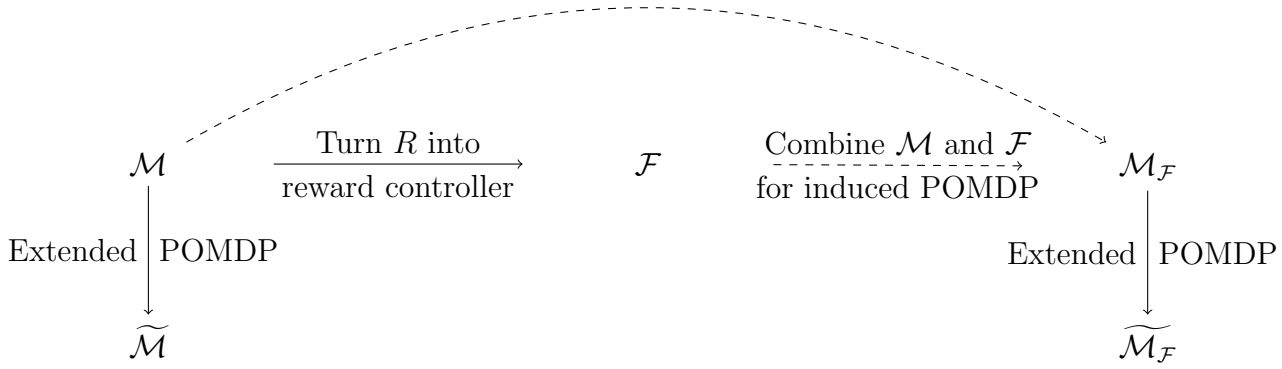


Figure 1.2: Overview

We have implemented procedures on how to obtain a RC from a list of sequences, as well as for a list of regular expressions together with their respective rewards. We have implemented combining the RC together with the original POMDP $\mathcal{M}$ and a limit T. We have implemented a function that allows us to obtain this POMDP in `prism` format. These implementations are used in the case study, to show how the state space blows up depending on the limit T.

Structure

- In Chapter 2 we present the preliminaries.
- Chapter 3 portrays relevant definitions and shows the background information used.
- In Chapter 4 we present the Reward Controllers, including how to obtain them from
  - a list of sequences with their respective rewards, and;
  - a list of regular expressions with their respective rewards.
- In Chapter 5 we show
  - the extended POMDP, an extension of a POMDP;
  - the induced POMDP, which is obtained from the original POMDP together with the RC that represents the history-based reward function, and;
  - the limit T, which forces the POMDP to terminate after T transitions, if the process has not already ended.
- In Chapter 6 we present a casy study involving a Roomba.
- Chapter 7 shows our conclusion and pointers for future work.

Assumptions

- For history-based reward functions, we obtain the reward after the stochastic process has been terminated.
- We only obtain the reward of the entire associated observation sequence.
- When defining a RC over sequences with rewards, we assume all sequences to be unique.

Chapter 2

Preliminaries

Set Theory

For any countable set S, we denote it's cardinality with $|S|$. Let S^* and S^ω be the set of finite and infinite sequences over S, respectively. For any sequence $w \in S^*$ we can denote the length by $|w|$.

Let an alphabet Σ be a a finite set consisting of letters. A word is defined as a sequence of letters $w = w_1 w_2 \dots w_n \in \Sigma^*$. A language L is a set of words given an alphabet Σ, for all languages we have that $L \subseteq \Sigma^*$ holds. Let λ denote the empty word, i.e. $|\lambda| = 0$. For obtaining the number of occurences of a letter $a \in \Sigma$ in a word $w \in \Sigma^*$, we write $|w|_a$.

A regular language is a language that can be defined by a regular expression. The language satisfied by a regular expression e is denoted as $L(e)$.

Probability Theory

For any countable set S we define a *discrete probability distribution* as $\psi : S \to [0, 1]$ where $\sum_{s \in S} \psi(s) = 1$. The set of all possible probability distributions over S is denoted as $\Pi(S)$. We denote the support of a *probability distribution* as $supp(\psi) = \{s \in S \mid \psi(s) > 0\}$.

Let X be a random variable with N outcomes, denoted as $x_1, x_2, \dots, x_n$. Then we can define the expectation of X as follows

$$E[X] = \sum_{i=1}^{N} x_i P(X = x_i)$$

Chapter 3

Background

In this chapter we show the definitions and usages of all relevant concepts. In Section 3.1 we will introduce a number of finite automata, specifically deterministic finite automata and Moore machines. In Section 3.2 we will introduce how Markov processes operate, focussing on Markov chains, Markov decision processes, partially observable Markov decision processes and belief Markov decision processes.

3.1 Finite Automata

3.1.1 Deterministic Finite Automata

Simple deterministic processes can be easily modeled with the help of a finite-state machine. When we are specifically interested in whether a word should be accepted or not given the structure of the language, we can use deterministic finite automata.

Definition 3.1 (DFA)**.** A deterministic finite automaton is a tuple $D = (Q, q_0, \Sigma, \delta, F)$ where

- Q, the finite set of states;
- q_0, the initial state;
- Σ, the input alphabet;
- $\delta : Q \times \Sigma \to Q$, the deterministic transition function;
- $F \subseteq Q$, the set of final states.

The size of any DFA D, will be denoted by $|D|$. The size itself is determined by the number of states, i.e. $|D| = |Q|$.

We had already stated that we are interested in knowing whether any given word should be accepted or not. Specifically, we want to know how a DFA handles certain words and in which state a DFA finishes after reading said word. Since DFAs are deterministic, this can be easily described.

Definition 3.2 (Extended transition function for DFA)**.** We define $\delta^* : Q \times \Sigma^* \to Q$, where $\delta^*(q, w)$ denotes the state we end up after reading word w starting from state q, as follows

$$\delta^*(q, w) = \begin{cases} q & \text{if } w = \lambda \\ \delta^*(\delta(q, a_1), a_2 \dots a_n) & \text{if } w = a_1 a_2 \dots a_n \end{cases}$$

Definition 3.3 (Language accepted by DFA)**.** We say the language accepted by a DFA $D = (Q, q_0, \Sigma, \delta, F)$ consists of all the words that start in the begin state and finish in any final state, i.e. $L(D) = \{w \in \Sigma^* \mid \delta^*(q_0, w) \in F\}$.

Example

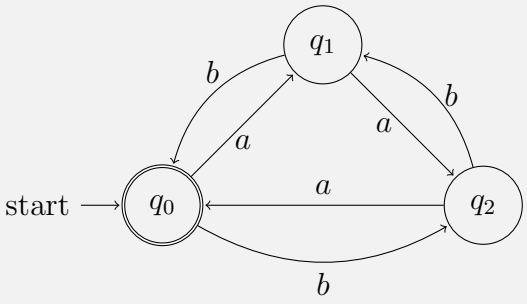


Figure 3.1: A DFA over $\Sigma = \{a, b\}$ which accepts a word w if $|w|_a \equiv |w|_b ($ mod 3).

3.1.2 Moore machine

We now introduce Moore machines, which is a finite state machine similar to the previously mentioned DFA. As has been shown, DFAs are used to show the acceptability of words. This is done by allowing some states to be final, i.e. encoding the acceptability in the states. However, instead of accepting words, Moore machine simply process words and present us with an output during or after reading a sequence. Thus instead of encoding acceptability in the states, we encode an output.

Based on the definition as presented in [8].

Definition 3.4 (Moore machine)**.** A Moore machine is a tuple $(Q, q_0, \Sigma, O, \delta, \sigma)$ where

- Q, the finite set of states;
- $q_0 \in Q$, the initial state;
- Σ, the finite set of input characters - the input alphabet;
- O, the finite set of output characters - the output alphabet;
- $\delta : Q \times \Sigma \to Q$, the input transition function, and;
- $\sigma : Q \to O$, the output transition function.

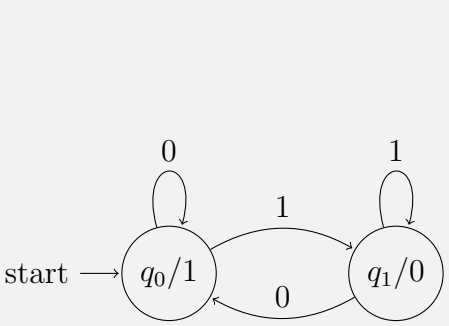
The size of any Moore machine is just as with a DFA determined by the number of states. So for a Moore machine B, the size is $|B| = |Q|$.

As previously mentioned, we can obtain an output while reading a sequence or after reading a sequence. First let us look at obtaining an output while reading.

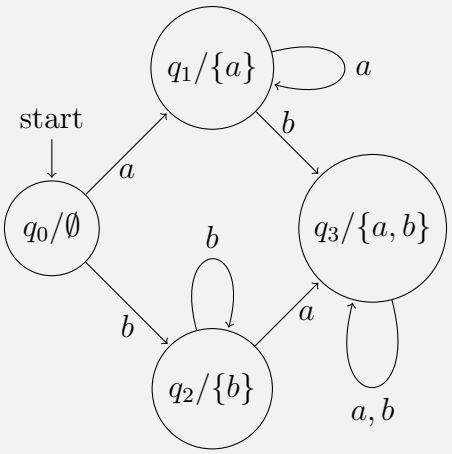
This can be interpreted as transforming some sequence into another sequence. As seen in the definition the output is encoded in the state, so by passing through a state, we obtain a singular output. After the entire input sequence is passed through the machine, we will have obtained a new sequence from the outputs encoded in the states. The second usage of Moore machines is to only obtain the *last* output after we are done with reading the entire sequence, resulting in a singular output.

Example

In Figure 3.2a for $\Sigma = O = \{0, 1\}$ we have a machine that inverts a given sequence. The inverted sequence will however also be preceded by a 1 per construction. For example, when we pass through the sequence 1110, we obtain 10001.
In Figure 3.2b we have $\Sigma = \{a, b\}$ and $O = \{\emptyset, \{a\}, \{b\}, \{a, b\}\}$. The machine outputs the set of used letters in the sequence after being done with reading the sequence. So after reading *aaa* we obtain $\{a\}$.



(a) Inverting sequence



(b) Returns set of used letters

Figure 3.2: Example usages of Moore machines

3.2 Markov Processes

A process can not always be defined using a deterministic machine. Instead, it is possible for a process transition from one state to another by a given probability. In this section we will take a look at some discrete-time stochastic processes, but only those who adhere to the Markov property.

Definition 3.5 (Markov property)**.** For any $s_0, s_1, \ldots, s_{n-1}, s_n \in S$:

$$P(X_n = s_n \mid X_0 = s_0, X_1 = s_1, \ldots, X_{n-1} = s_{n-1}) = P(X_n = s_n | X_{n-1} = s_{n-1})$$

This property states that the probability distribution of X_n is only dependent on its immediate past, namely only on X_{n-1}. So for any stochastic process, given the current state, we know that the future state is not dependent on the past states.

Note that in the entirety of this thesis, we will only be discussing discrete-time Markov processes.

3.2.1 Markov chain

A simple stochastic process that only consists of a set of events which are connected by some probabilities and upholds the Markov property is called a Markov chain.

Definition 3.6 (MC)**.** A Markov chain consists of a set of states S, and initial state $s_I \in S$ and a probabilistic transition function $T : S \to \Pi(S)$.

To obtain the probability to go from state s to state s', we can use $T(s, s') = T(s)(s')$. Note that the probabilistic transition function of any Markov chain can also be described just by the matrix form. The size of a Markov chain is given by the number of states it has, i.e. for a Markov chain H, we state that $|H| = |S|$.

Example

The probabilities of what activity a student partakes in, given the current activity, can be represented as a Markov chain. The activities that are modelled are `sleep`, `study` and being `social`. The first activity of the student is sleeping. How the activities are connected can be seen in Figure 3.3. For example, if the student is studying they have a 0.4 probability of choosing to study as next activity.

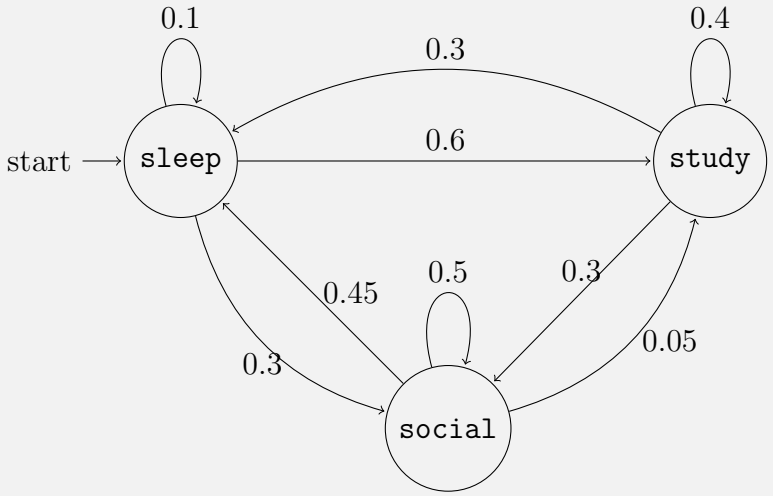


Figure 3.3: Markov chain describing student activities

3.2.2 Markov decision processes

We can see Markov chains as stochastic processes without outside influence, so let us take a look at what happens when we do allow outside influence. This is done by extending the Markov chain with a set of actions, allowing for this influence through the means of performing an action.

Definition 3.7 (MDP)**.** A Markov decision process is a tuple $M = (S, s_I, A, T)$ where

- S, the finite set of states;
- $s_I \in S$, the initial state;

- A, the finite set of actions;
- $T : S \times A \times S \to [0, 1]$, the probabilistic transition function where we require the following for any state $s \in S$ and any action $a \in A$

$$\sum_{s' \in S} T(s, a, s') \in \{0, 1\}$$

The size of any MDP M is determined by the number of states, i.e. $|M| = |S|$.

The *available actions* for a state s are given by $Act(s) = \{ a \in A \mid \exists s' \in S : T(s, a, s') > 0\}$ and we can obtain the *possible successors* of state s in a similar matter through $Succ(s) = \{s \in S \mid \exists a \in A : T(s, a, s') > 0\}$.

A finite *trajectory* or *run* of a MDP is realization of the stochastic process performed by the MDP denoted by the finite sequence $s_1 a_1 s_2 a_2 \dots s_{n-1} a_{n-1} s_n \in (S \times A)^* \times S$. To obtain the last state of a trajectory we can use the function $last : (S \times A)^* \times S$ which works as follows.

$$last(s_1 a_1 s_2 a_2 \dots s_{n-1} a_{n-1} s_n) = s_n$$

Example

Let us model the efficiency of working, given certain physical states of a human body. The states a human can find themselves in are `tired`, `rested` or `energetic`. When someone works when they are tired, they will ultimately remain tired, hence the probability of 1. However, when they would work when `rested` or `energetic` they have a high change of remaining in the same state. To stop being `tired`, the human can take the action `sleep` to either be `rested` or `energetic`.

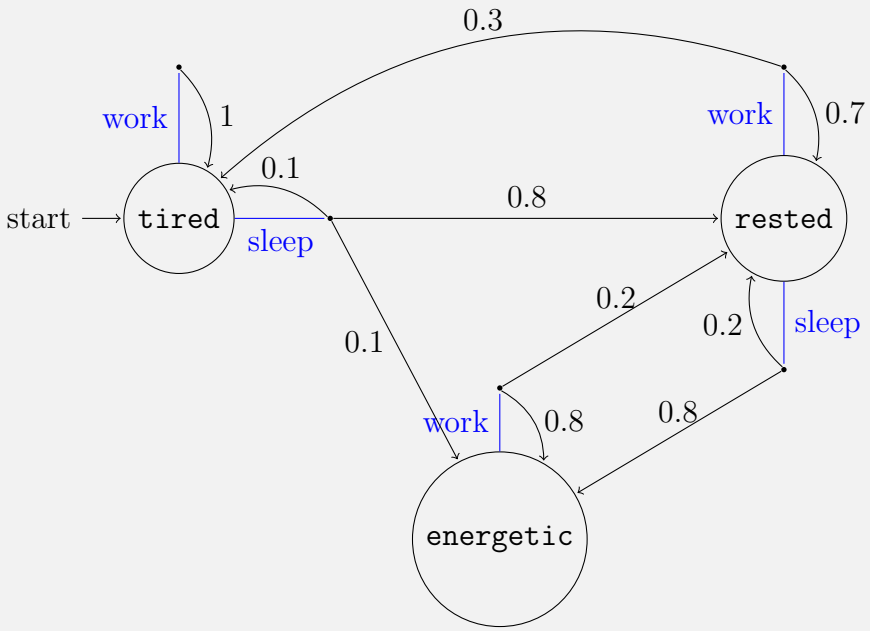


Figure 3.4: MDP modeling work/sleep efficiency

Rewards

We can extend MDPs further with a *reward function* R which assign a reward for taking some action in a state.

Definition 3.8 (Simple reward function)**.** A simple reward function is a reward function that determines a reward based on the current state, action, and obtained state. The reward is independent of its history.

The most conventional notation for this simple reward function is $R : S \times A \to \mathbb{R}$, where we only consider the current state and the taken action. Another possible definition is $R : S \times A \times S \to \mathbb{R}$, in which we through $R(s, a, s')$ consider the specific transition from s to s' by using action a, or $R : S \to \mathbb{R}$ where for $R(s)$ we only consider the visited state s. In Figure 3.4, we could include a simple reward function. For example, we could promote the behavior of only working when energetic by returning a very high reward for performing the action `work` in the state `energetic`. The reward for performing the action `work` in either state `tired` or `rested` should then return a much lower reward.

When modeling complex systems drawn from real world problems, we often encounter that obtaining a certain reward is not only dependent on the current event but also on the states (and actions) that were encountered previously. These *history-based reward functions* are just as versatile as simple reward functions. A few examples are

- $R : S^* \to \mathbb{R}$ - which only looks at the finite states visited, or;
- $R : (SA)^* \to \mathbb{R}$ - which looks at the finite (sub)trajectory without the last obtained state, or;
- $R : (SA)^*S \to \mathbb{R}$ - which looks at the finite (sub)trajectory.

Policy

As stated above, we can extend MDPs with reward functions. Now when modeling a system, we usually want to obtain the expected maximum reward (or minimize the costs involved). However, just obtaining this reward is not enough without knowing how to obtain this. We wish to know what *strategy* we need to apply to obtain this optimal value. For this we use strategies, also known as policies.

Definition 3.9 (Policy)**.** A policy for a MDP M is a function $\pi : (SA)^*S \to \Pi(A)$, which maps a trajectory to a probability distribution over all actions.

We call a policy *memoryless* if the function only considers the last state of the trajectory in deciding the actions. We can apply these types of policies to a MDP to remove the non-determinism, resulting in an induced Markov chain.

Definition 3.10 (Induced MC)**.** Given a MDP $M = (S_M, s_I, A, T_M)$ and a memoryless policy $\pi : S \to \Pi(A)$, we obtain the induced Markov chain $M^\pi = (S, T)$ where we define the probabilistic transition function through

$$T(s, s') = \sum_{a \in A} \pi(s, a) T(s, a, s')$$

Solving for optimal reward

We introduce a discounting factor $\gamma \in [0, 1]$. This factor determines how interesting the immediate reward is. The closer γ is to zero, the more we are interested in the immedate rewards obtained. However, when we are more interested in future rewards, the γ will be closer to one.

Let ρ_t be the expected immediate reward obtained at a moment t in the timeline used. The expected cumulative reward for simple reward functions with an undefined horizon starting in state s, given a policy π, is defined as

$$J^{\pi}(s) = E\big[\sum_{t=0}^{\infty} \gamma\rho_t \mid s, \pi\big]$$

With this we can obtain the optimal policy through the following equation

$$\pi^* = \arg\max_{\pi \in \Pi(A)^S} J^{\pi}(s_I) \tag{3.1}$$

We can solve the problem of how to obtain a policy for the optimal expected reward recursively, due to Bellman's principle of optimality [3]. Let V_i return the accumulated maximum reward up till point i in the given timeline.

$$V_0(s) = 0 \tag{3.2}$$

$$V_n(s) = \max_{a \in A} \big[R(s, a) + \gamma \sum_{s' \in S} T(s, a, s')V_{n-1}(s')\big] \tag{3.3}$$

If we let H be the possible infinite horizon of the problem, the optimal value can be concluded from $J^{\pi^*}(s) = V_{n=H}(s)$. We can also apply a policy to the value function, to obtain the expected reward using policy π as follows

$$V_{\pi,0}(s) = 0$$
$$V_{\pi,n}(s) = \sum_{a \in A} \pi(s, a)\big(R(s, a) + \gamma \sum_{s' \in S} T(s, a, s')V_{\pi,n-1}(s')\big)$$

3.2.3 Partial observability

Having full observability allows us to calculate the optimal expected reward. Finding a policy to maximize the expected reward can be done through policy iteration [1], linear programming [2], or through value iteration as seen in (3.2) and (3.3). However, when modeling a lot of real world issues, we unfortunately do not have all the information readily available to us.

For example, take a machine that breaks down over a period and needs repairs [9]. We know that certain parts in this machine deteriorate at different rates, but we do not know the exact state of each part in this machine. However, we can *observe* how deteriorated the entire system is, which only provides us with partial knowledge of the machine. We can model these types of systems through partially observable MDPs.

Definition 3.11 (POMDP)**.** A partially observable Markov decision process (POMDP) is a tuple $\mathcal{M} = (M, \Omega, O)$ where

- $M = (S, s_I, A, T)$, the hidden MDP;
- Ω, the finite set of observations;
- $O : S \to \Omega$, the observation function.

The size of a POMDP $\mathcal{M}$ is determined by the number of states in the hidden MDP, i.e. $|\mathcal{M}| = |S|$. Let $O^{-1} : \Omega \to 2^S$ be the inverse function of the observation function in which we simply obtain all states in S that have observation o, through $O^{-1}(o) = \{s \in | \ O(s) = o\}$. Without loss of generality we assume that states with the same observations have the same set of available actions, thus $O(s_1) = O(s_2) \Rightarrow A(s_1) = A(s_2)$.

Example

Let us assume an agent works at a dangerous border control post. Most of the time they are waiting (action `w`) for work at our `station`, since they observe no vehicles around to inspect. Once every while a vehicle will approach, but they are uncertain if the vehicle is `friendly` or `hostile`. They can approach the vehicle agressively (action `a`) or normally (action `n`). Let $O($`station`$) =$ `no_vehicle` and $O($`friend`$) = O($`hostile`$) =$ `vehicle`.

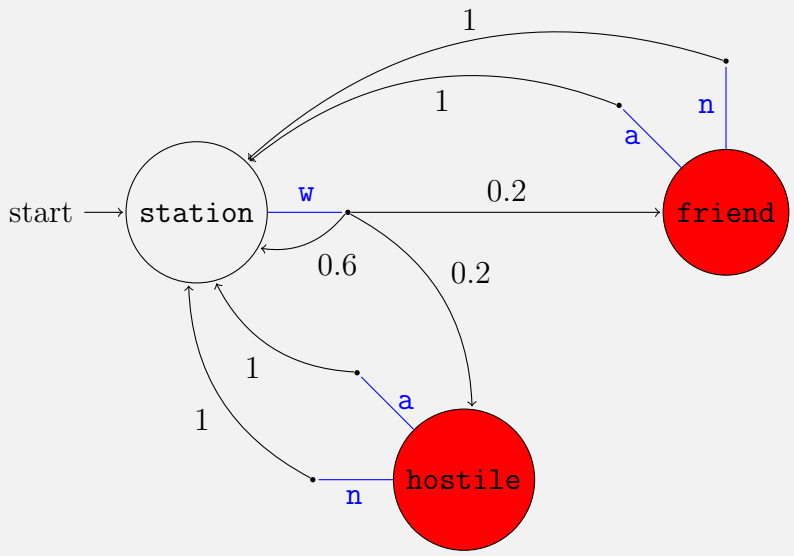


Figure 3.5: POMDP for border control

Since the actual states in a trajectory of the hidden MDP M are not visible, we argue about an *observed trajectory* of the POMDP $\mathcal{M}$. This does not consist of a sequence of states and actions, but instead consists of a sequence of observations and actions, i.e. an element of $(\Omega A)^*\Omega$. The set of all possible finite observed trajectories of POMDP $\mathcal{M}$ will be denoted as $ObsSeq^{\mathcal{M}}$. We can argue about the observed trajectory through the observation function, which will be extended over trajectories:

$$O(s_1 a_1 s_2 a_2 \dots s_{n-1} a_{n-1} s_n) = O(s_1) a_1 O(s_2) a_2 \dots O(s_{n-1}) a_{n-1} O(s_n)$$

Rewards

Usually the reward functions for POMDPs are the rewards defined over the hidden MDP, and will remain an extension of the MDP. In this thesis, we mainly look at history-based reward functions which are defined over all past observations, which is why we use the notation $R : \Omega^* \to \mathbb{R}$.

Policy

Defining a policy over a POMDP is trickier since we only obtain the observation of a state and not which state it actually is, nor how the transitions are defined for that state. For the policy we now base our decision over which action to take given only the observation(s). This provides us with observation-based strategies.

Definition 3.12 (Observation-based strategy)**.** An observation-based strategy of a POMDP $\mathcal{M}$ is a function $\pi : ObsSeq^{\mathcal{M}} \to \Pi(A)$ such that for all $s_1 a_1 \dots s_{n-1} a_{n-1} s_n \in (SA)^* S$:

$$supp(\pi(O(s_1 a_1 \dots s_{n-1} a_{n-1} s_n))) \subseteq Act(last(s_1 a_1 \dots s_{n-1} a_{n-1} s_n))$$

Chapter 4

Reward Controllers

The problem with history-based reward functions is that we have to remember all the previous observations. Because only then are we able to calculate the associated reward, instead of simply calculating the reward per transition without having to store extra information as is done with *simple* reward functions.

In this chapter we are going to take history-based reward functions and transform them into something more tangible. We are going to transform it into an deterministic machine that keeps track of its history and rewards associated. First we will give a formal definition of the machine we are using to represent the reward function. In Section 4.2 we will describe how to obtain such a machine given a list of observation sequences together with their rewards and in Section 4.3 we do the same but for a series of regular expressions.

4.1 Definition

The idea is that we have some history-based reward function $R : \Omega^* \to \mathbb{R}$ which belongs to some POMDP $\mathcal{M}$, where Ω is the set of observations for $\mathcal{M}$. Based on the reward function alone, we are going to build a machine that controls the reward associated to its sequence.

Since a sequence of obersations is nothing more than a word in Ω^*, we are going to build a finite automaton over the alphabet Ω. Then when we have read any word $\texttt{seq} \in \Omega^*$, we want to ensure that the node we end up in contains the reward associated with `seq`. This is a Moore machine where we do not apply σ to every node we encouter, but we only use σ on the last node obtained.

Definition 4.1 (Reward controller)**.** A reward controller $\mathcal{F}$ is a Moore machine $(N, n_I, \Omega, \mathbb{R}, \delta, \sigma)$, where

- N, the finite set of memory nodes;
- $n_I \in N$, the initial memory node;
- Ω, the input alphabet;
- $\mathbb{R}$, the output alphabet;
- $\delta : N \times \Omega \to N$, the memory update;
- $\sigma : N \to \mathbb{R}$, the reward output.

When reading a sequence of observations, or a word in Ω^*, we wish to know in what memory node we end up in because we are interested in the reward encoded into that node. For this we use the following definition, which is similar to the definition given for DFAs.

Definition 4.2 (Extended transition function for RC)**.** We define $\delta^* : N \times \Omega^* \to N$ where $\delta^*(n, \texttt{seq})$ denotes the node we end up after reading $\texttt{seq}$ starting from node n as follows

$$\delta^*(n, \texttt{seq}) = \begin{cases} n & \text{if } \texttt{seq} = \lambda \\ \delta^*(\delta(n, o_1), o_2 \dots o_n) & \text{if } \texttt{seq} = o_1 o_2 \dots o_n \end{cases}$$

Currently our model of a RC is based on the input alphabet Ω, allowing only history-based reward functions which are specified over the observations. Note that this can be easily adjusted to allow for other history-based reward function R. Instead of only allowing for observations, we could also include the actions. Another option would be for history-based reward functions to be defined over the states of the hidden MDP with or without the actions.

Please note that in the rest of the chapter for irrelevant nodes, we use the reward zero. This is a choice to ensure that we can easily calculate for maximizing any history-based **reward**. If we instead wish to model a history-based cost function, which we want to minimize, some slight adjustments need to be made. Since we will be maximizing the reward, we need to 'invert' the costs. Let $r_{\texttt{max}}$ be the maximum reward defined for any given sequence or regular expression with regards to R. Instead of using R to create the reward controller we will use R^{-1}, for every input r over which R is defined over, we have that $R^{-1}(r) = r_{\texttt{max}} - r$. Another option is to set the default reward higher than $r_{\texttt{max}}$ for all irrelevant nodes, and minimize the accumulated costs. These are both adjustments that need to be made for either Definition 4.3 or Definition 4.6

4.2 Generating RCs from sequences with rewards

Let us say we are designing a model for an engineer and they want certain observation sequences to connect to a reward. Thus we are given a number of observation sequences $\texttt{seq}_1, \texttt{seq}_2, \dots, \texttt{seq}_n$ together with their associated real valued rewards $r_1, r_2, \dots, r_n$.

Definition 4.3 (History-based reward function obtained from sequences and associated rewards)**.** Given the observation sequences $\texttt{seq}_1, \texttt{seq}_2, \dots, \texttt{seq}_n$ and their associated rewards $r_1, r_2, \dots, r_n$ we define the history-based reward function $R : \Omega^* \to \mathbb{R}$, which we create as follows

$$R(w) = \begin{cases} r_i & \text{if } w = \texttt{seq}_i \text{ for } i \in \{1, \dots, n\} \\ 0 & \text{otherwise} \end{cases}$$

In R we simply connect the observation sequence $\texttt{seq}_i$ to their respective reward r_i and every other sequence in Ω^* is connected to zero.

We only want to obtain any reward if the associated observation sequence has been entirely observed. Thus we create a reward controller in which we encode the reward in the node we end up in after reading the entire sequence. The idea is as follows: if we read the observation sequence `seq` and we end up in a certain node n, we obtain the reward corresponding to `seq`. I.e. the reward corresponding to `seq` is encoded in n through σ, the reward output function. It is important to note that if we, for example, have $R(\blacksquare\blacksquare) = 2$ and $R(\blacksquare\blacksquare\square) = 3$, and we read $\blacksquare\blacksquare\square$, we will only obtain reward 3.

Given all the sequences over which the history-based reward function is defined, let us create a reward controller through the following procedure. Note that we assume that all the sequences are unique.

Algorithm 1 Procedure for turning a list of sequences into a reward controller

```
 1: procedure CreateRewardController(sequences, R)
Require: sequences
Require: R : Ω* → ℝ
 2:     n_I ← new Node()                                   ▷ initial node
 3:     n_F ← new Node()                                   ▷ dump node
 4:     path(n_I) = λ
 5:     N ← {n_I, n_F}
 6:     for all seq = o_1 o_2 ... o_k in sequences do
 7:         n ← n_I
 8:         for i ← 1, ..., k do
 9:             if δ(n, o_i) is undefined then
10:                 n' ← new Node()                        ▷ create new memory node
11:                 path(n) = o_1 ... o_i
12:                 N ← N ∪ {n'}
13:                 δ(n, o_i) ← n'
14:             n ← δ(n, o_i)                              ▷ update memory node
15:         σ(n) ← R(seq)                                  ▷ set reward
16:     for all n ∈ N do                                   ▷ makes δ and σ deterministic
17:         for all o ∈ Ω do
18:             if δ(n, o) is undefined then               ▷ useless transition
19:                 δ(n, o) ← n_F
20:         if σ(n) is undefined then
21:             σ(n) ← 0
22:     return (N, n_I, Ω, ℝ, δ, σ)
```

We start by creating an initial node in Line 2 and a dump node in Line 3. The idea of the dump node is as follows: since the reward controller is deterministic, if we have a sequence without any relevant reward, we want to ensure that this has a reward of zero. If this sequence is longer than any known sequence, we simply make a transition to this dump state, which will account for most useless sequences. This

state, will only consist of self-loops for all observations and has an encoded reward of zero.

Then for every sequence which we are given, we walk through it. If we come across a transition which is not yet defined, we define it by making a new memory node in Line 10, adding it to N, and setting the transition to this new node. If the transition already exists, we simply update the temporary memory node. After we are done with reading the sequence, we encode the reward into the node itself in Line 15.

Since the reward controller needs to be deterministic as previously mentioned, we set the other undefined values. Every other transition that has not been made yet, will be transferred to the dump node in Line 19. Furthermore, there are still nodes in which the reward is undefined. None of the given sequences ended up in these nodes, so per Definition 4.3 we encode those to zero in Line 21.

Note that the set of nodes N without n_F together with the memory update function is represented as a directed acyclic graph. This indicates that for every node n there is an unique path from the initial node n_I to node n. This unique path is encoded in the function `path`: $N \setminus \{n_F\} \to \Omega^*$. This function is well-defined, since it is defined for n_I in Line 4. Every other time a new node is neccesary, it is created in Line 10, and `path` is then immediately defined for the new node. This `path` function is needed for proving the following lemma.

Lemma 4.4. *For any sequence* $\mathtt{seq} \in \Omega^*$*, let* $r = R(\mathtt{seq})$ *be its associated reward. Then* $\sigma(\delta^*(n_I, \mathtt{seq})) = r$*.*

Proof. Given a sequence `seq`, we set n to be the node we end up in, i.e. $n = \delta^*(n_I, \mathtt{seq})$. If $n = n_F$, we know that the associated reward is zero since $\sigma(n_F) = 0$ per construction. A sequence can only end up in n_F if it was not a part of the pre-defined sequences and following Definition 4.3 the reward is then zero.
However if $n \in N \setminus \{n_F\}$, we can obtain the unique path to node n through $\mathtt{path}(n)$. We know that this is equal to `seq`, so the associated reward is thus

$$\sigma(\delta^*(n_I, \mathtt{seq})) = \sigma(n) = R(\mathtt{path}(n)) = R(\mathtt{seq}) = r$$

□

We observe that $|\mathcal{F}|$, which is the number of memory nodes, is bounded by $|\Omega|^k + 1$ where $k = \max_{seq \in sequences} |seq|$. This will cause rather large RCs. Note that when using state of the art automata learning, the sizing of the reward controller obtained for a number of sequences can be decreased.

Example

Let us assume we are given the following sequences and rewards.

(1) □ □ with a reward of 15

(2) ■ □ ■ with a reward of 20

(3) □ □ ■ □ with a reward of 12

(4) ■ with a reward of 2

Following Procedure 1 we create the associated reward controller. To show how the procedure works, we will show you the intermediate reward controller after processing every sequence.

After sequence (1)

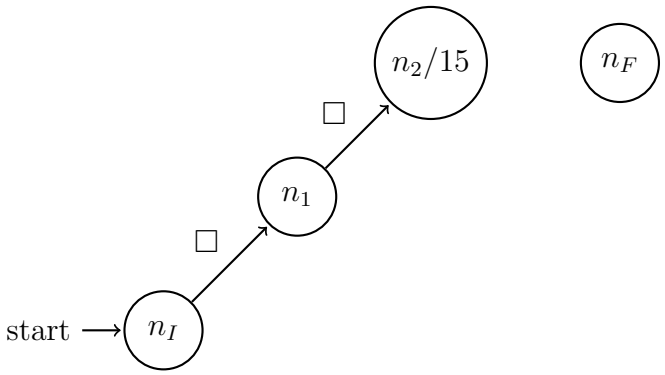


Figure 4.1: Reward controller after sequence (1)

After sequence (2)

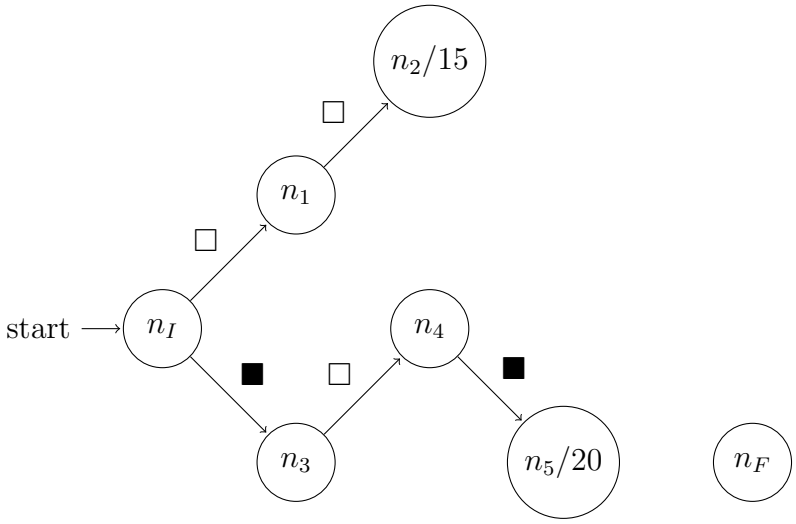


Figure 4.2: Reward controller after sequence (1) and (2)

After sequence (3)

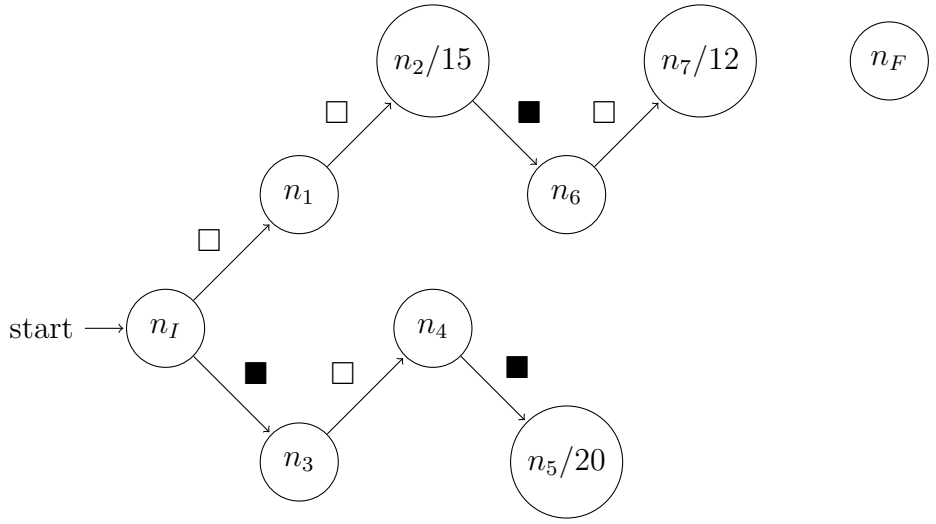


Figure 4.3: Reward controller after sequence (1), (2) and (3)

After sequence (4)

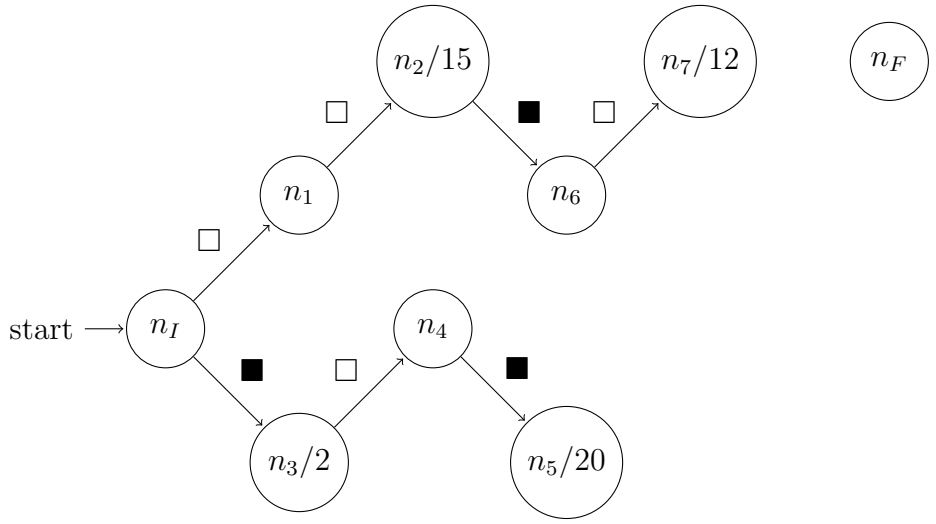


Figure 4.4: Reward controller after sequence (1), (2) and (3)

Finalized Reward Controller

Now we complete the reward controller by completing the rest of the transitions and. Note that `path` was only used for proving Lemma 4.4, so it is not included in any of the figures. In Figure 4.5 the dashed line denotes all the other possible letters for which the transition function δ was not defined. We also update the associated reward for every memory node.

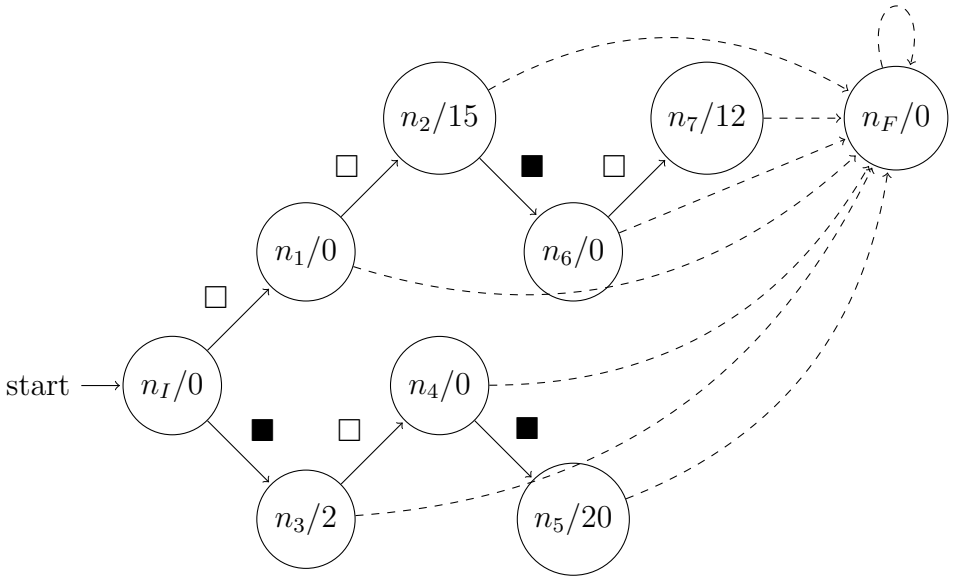


Figure 4.5: Final reward controller

4.3 Obtaining RCs from regular expressions with rewards

It is also possible to define a history-based function in terms of regular expressions to mimic wanted properties with regards to the model. For example, in regards to self-driving vehicles, we want the agent to not crash into any objects while driving towards their destination before finally reaching the destination. In broad terms, this can be written as the following regular expression with the observations `no_crash` and `destination`:

$$(\texttt{no\_crash})^*\texttt{destination}$$

Let us give this regular expression the associated positive reward r. Then every observation sequence in $\{\texttt{no\_crash}, \texttt{destination}\}$ generated by the regular expression above, would be assigned the reward r. Every other observation sequence would be assigned the reward zero. Of course, if there are more observations in this scenario, they have to be taken into account. In this scenario we have to remember if any crashes have occured, before finally observing the destination. Only then do we award the agent with the reward.

Definition 4.5 (History-based reward function obtained from regular expressions and associated rewards)**.** Given n regular expressions $e_1, e_2, \ldots e_n$ together with their associated rewards $r_1, r_2, \ldots, r_n$, we define the history-based reward function $R : \Omega^* \to \mathbb{R}$ as

$$R(\texttt{seq}) = \sum_{\substack{i \in \{1,\ldots,n\} \\ \texttt{seq} \in L(e_i)}} r_i$$

We want to create a reward controller that mimics the behaviour of several regular expressions and their associated rewards. Note that we only want a reward when the sequence of observations is accepted by the language generated by the

regular expression. The first step is to create a DFA that is generated by the regular expression given, which can be done by using known methods [10]

Given regular expression $e_1, e_2, \ldots, e_n$, we create n DFAs. Let $D_i = (Q_i, q_{0,i}, \Omega, \delta_i, F_i)$ be the DFA that accepts the language generated by e_i. Per construction we have that $L(D_i) = L(e_i)$.

Note that to obtain a RC that encapsulates the behavior of all the regular expressions combined, we have to encode the reward in the nodes. This is solved by only encoding the reward of DFA D_i in all states of F_i. For example if $\texttt{seq} \in \Omega^*$ gets accepted by D_i, we have to ensure that the reward is only encoded in the accepting states of D_i. The following definition helps us achieve this goal.

Definition 4.6 (Associated reward function)**.** Let $R_A : Q_1 \cup Q_2 \cup \cdots \cup Q_n \to \mathbb{R}$ be the associated reward function that maps any state q of all the state spaces of $D_1, D_2, \ldots, D_n$ to their respective rewards. If q is a final state of DFA D_i it should get the reward corresponding to the regular expression used for that specific DFA, i.e. r_i. In other words,

$$R_A(q) = \begin{cases} r_i & \text{if } q \in F_i \\ 0 & \text{otherwise} \end{cases}$$

Having obtained all these seperate DFAs, we can now create a DFA that will accept any word that is accepted by any of the seperate DFAs as follows.

Definition 4.7 (Product DFA)**.** The induced product DFA for given DFAs $D_1, D_2, \ldots, D_n$ where $D_i = (Q_i, q_{0,i}, \Sigma, \delta_i, F_i)$ is a tuple $D = (Q, q_0, \Sigma, \delta, F)$ where

- $Q = Q_1 \times Q_2 \times \cdots \times Q_n$
- $q_0 = \langle q_{0,1}, q_{0,2}, \ldots, q_{0,n} \rangle$
- Σ, the same input alphabet
- $\delta(\langle q_1, q_2, \ldots, q_n \rangle, a) = \langle \delta_1(q_1, a), \delta_2(q_2, a), \ldots, \delta_n(q_n, a) \rangle$
- $F = \{\langle q_1, q_2, \ldots, q_n \rangle \mid \exists i \in \{1, 2, \ldots, n\} : q_i \in F_i\}$

To show that the obtained product DFA does indeed accept any word that is accepted by any of the DFAs it was constructed of, we use the following Lemma.

Lemma 4.8. *Given n DFAs where $D_i = (Q_i, q_{0,i}, \Sigma, \delta_i, F_i)$, let D be the product automaton as obtained in Definition 4.7. Then we have that*

$$L(D) = \bigcup_{i=1}^{n} L(D_i)$$

Proof.

$$\begin{aligned} w \in L(D) &\iff \delta_N^*(q_0, w) \in F \\ &\iff \langle \delta_1^*(q_{0,1}, w), \delta_2^*(q_{0,2}, w), \ldots, \delta_n^*(q_{0,n}, w) \rangle \in F \\ &\iff \exists i \in \{1, \ldots, n\} : \delta_i^*(q_{0,i}, w) \in F_i \\ &\iff \delta_1^*(q_{0,1}, w) \in F_1 \text{ or } \delta_2^*(q_{0,2}, w) \in F_2 \text{ or } \ldots \text{ or } \delta_n^*(q_{0,n}, w) \in F_n \\ &\iff w \in L(D_1) \text{ or } w \in L(D_2) \text{ or } \ldots \text{ or } w \in L(D_n) \\ &\iff w \in L(D_1) \cup L(D_2) \cup \cdots \cup L(D_n) \end{aligned}$$

□

The only step left to obtain the reward controller is to connect the correct memory nodes with the right reward, this is done by combining the product DFA of Definition 4.7 together with the associated reward function of Definition 4.6.

Definition 4.9 (Induced reward controller)**.** Given a (product) DFA $N = (Q, q_0, \Omega, \delta, F)$ and the associated reward function R_A, we define the induced reward controller $\mathcal{F} = (N, n_I, \Sigma, O, \delta_{\mathcal{F}}, \sigma)$ as follows

- $N = Q$, the finite set of memory nodes;
- $n_I = q_0$, the initial memory node;
- $\Sigma = \Omega$, the input alphabet;
- $O = \mathbb{R}$, the output alphabet;
- $\delta_{\mathcal{F}} = \delta$, the memory update;
- $\sigma : Q \to \mathbb{R}$, the reward output where

$$\sigma(\langle q_1, q_2, \ldots, q_n \rangle) = \sum_{i=1}^{n} R_A(q_i)$$

Note that σ is defined by taking the sum over the associated rewards. This is because if we have a sequence $\texttt{seq} \in \Omega^*$ that is accepted by several regular expressions given, it should obtain all the seperate rewards associated with those regular expressions. Through the following lemma we ensure that for any sequence $\texttt{seq} \in \Omega^*$ the reward controller obtains the combination of rewards depending on the final state after having read `seq`.

Lemma 4.10. *Given $e_1, e_2, \ldots, e_n$, the sequence of regular expressions together with their associated rewards $r_1, r_2, \ldots, r_n$, let D be the product automaton as defined in Definition 4.7 build from the DFAs D_i for which $L(D_i) = L(e_i)$. Then let $\mathcal{F} = (N, n_I, \Omega, \mathbb{R}, \delta, \sigma)$ be the reward controller as defined in Definition 4.9 given D. We say that for all possible words $\texttt{seq} \in \Omega^*$ the following holds:*

$$\sigma(\delta^*(n_I, \texttt{seq})) = R(\texttt{seq})$$

Proof.

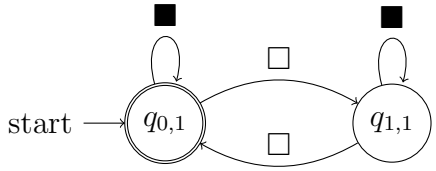
$$\sigma(\delta^*(n_I, \texttt{seq})) = \sigma(\langle q1, q2, \ldots, q_n \rangle) \tag{4.1}$$
$$= \sum_{i}^{n} R_A(q_i) \tag{4.2}$$
$$= \sum_{\substack{i \in \{1,\ldots,n\} \\ q_i \in F_i}} R_A(q_i) \tag{4.3}$$
$$= \sum_{\substack{i \in \{1,\ldots,n\} \\ q_i \in F_i}} r_i \tag{4.4}$$
$$= \sum_{\substack{i \in \{1,\ldots,n\} \\ \delta^*(q_{0,i}, \texttt{seq}) \in F_i}} r_i \tag{4.5}$$
$$= \sum_{\substack{i \in \{1,\ldots,n\} \\ \texttt{seq} \in L(D_i)}} r_i \tag{4.6}$$
$$= \sum_{\substack{i \in \{1,\ldots,n\} \\ \texttt{seq} \in L(e_i)}} r_i \tag{4.7}$$
$$= R(\texttt{seq}) \tag{4.8}$$

For Equation (4.1) we simply use Definition 4.2 and the fact that D is deterministic, so it ends up in an unique state after reading `seq`. For Equation (4.2) we use the definition for σ as seen in Definition 4.9. For Equation (4.3) we use that fact that in Definition 4.6 we observe that $R_A(q_i)$ is equal to zero if $q_i \notin F_i$ and only produces a non-zero value for all $q_i \in F_i$. Thus we only look at the q_i which return a non-zero value. Since we now know we only look at the non-zero reward values, we can use Definition 4.6 again in Equation (4.4). From Definition 3.2 we can rewrite the equation in Equation (4.5). For Equation (4.6) we use Definition 3.3. Since per construction $L(e_i) = L(D_i)$ for all $i \in \{1, \ldots, n\}$, we rewrite the term in Equation (4.7). Finally in Equation (4.8) we simply apply Definition 4.5. □

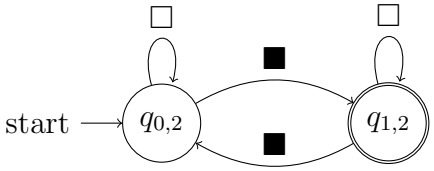
Logically, the size of the reward controller obtained through a series of n automata is bounded by $\prod_{i \in \{1,2,\ldots,n\}} |D_i|$, where D_i is the automaton obtained through the regular expressions e_i. The size of the resulting RC can usually be optimized by using algorithm to determine states that are equal. For example, we can use Algorithm 5.7.2 from Languages and Machines [11]

Example

Let us state that we are given 2 regular expressions. One is that an even number of □ produces a reward of 10 and the other states that an odd number of ■ gives a reward of 15. In other words $R(e_1) = R(\texttt{even number of } \square) = 10$ and $R(e_2) = R(\texttt{odd number of } \blacksquare) = 15$. Let us first obtain the two DFAs that are generated by e_1 and e_2. Those can be seen in Figure 4.6.



(a) DFA for regular expression even number of □



(b) DFA for regular expression for odd number of ■

Figure 4.6

Then we create the product automaton as defined in Definition 4.7. The result can be seen in Figure 4.7.

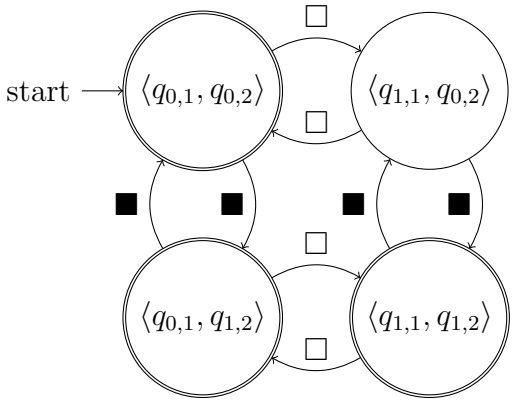


Figure 4.7: Product DFA for both regular expressions

From this we then obtain the reward controller as per Definition 4.9, and can be found in Figure 4.8. Note that

$$R_A(q_{0,1}) = 10$$
$$R_A(q_{1,1}) = R_A(q_{0,2}) = 0$$
$$R_A(q_{1,2}) = 15$$

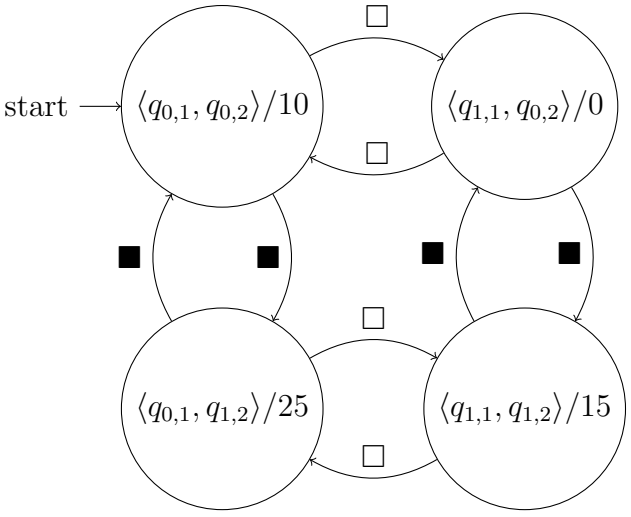


Figure 4.8: Reward Controller for R

Chapter 5

Integrating the history

In this chapter we combine the obtained reward controller $\mathcal{F}$, which is based on the history-based reward function $R : \Omega^* \to \mathbb{R}$, together with the original POMDP M. This allows us to remove the history-based aspect of the reward function, allowing us to calculate the reward step-by-step. This can be seen in Section 5.2. First, since there is no method to ending the process we add another action to forcibly *end* the process in Section 5.1. In Section 5.3 we will show an example. In Section 5.4, we discuss how all the definitions are related. Finally, in Section 5.5 we talk about on how to limit the computational process by only allowing sequences up to a certain length.

5.1 Ended the stochastic process

In our assumptions we have stated that we obtain the reward from a history-based reward function only when we are finished with the process. So given a POMDP with a history-based reward function, we need an active way to end the process to obtain the reward. This can be solved by adding an action to actively end the model and this can be done by extending the model with the action `end` together with a final state. This final state then should only consist of deterministic loops.

Definition 5.1 (Extended POMDP)**.** The extended POMDP for a given POMDP $\mathcal{M} = (M, \Omega, O)$ where $M = (S, s_I, A, T_M)$ is a new POMDP $\widetilde{\mathcal{M}} = (\widetilde{M}, \widetilde{\Omega}, \widetilde{O})$ where

- $\widetilde{M} = (S', s_I, A', T'_M)$, the hidden MDP where:
  - $S' = S \cup \{s_F\}$, the finite set of states;
  - $A' = A \cup \{\texttt{end}\}$, the finite set of actions;
  - $T'_M : S' \times A' \to S' \to [0, 1]$, the probabilistic transition function defined as:

$$T'_M(s, a, s') = \begin{cases} 1 & \text{if } s' = s_F \text{ and } a = \texttt{end} \\ T_M(s, a, s') & \text{if } s \neq s_F \\ 0 & \text{otherwise} \end{cases}$$

- $\widetilde{\Omega} = \Omega \cup \{o_F\}$

- $\widetilde{O} : S' \to \widetilde{\Omega}$ where

$$\widetilde{O}(s) = \begin{cases} o_F & \text{if } s = s_F \\ O(s) & \text{otherwise} \end{cases}$$

We also need to adjust the reward function for the extended POMDP, to accomodate for the new information to actually ensure we can obtain the reward.

Definition 5.2 (Adjusted reward function for extended POMDP)**.** Given the extended POMDP $\widetilde{\mathcal{M}}$ and the original history-based reward function $R : \Omega^* \to \mathbb{R}$, we obtain the new reward funtion $\widetilde{R} : \Omega^* \times A \to \mathbb{R}$ where

$$\widetilde{R}(o_1 o_2 \dots o_n, a) = \begin{cases} R(o_1 o_2 \dots o_n) & \text{if } a = \texttt{end} \\ 0 & \text{otherwise} \end{cases}$$

5.2 Combining the RC with the original POMDP

We will now combine the reward controller $\mathcal{F}$ with the related PODMP to obtain an induced POMDP where we map the memory into the structure. This ensures that we do not have to keep the observation sequence in memory and that allows us to calculate the reward step by step.

Definition 5.3 (Induced POMDP)**.** The induced POMDP for reward controller $\mathcal{F} = (N, n_I, \Omega, \mathcal{R}, \delta, \sigma)$ on a POMDP $\mathcal{M} = (M, \Omega, O)$ where $M = (S, s_I, A, T_M)$ is a tuple $\mathcal{M}_\mathcal{F} = (M_\mathcal{F}, \Omega, O_\mathcal{F})$ where

- $M_\mathcal{F} = (S_\mathcal{F}, s_{I,\mathcal{F}}, A, T_{M_\mathcal{F}})$, the hidden MDP where:
  - $S_\mathcal{F} = S \times N$, the finite set of states;
  - $s_{I,\mathcal{F}} = \langle s_I, \delta(n_I, O(s_I)) \rangle$, the initial state;
  - $T_{M_\mathcal{F}} : S_\mathcal{F} \times A \times S_\mathcal{F} \to [0, 1]$, the probabilistic transition function defined as:

$$T_{M_\mathcal{F}}(\langle s, n \rangle, a, \langle s', n' \rangle) = \begin{cases} T_M(s, a, s') & \text{if } \delta(n, O(s')) = n' \\ 0 & \text{otherwise} \end{cases}$$

- $O_\mathcal{F} : S_\mathcal{F} \to \Omega$, the observation function where

$$O_\mathcal{F}(\langle s, n \rangle) = O(s)$$

The new set of states is a product of the set of states S of the hidden MDP and the set of memory nodes N of the RC. We only allow transitions between states of $S_\mathcal{F}$ if they align with regards to the transition function δ of $\mathcal{F}$. When transitioning between states in $S_\mathcal{F}$, we always look at the observation of the state we are going to reach.

Note that when we start any stochastic process, we always obtain the observation of the initial state. Since the transition function $T_{\mathcal{M}_\mathcal{F}}$ only looks at the observation of state we transition to, we still need to take the observation of the initial state into account. This is the reason why the initial state is $\langle s_I, \delta(n_I, O(s_I)) \rangle$ instead of $\langle s_I, n_I \rangle$.

Definition 5.4 (Adjusted reward function for induced POMDP)**.** The reward function of the induced POMDP $\mathcal{R}_{\mathcal{F}} : S_{\mathcal{F}} \to \mathbb{R}$ is defined as

$$\mathcal{R}_{\mathcal{F}}(\langle s, n \rangle) = \sigma(n)$$

where σ is the reward output function of the reward controller $\mathcal{F}$.

It is important to note that we still only want the actual involved reward after we have decided we are done with the process. This indicates that we are not interested in the reward function of the induced POMDP, but instead in the one of the extended induced POMDP. So we extend this induced POMDP $\mathcal{M}_{\mathcal{F}}$ as presented in Definition 5.1, yielding $\widetilde{\mathcal{M}_{\mathcal{F}}}$. This can easily be derived from Definition 5.2 and Definition 5.4.

Definition 5.5 (Reward function for extended induced POMDP)**.** The reward function $\widetilde{\mathcal{R}_{\mathcal{F}}} : (S_{\mathcal{F}})' \times A \to \mathbb{R}$ for the extended induced POMDP $\widetilde{\mathcal{M}_{\mathcal{F}}}$ is defined as follows:

$$\widetilde{\mathcal{R}_{\mathcal{F}}}(s, a) = \begin{cases} \sigma(n) & \text{if } a = \texttt{end} \text{ and } s = \langle s', n \rangle \\ 0 & \text{otherwise} \end{cases}$$

Note that for the original POMDP $\mathcal{M}$ we could only calculate the reward after we were done with the process. However, for the newly obtained POMDP $\mathcal{M}_{\mathcal{F}}$ we obtain the reward as the process continues, since it is now dependent only on the state and action.

Since we encorporate the reward function in the states, we blow up the state space. The size of the induced POMDP is bounded by $|\mathcal{F}| \cdot |\mathcal{M}|$, which indicates that the extended induced POMDP is bounded by$|\mathcal{F}| \cdot |\mathcal{M}| + 1$. The size of the POMDP $\mathcal{M}$ is set, so it is important that the created RC $\mathcal{F}$ is minimalized as much as possible.

In Section 4.1 it was mentioned that other reward functions are possible. If we were to use a different input alhabet, we have to note that Definition 5.3 will change as well. For our current used history-based reward function over the observations, we observe that a transition between states within the induced POMDP are only available if there is a transition within $\mathcal{F}$ for the given observation. This is because the input language of the RC is Ω. If we were to change the input language, this condition for allowing a transition between states would have to be adjusted accordingly. The initial state would also have to be adjusted.

5.3 Example

Assume we are given a POMDP as seen in Figure 5.1, where $\Omega = \{\square, \blacksquare\}$. For this POMDP we are given a simple history-based reward function R, which states "if I have observed an even number of $\square$, then my reward is 10". We choose the following regular expression for this statement:

$$(\blacksquare|\square\blacksquare^*\square\blacksquare^*)^*$$

The corresponding reward controller can be seen in Figure 5.2.

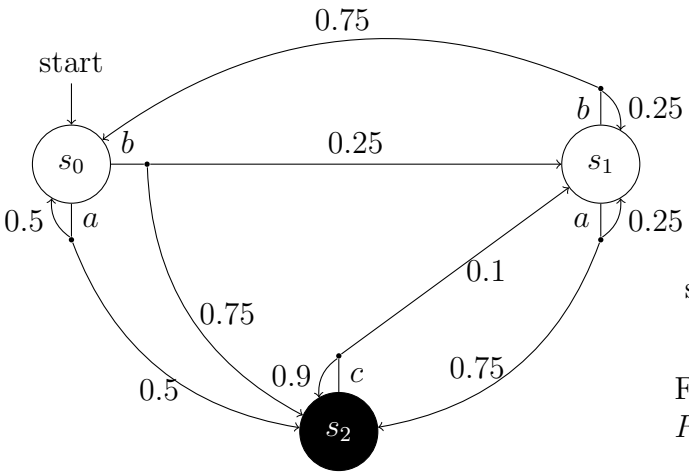


Figure 5.1: POMDP with $\Omega = \{\square, \blacksquare\}$

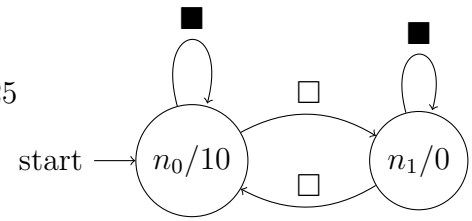


Figure 5.2: Reward Controller for R

The induced POMDP corresponding with Definition 5.3 can be seen in Figure 5.3. If we then extend this induced POMDP and `end` the process in state $\langle s_0, n_0\rangle$, $\langle s_1, n_0\rangle$ or $\langle s_2, n_0\rangle$ we would then obtain a reward of 10, otherwise we obtain a reward of zero. Only in those states we have met the requirement of having an even number of observation $\square$.

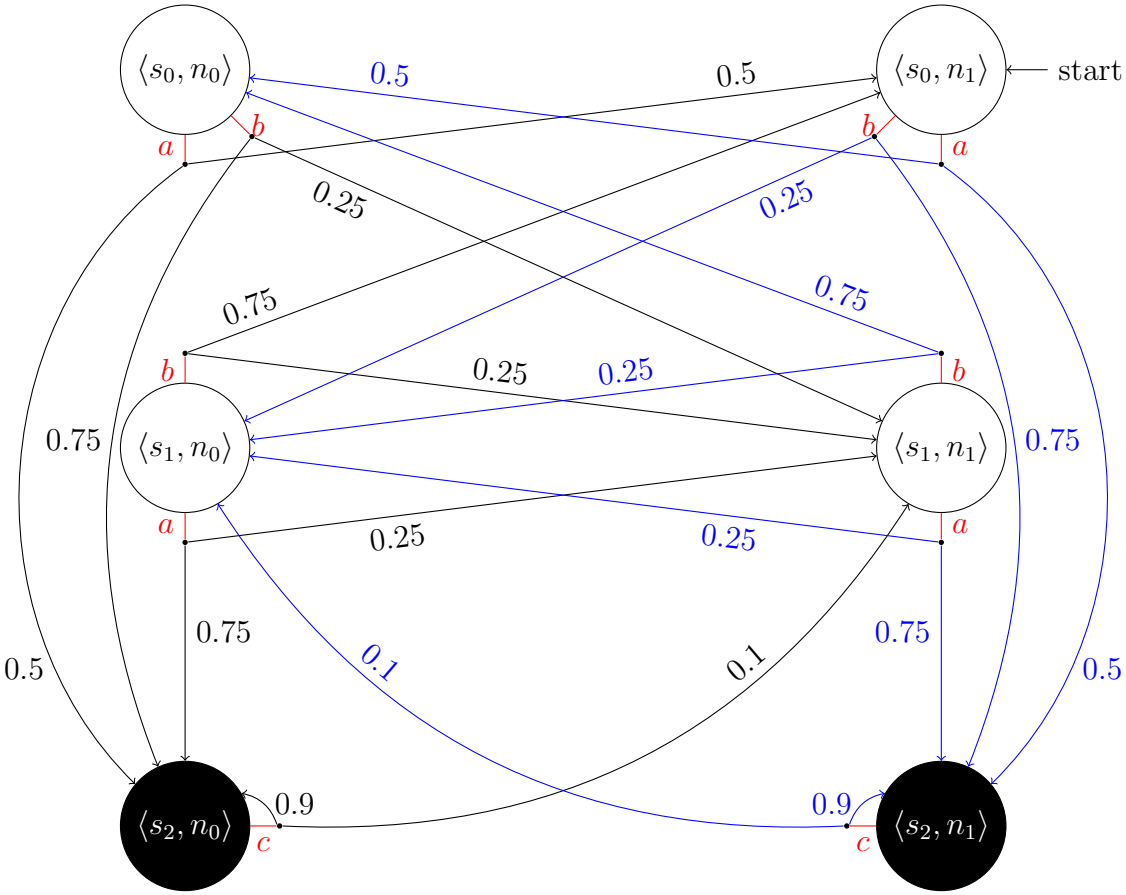


Figure 5.3: POMDP with $\Omega = \{\square, \blacksquare\}$

5.4 Obtaining policy for original POMDP

See Figure 5.4 for an overview of all the definition presented previously. Note that for the creation of $\mathcal{M}_{\mathcal{F}}$ both $\mathcal{M}$ and $\mathcal{F}$ are needed. Let R be the history-based reward function that is defined over $\mathcal{M}$.

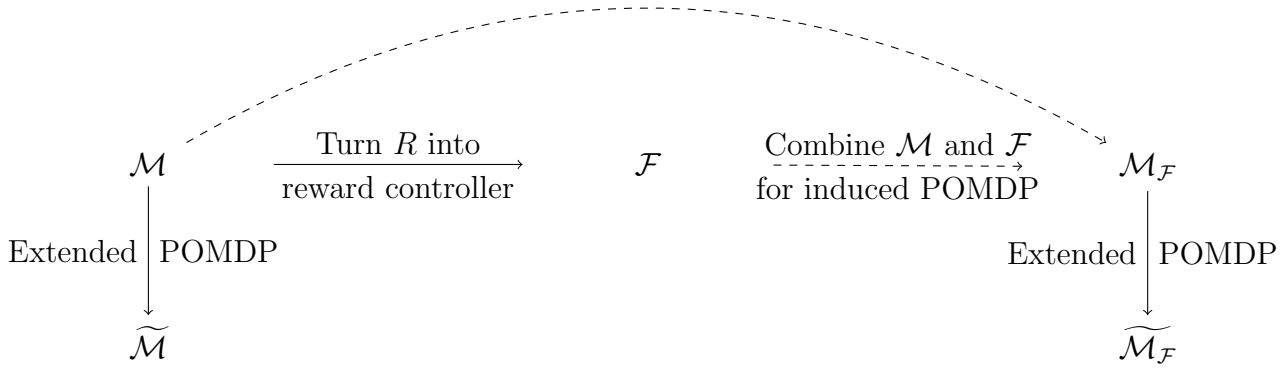


Figure 5.4: Overview

If we have obtained an optimal policy for $\widetilde{\mathcal{M}_{\mathcal{F}}}$ we also obtain an optimal policy for $\widetilde{\mathcal{M}}$, since the observations and actions are the same for both. This policy can then easily be transformed to get the corresponding policy for $\mathcal{M}$.

There are a few differences between $\mathcal{M}$ and $\widetilde{\mathcal{M}}$, namely the observations and the actions which need to be translated to account for a proper policy for $\mathcal{M}$. The observations only differ because $\widetilde{\mathcal{F}}$ includes the extra state s_F. When you enter this state, there are not any actions available since this is a state that only consists of deterministic loops. This indicates that no actions need to be taken for when observing $O(s_F) = o_F$ and since $\mathcal{M}$ does not contain s_F nor o_F for that matter, we do not need the policy to account for this. The other issue is that the set of actions of $\widetilde{\mathcal{M}}$ also contains the actions `end` to actively end the stochastic process. This could be transformed into just not taking any action and in that way stop the process for $\mathcal{M}$.

In between the different forms of POMDPs, either induced or extended we have changed the history-based reward function, to allow us to calculate the reward step-by-step instead of only at the end. Having obtained the original POMDP $\mathcal{M}$ with the history-based reward function R, we first transform $\mathcal{M}$ into $\mathcal{M}_{\mathcal{F}}$. By doing this, we also obtain $\mathcal{R}_{\mathcal{F}}$, which encodes the RC $\mathcal{F}$ into the state space of the original POMDP. $\mathcal{M}_{\mathcal{F}}$ still behaves on the notion that we only obtain the reward when we are done with the process. This is why the rewards are encoded in the states themselves, which is based on the behaviour of the RC $\mathcal{F}$. Then we simple extend this induced POMDP to $\widetilde{\mathcal{M}_{\mathcal{F}}}$, which provides us with $\widetilde{\mathcal{R}_{\mathcal{F}}}$. This transformation only allows us to actively end the process with `end`, in which we then obtain the reward when we perform this action. To formally show that the rewards did not change, we use the following lemma.

Lemma 5.6. *Let $R : \Omega^* \to \mathbb{R}$ be the original history-based reward function of POMDP $\mathcal{M}$. Then let $\widetilde{\mathcal{M}_{\mathcal{F}}}$ be the extended(Definition 5.1) induced(Definition 5.3)*

POMDP associated. Let this new POMDP have the reward function as defined in Definition 5.5. Then for all $\langle s, n\rangle \in S_{\mathcal{F}}$ we have

$$\widetilde{\mathcal{R}_{\mathcal{F}}}(\langle s, n\rangle, \texttt{end}) = R(o_1 o_2 \dots o_n)$$

where $o_1 o_2 \dots o_n$ is the observation sequence observed up until that point.

Proof. Since we have observed $o_1 o_2 \dots o_n$ and finish in $\langle s, n\rangle$, we know that $O(s) = o_n$.

$$\begin{aligned} R(o_1 o_2 \dots o_n) &= \sigma(\delta^*(n_I, o_1 o_2 \dots o_n)) \text{ Lemma 4.4 or Lemma 4.10} \\ &= \sigma(n) \\ &= \widetilde{\mathcal{R}_{\mathcal{F}}}(\langle s, n\rangle, \texttt{end}) \text{ using Definition 5.5} \end{aligned}$$

□

5.5 Limiting the observation sequence

We now introduce a limit T which forces the process to terminate after T steps. In short, if the action `end` has not been used before T steps, the action will be enforced and thus the sequence will be terminated.

When we try to calculate the probability for ending up in s_F ($P[\mathtt{F}\, s = s_F]$), we encounter the problem that this does not conclude to a probability of 1. This is because there is no certainty that `end` will ever be performed, because the sequences can extend infinitely long. To solve this problem, we present the following definition where we adapt the POMDP with said limit T.

Definition 5.7 (Limited POMDP)**.** We can extend the POMDP with the hidden MDP $M = (S \cup \{s_F\}, s_I, A, T_M)$ of the extended POMDP $\widetilde{M} = (M, \Omega, O)$ with some given counter T, creating a limited POMDP, where the new hidden MDP (S', s'_I, A, T'_M) consists of

- $S' = S \times \{1, \dots, T\}$
- $s'_I = \langle s_I, 1\rangle$
- $T'_M : S' \times A \times S' \to [0, 1]$ where

$$T'_M(\langle s_1, t_1\rangle, a, \langle s_2, t_2\rangle) = \begin{cases} T_M(s_1, a, s_2) & \text{if } t_2 = t_1 + 1 \text{ and } t_2 \neq T \\ 1 & \text{if } t_2 = T \text{ and } s_2 = s_F \\ 0 & \text{otherwise} \end{cases}$$

The observation space Ω will remain the same and the observation function will for every $\langle s, t\rangle$ return $O(s)$ as observation.

The reward function $R : S' \times A \times S' \to \mathbb{R}$ will only look at the states of S and will be transformed into

$$R(\langle s_1, t_1\rangle, a, \langle s_2, t_2\rangle) = R(s_1, a, s_2)$$

This way we enforce that the number of observation is limited to T. We still are allowed to end the sequence at any moment, up untill the sequence of observation has length T at which point we will be forced to terminate. Every obtained observation sequence $o_1 o_2 \dots o_n$ will have that $n \leq T$.

The probability of ending up in the final state s_F now has a probability of 1. Since we encoded the reward into the "final" transition, i.e. the transition for when we enter the final state, we can now use the following policy

$$R_{\max}[\mathtt{F}\ \ s = s_F]$$

This policy allows us to now obtain the maximum expected reward.

Given an extended induced POMDP $\widetilde{\mathcal{M}_{\mathcal{F}}}$, the size of the limited version is bounded by $|\mathcal{M}| \cdot |\mathcal{F}| \cdot T$. Increasing T will lead to more accurate maximum expected rewards, but this will also increase the size complexity with $\mathcal{O}(T)$.

Chapter 6

Case Study

Scenario

Based on a grid world found in [12], we present the following scenario.

Let us have a Roomba in a 10×10 grid as seen in Figure 6.1, and it notices that the battery is empty. The Roomba begins their journey towards their charging station (F). It so happens that the Roomba always registers that the battery needs to be charged on one of four possible starting positions (P). We want the Roomba to go back to their charging station as quickly as possible. However, there are a number of small items (O) that are on the floor. These obstacles are small enough that the Roomba can drive over the item, but the item will break. Another thing to note is the floor has been waxed recently, so the Roomba has a small chance of slipping. When slipping, the Roomba will move two tiles instead of one. The Roomba is only allowed to move in a grid-like manner, allowing it to move north, east, south or west.

									F
						O	P	O	
									O
	P								
	P	P							O
	O								

Figure 6.1: Overview for Roomba problem

The Roomba has the capability to observe 4 different things:

1. `start`: the Roomba needs to start the journey to the charging station.
2. `obstacle`: the Roomba has driven over an item.
3. `goal`: the Roomba has reached the charging station.
4. `notbad`: nothing of interest has happened.

The original POMDP has 101 states, which consists of an intial placement state for deciding which initial position (P) the Roomba has, and one state for each position of the grid.

Obtaining information

To obtain information about the induced POMDP we create with the history-based rewards mentioned in this chapter together with the original POMDP itself, we use the following code [13] for sizing of the POMDPs. For obtaining the reward, I used the following code [14]. We show the size of the POMDP obtained in terms of states, but we also mention the number of "final" states. These final states are the ones who are marked with s_F.

Simple reinforcement learning

There are some different things we can let the Roomba learn in this situation. Let us start off simple. The Roomba has to reach the charging station without driving over any obstacle.

With reward controllers

We can use a history-based reward function as follows:

Reaching the goal, without encountering any obstacles : 1

Which in turn can be represented in the following regular expression:

`start notbad* goal` : 1

presenting us with the following reward controller:

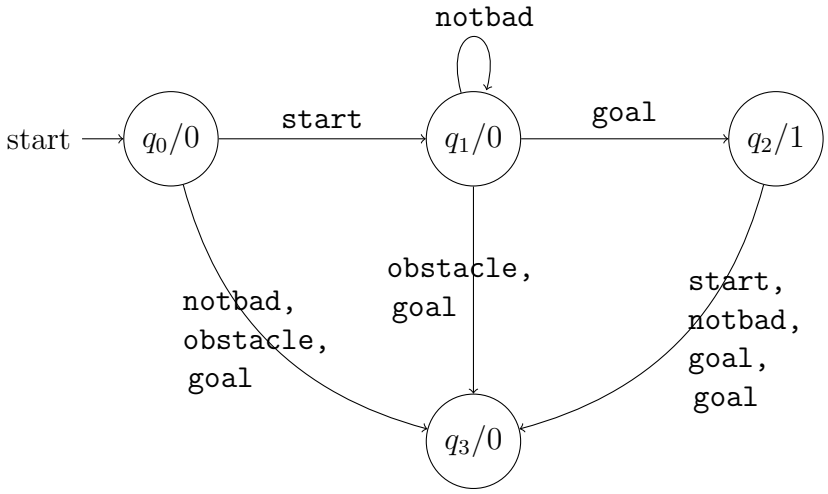


Figure 6.2: Reward Controller for reinforcement learning

Per construction of the original POMDP, there will not be any observation possible after the observation `goal` has been observed. However, since the RC is a deterministic automaton we will show all the possible transitions for now.

To solve this problem given the reward controller mentioned above, we find a policy for maximizing the involved rewards to ensure the Roomba finds a policy to reach the goal without hitting any obstacles.

T	5	10	15	20	25	30	35	40	45	50
States	333	2185	4135	6085	8035	9985	11935	13885	15835	17785
End states	233	1190	2165	3140	4115	5090	6065	7040	8015	8990
Result (e^{-4})	1.709	7.0406	7.3590	7.3828	7.3859	7.3866	7.3867	7.3867	7.3868	7.3868

Table 6.1: Results for using Figure 6.2

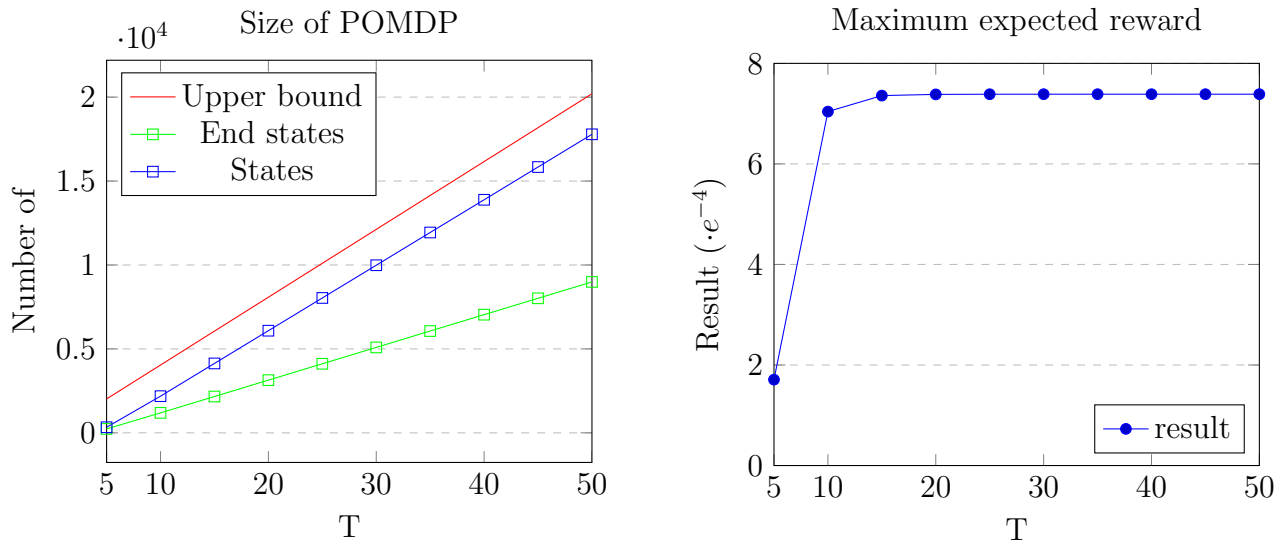


Figure 6.3: Overview of data found in Table 6.1

The size of the induced POMDP together with the expected reward in terms of T can be found in Table 6.1. We can see that the optimal result is obtained for $T = 45$, but that starting from $T = 30$ the results are very close. There is a slight improvement to be found for increasing the T further after $T = 30$, but the improvement is almost negligible. In short, for $T = 30$ we would already have a good estimate. For $T = 30$ we see that the new POMDP roughly has 100 times more states of which more than half is a final state without outgoing transitions. The upper bound of the size of the induced POMDP for $T = 30$ is $101 \cdot 4 \cdot 30 = 12120$, so we see that the number of actual states is a bit below that.

Alternative approach

Another way to tackle the problem of reaching the charging station without hitting any obstacle, is to give a penalty for hitting and thus observing any obstacles. A simple way to handle this problem would be to specify that every time you observe an obstacle, the agent would get a penalty of 50 for example. To find a policy for this, we ask to minimalize the involved costs while still reaching for the charging station. This way there is no blow-up in size, since the stucture of the POMDP will not be changed.

Realistic cost function

However, such a problem is not always linear in real life. A more realistic approach for giving penalties would be to give a small penalty for the first occurence, but higher penalties for when it happens more than once. For example, if the Roomba drives over an object you might still be able to fix the object. But, if the Roomba drives over multiple ones, it will be easier to just throw all of the items out, resulting in a much higher cost. Let us take a look at the following history-based cost function:

- No obstacles were hit: 0.
- One obstacle was hit: 10.
- Two obstacles were hit: 70.
- Three (or more) obstacles were hit: 100.

Using reward controller

To approximate this, let us consider the following approach. Instead of minimizing the involved costs, we want the maximize the profit we get for not having to fix or throw away the items involved. This problem can be transformed into: "Assuming the costs are 100, how much do I end up gaining?". This is represented through the following history-based reward function:

- No obstacles: 100.
- One obstacle: 90.
- Two obstacles: 30.

- Three or more obstacles: 0.

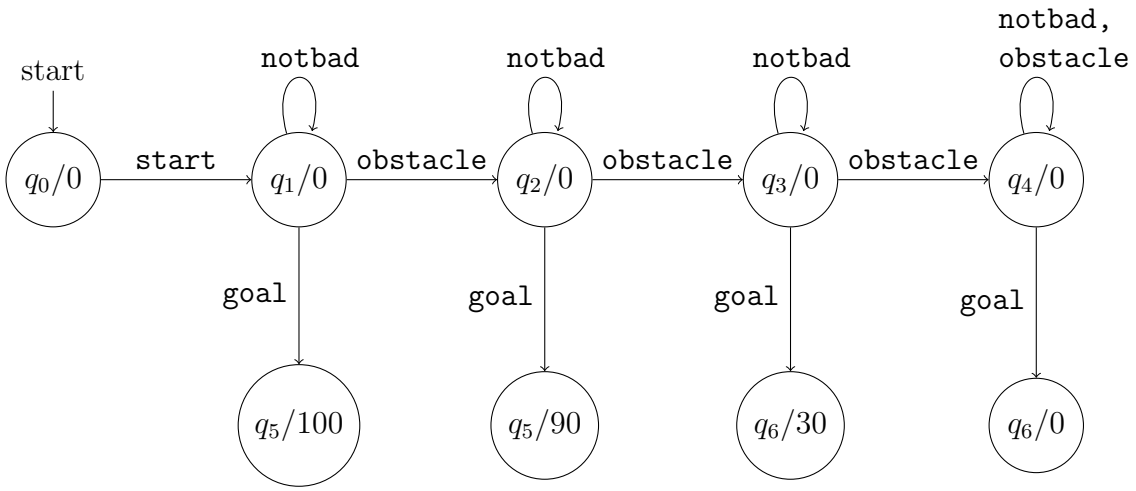


Figure 6.4: Reward Controller for broken obstacles

These combined rewards can be combined into the reward controller shown in Figure 6.4. Some transitions are left out for readability. Whenever a transition occurs that is not specified it will go to q_6, because that state acts like a dump state in this scenario. This way the size of the reward controller was optimized to 9 states.

T	5	10	15	20	25	30	35	40	45	50
States	363	3607	7557	11507	15457	19407	23357	27307	31257	35207
End states	261	2001	3976	5951	7926	9901	11876	13851	15826	17801
Result (e^{-2})	2.3438	8.6909	8.9243	8.9380	8.9391	8.9392	8.9392	8.9392	8.9392	8.9392

Table 6.2: Results for using Figure 6.4

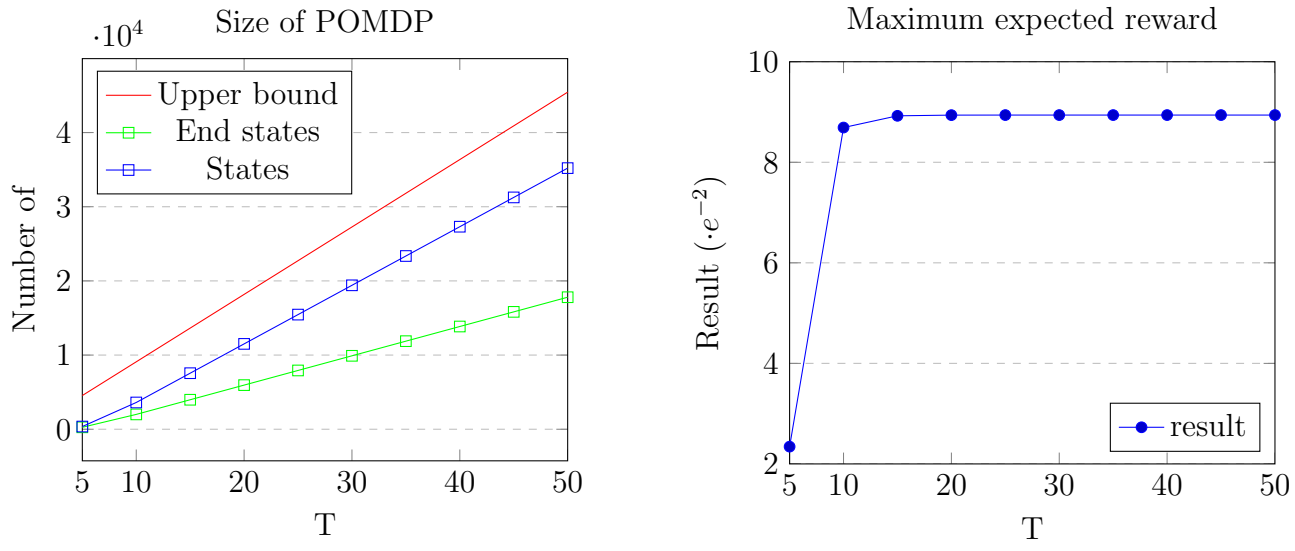


Figure 6.5: Overview of data found in Table 6.2

The size of the induced POMDP together with the expected reward with regards to T can be found in Table 6.2. We observe that for $T = 30$ the maximum result has been obtained and for $T = 25$ the reward is the same with negligible difference. For $T = 25$, we obtain a POMDP which roughly has 150 times more states than the original POMDP of which about half is an end-state with no outgoing transitions. This is still far under the upper limit of size. See, for $T = 25$ the size of the induced POMDP is upper bounded by $101 \cdot 25 \cdot 9 = 22725$. Around half of the states of the induced POMDP turn out to be deterministic final states.

Alternative approach

Since this is not simple reinforcement learning, where we are more interested in the behavior of the agent instead of the result, we cannot mimic the wanted behavior through a simple change in reward function. To tackle this problem, we could approximate the reward function in a linear fashion with regard to how high the costs/rewards are in comparison to how many obstacles we hit. Then we could simply state that for every obstacle hit, we would obtain a certain cost. Otherwise, to actually use the given costs/rewards, we need to customize the original POMDP to account for keeping track of how many obstacles are hit and rework the structure of every transition. This approach could result in a smaller POMDP, but a lot of customisation is needed.

Chapter 7

Conclusion

Currently, the definitions used for this induced POMDP is build on the notion that the history-based reward function is defined over observations. As mentioned previously this history-based reward function is not limited to observations, and can also be extended with actions or we can use the states of the underlying MDP. For this to happen, Definition 4.1 and Definition 5.3 will need to be minimally adjusted to accomodate for the new input language.

Since we are encoding a system that models history-based reward functions into the memory of the orignal POMDP, a complex reward function will lead to a large induced POMDP. The POMDP which we will work is the one described in Definition 5.7, which is bounded by not only the size of the resulting reward controller, but also by the memory limit T. Luckily, from the data gathered and dicussed in Chapter 6 we can observe that we do not necessarily need extremely large T to optain an optimal result. Furthermore, from the small data sample we can observe that the number of states that are created for the induced POMDP are often well below the upper bound.

Future Work

As mentioned previously the size complexity of solving the problem for a given POMDP $\mathcal{M}$, together with their accompanied reward controller $\mathcal{F}$ and the memory limit T is in $\mathcal{O}(|\mathcal{M}| \cdot |\mathcal{F}| \cdot T)$. This indicates that if we want to limit the size complexity, we have to either optimize the size of the reward controller $\mathcal{F}$ or find an optimal T. For optimizing the size of $\mathcal{F}$, we can look into state of the art automata learning to find a minimal RC for sequences or for finding a minimal RC for a number of regular expressions. The other optimalization step would be to find the minimal T neccesary to provide a decent result within some given bounds.

Bibliography

[1] Dimitri P. Bertsekas. *Dynamic Programming and Optimal Control, Vol. II.* Athena Scientific, 3rd edition, 2007.

[2] Arie Hordijk and Lodewijk Kallenberg. Linear programming and markov decision chains. *Management Science*, 25:352–362, 04 1979.

[3] Richard Bellman. The theory of dynamic programming. *Bulletin of the American Mathematical Society*, 60(6):503 – 515, 1954.

[4] Maor Gaon and Ronen I. Brafman. Reinforcement learning with non-markovian rewards, 2019.

[5] Ronen I. Brafman and Giuseppe De Giacomo. Regular decision processes: A model for non-markovian domains. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, pages 5516–5522. International Joint Conferences on Artificial Intelligence Organization, 7 2019.

[6] Zhe Xu, Ivan Gavran, Yousef Ahmad, Rupak Majumdar, Daniel Neider, Ufuk Topcu, and Bo Wu. Joint inference of reward machines and policies for reinforcement learning. In *Proceedings of the 30th International Conference on Automated Planning and Scheduling (ICAPS)*, pages 590–598. AAAI Press, 2020.

[7] Leonore Winterer, Ralf Wimmer, Nils Jansen, and Bernd Becker. Strengthening deterministic policies for pomdps. *CoRR*, abs/2007.08351, 2020.

[8] Edward F. Moore. Gedanken-experiments on sequential machines. In Claude Shannon and John McCarthy, editors, *Automata Studies*, pages 129–153. Princeton University Press, Princeton, NJ, 1956.

[9] Oktay Karabağ, Ayse Sena Eruguz, and Rob Basten. Integrated optimization of maintenance interventions and spare part selection for a partially observable multi-component system. *Reliability Engineering & System Safety*, 200:106955, 2020.

[10] Chia-Hsiang Chang and Robert Paige. From regular expressions to dfa's using compressed nfa's. In Alberto Apostolico, Maxime Crochemore, Zvi Galil, and Udi Manber, editors, *Combinatorial Pattern Matching*, pages 90–110, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.

[11] Thomas A. Sudkamp. *Languages and Machines: An Introduction to the Theory of Computer Science*. Addison-Wesley Longman Publishing Co., Inc., USA, 1997.

[12] S. Junges. Gridworld by storm. `https://github.com/sjunges/gridworld-by-storm`, 2021.

[13] S. Rietbergen. history-rewards-for-pomdps. `https://github.com/srnapls/history-rewards-for-pomdps`, 2021.

[14] Anass Fakir. Internship_toolchain. `https://github.com/unsigned-decimal/Internship_Toolchain`, 2020.