Radboud University Nijmegen

Faculty of Science

History-based Rewards for POMDPs

-

Thesis MSc Computing Science

Supervisor:
dr. Nils Jansen

Author:
Serena Rietbergen

Second reader:
dr. Sebastian Junges

-

Contents

Abstract

We study reward controllers (RCs), which emulate history-based reward functions specified for partially observable Markov decision processes (POMDPs). We show how to obtain these reward controllers from a number of sequences or regular expressions together with their respective rewards. We combine this reward controller together with the POMDP it is defined over to create an induced POMDP. This allows us to obtain a policy for obtaining the maximum expected reward for the original POMDP.

Chapter 1

Introduction

In reinforced learning we are often shown models in which the agent gets a reward for certain behavior. This behavior can also include a reward that will given only after a certain time or only given a certain history. These history-based reward functions have been studies quite a lot. More specifically, a number of results have been found to obtain a policy concerning the expected optimal reward for these history-based reward functions for MDPs[11][2]. However for POMDPs, the task of obtaining an policy that ensures the maximum reward (or minimum cost) for history-based reward functions, is computative heavy. The belief MDP representing a POMDP can be continuous, storing the history observed for sequences and not knowing how long the sequence needs to be can be memory intensive.

talk a bit about these?

Removing the history-based function would remove some of this memory intensive process. For this we present so-called Reward Controllers, which are finite automata which represent the history-based reward function used for the POMDP. This reward controller can then be combined with the original POMDP, to create an induced POMDP. This induced POMDP doesn't need to store the relevant history anymore, which was necessary for the original POMDP. The new reward function is then only based on the current state and the action, allowing us to use known methods to calculate the optimal expected reward and to find a policy.

Problem Formulation

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

Structure

In chapter 2 we present the preliminaries. Chapter 3 portrays all relevant definitions and shows the background information used. In chapter 4 we present the Reward Controllers and how to obtain them from a given history-based reward function. In chapter 5 we show the newly induced POMDP which is obtained from the original POMDP together with the Reward Controller that represents the related history-based reward function.

Chapter 2

Preliminaries

Set Theory

Let S be any countable set, then $|S|$ denotes the cardinality. We let S^* and S^ω denote the set of finite and infinite sequences over S, respectively. For a sequence $\pi \in S^*$ we can denote the length by $|\pi|$.

Let an alphabet Σ be a a finite set consisting of letters. A word is defined as a sequence of letters $w = w_1 w_2 \ldots w_n \in \Sigma^*$. A language L is a subset of all possible words given an alphabet Σ, so $L \subseteq \Sigma^*$. Let λ denote the empty word, so $|\lambda| = 0$.

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

Probability Theory

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

X is a random variable with a N outcomes, denoted as $x_1, x_2, \ldots, x_n$. Then we can define the expectation of X as

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

Chapter 3

Background

3.1 Finite Automata

Deterministic Finite Automata

Simple deterministic processes can be easily modeled with the help of a finite-state machine. Specifically, if we are interested in wether an input string should be accepted, 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.

Example

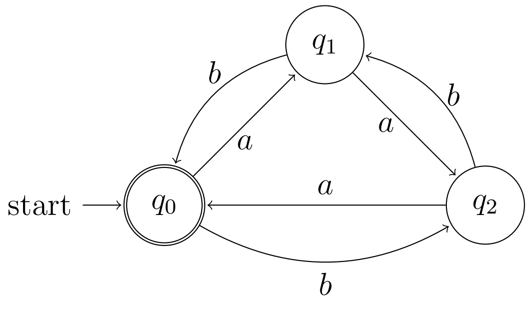


Figure 3.1: DFA over $\Sigma = \{a, b\}$ which accepts words if the number of a's and b's are equal modulo 3.

Since we are interested in wether an input string should be accepted or not, we are specifically interested in how a DFA handles certain words and where a DFA will finish after reading a word. Since DFAs are deterministic, this can be easily described.

Definition 3.2. 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. 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. Thus $L(D) = \{w \in \Sigma^* \mid \delta^*(q_0, w) \in F\}$.

Moore machine

A Moore machine is a finite state machine, similar to the previously mentioned DFA. As we have seen, 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 while 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. 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.

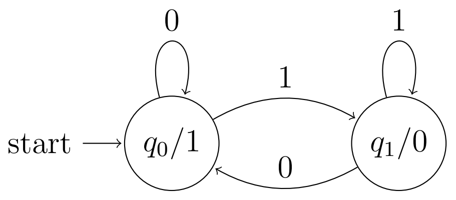
Example

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 have obtain a new sequence based on the outputs encoded in the states.

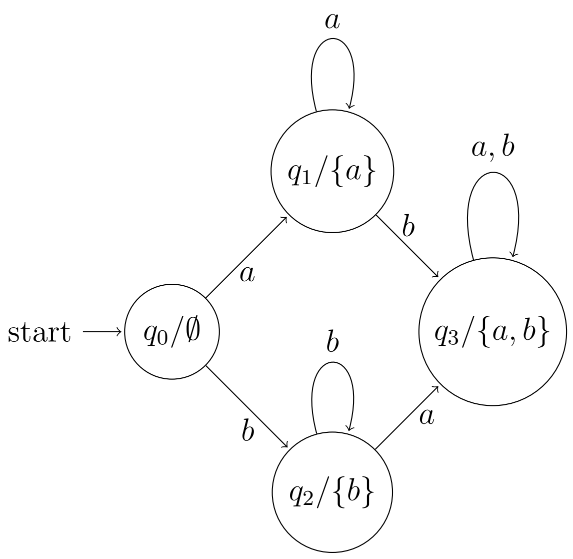
In Figure 3.2a we have for $\Sigma = O = \{0, 1\}$ a machine that inverts a given sequence. The inverted sequence will however also be preceded by a 1 per construction. For example, when we pass through the sequence 1110, we obtain 10001.

Another usage of Moore machines is to only obtain the output after we are done with reading the sequence. For example, 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 the sequence aaa, we will then obtain $\{a\}$.



(a) Inverting sequence



(b) Returns set of used letters

3.2 Markov Processes

A machine is not always defined deterministically. Instead, a process can 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 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 discudding discrete-time Markov processes.

Markov chain

Given a simple stochastic process, that conforms to the Markov property as seen in Definition 3.5 is called a Markov chain. This is simply a set of events, which are connected by some given probabilities.

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 $T : S \to \Pi(S)$.

Note that P ,the probabilistic transition function, can also be represented as a matrix.

Example

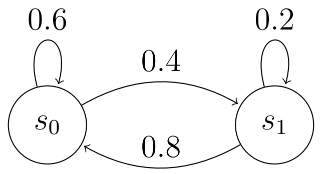


Figure 3.3: A simple Markov chain

The transition matrix for this Markov chain is $\begin{pmatrix} 0.6 & 0.4 \\ 0.8 & 0.2 \end{pmatrix}$.

Markov decision processes

While we can see Markov chains as stochastic processes without outside influence, we can also take a look at these processes where we allow outside influence. This is done by extending the Markov chain with a set of actions, allowing for this influence.

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;
- $T : S \times A \to \Pi(S)$, the probabilistic transition function.

Note that given $s \in S, a \in A$, we assign a probability distribution over S through $T(s,a)$. To obtain the probability of ending up in a certain state s' when starting in state s and performing action a, we simply calculate $T(s,a,s')$ which we obtain through $T(s,a)(s')$.

The *available actions* for a state s are given by $A(s) = \{\, a \in A \mid \exists s' \in S : T(s,a,s') > 0\}$. We can give 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 $s_1a_1s_2a_2\ldots s_{n-1}a_{n-1}s_n \in (S \times A)^* \times S$. To obtain the last state of a trajectory we can use the following

$$last(s_1a_1s_2a_2\ldots s_{n-1}a_{n-1}s_n) = s_n$$

Example

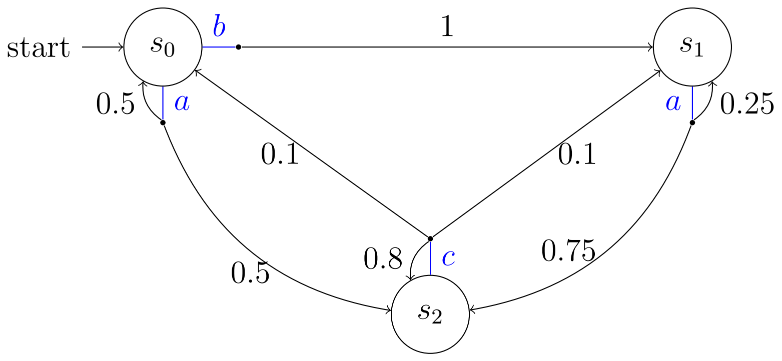


Figure 3.4: MDP

Rewards

We can extend MDPs with a *reward function* R which assign a reward for taking some action in state. Let us first look at simple reward functions which can determine a reward based on the current state, action and obtained state, independent of its history. The most conventional notation for this simple reward function is $R : S \times A \to \mathbb{R}$, where we consider the current state and the taken action. Another possible definition is $R : S \times A \times S \to \mathbb{R}$, where in $R(s, a, s')$ we consider the specific transition from s to s' by using action a, or $R : S \to \mathbb{R}$ where in $R(s)$ we only consider the visited state s.

When modeling complex systems drawn from real world problems, we often encounter that obtaining a certain reward is not only dependent on the current events but also on the states (and actions) that were seen 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 3.8 (Policy)**.** A policy for a MDP M is a function $\pi : (S \times A)^* \times S \to \Pi(A)$, which maps a trajectory to a probability distribution over all actions.

We call a policy *memoryless* if the function only considers the last state in deciding the actions. Note that we write $\pi(s, a)$ for $\pi(s)(a)$, which gives us the probability of performing action a given the state s. We can apply these types of policies to a MDP to remove the non-determinism, resulting in an induced Markov chain.

Definition 3.9. 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 as follows

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

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, we are mainly interested in the immedate rewards. However, the closer γ is to one, we are more interested in all the future rewards.

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 ss, given a policy pi, is defined as

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

With this information, we can obtain the optimal policy through

$$\pi^* = \operatorname*{arg\,max}_{\pi \in \Pi(A)^S} J^\pi(s_I) \tag{3.1}$$

We can solve the problem of how to obtain a policy for the optimal expected reward recursively, due to Bellman's principle of optimality[1].

$$\begin{aligned} V_0(s) &= 0 \\ V_n(s) &= \max_{a \in A} \big[R(s, a) + \gamma \sum_{s' \in S} T(s, a, s') V_{n-1}(s')\big] \end{aligned}$$

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

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

Partial observability

Having full observability over a system makes it simple to calculate the optimal expected reward and there has been numerous research finding the optimal expected reward or finding a policy that ensures a certain wanted property. However, when modeling a lot of real world issues, we unfortunately do not have all the information readily available to us.

Take for example a machine that breaks down over a period and needs repairs[7]. 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* the entire system, which only providus us with partial knowledge of the machine. We can model these types of systems with the help of partially observable MDPs.

Definition 3.10 (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.

Let $O^{-1} : \Omega \to 2^S$ be the inverse function of the observation function - $O^{-1}(o) = \{s \in \mid O(s) = o\}$ - in which we simply obtain all states in S that have observation o. Without loss of generality we assume that states with the same observations have the same set of available actions, thus $O(s_1) = O(s_2) \Rightarrow A(s_1) = A(s_2)$.

Since the actual states in a trajectory of the hidden MDP are not visible to the observes, we argue about an *observed trajectory* of the POMDP $\mathcal{M}$. This is not consist of a sequence of states and actions, but instead a sequence of observations are actions, thus 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, like so

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

Example

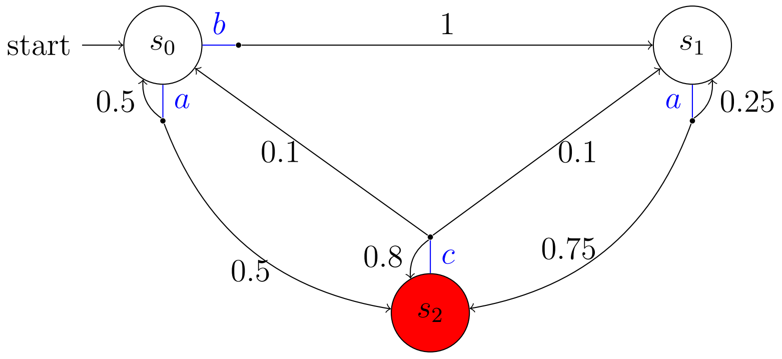


Figure 3.5: Example POMDP where $\Omega = \{\texttt{white}, \texttt{red}\}$

Rewards

Just like for MDPs, we can extend POMDPs with a reward function. These can be function over the hidden MDP, and thus the reward function will remain an extension of the MDP. Or, the function can be based on the observations of the states instead of the states themselves. So instead of $R : S \times A \times S \to \mathbb{R}$, we can have that $R : \Omega \times A \times \Omega \to \mathbb{R}$. This way we only base the reward function on the observation of the current and future state together with the action taken. There is also a possibility for history-based reward function based on the observations, e.g. $R : \Omega^* \to \mathbb{R}$.

Policy

Defining a policy over a POMDP is a trickier, since we only obtain the observation of a state and not all the information. So now we have to make a decision over what action to take, given only the observations. This provides us with observation-based trategies.

Definition 3.11. An observation-based strategy of a POMDP $\mathcal{M}$ is a function $\pi : \mathit{ObsSeq}^{\mathcal{M}} \to \Pi(A)$ such that

$$\mathit{supp}(\pi(O(s_1a_1 \ldots s_{n-1}a_{n-1}s_n))) \subseteq A(\mathit{last}(s_1a_1 \ldots s_{n-1}a_{n-1}s_n))$$

for all $s_1a_1 \ldots s_{n-1}a_{n-1}s_n \in (S \times A)^* \times S$.

Belief MDP

As stated in the previous section, POMDPs only have partially observable, i.e. we do not have full knowledge over the system. More specifically, we do not have full information available over which state we are in, only the observation. Since we cannot argue about which exact state we are in, we argue about which state we *belief* we are in.

Definition 3.12 (Belief state)**.** A belief state $b : S \to [0,1]$ is a probability distribution over S. For every state s, $b(s)$ denotes the probability of currently being in state s.

When we perform an action after observing a certain observation and then observe (a different) observation, we need to update our belief about the state we are in. After the transition, we have a transitioned into a new belief state. This update is called a *belief update*.

Definition 3.13 (Belief update)**.** Given the current belief state b, then after performing action $a \in A$ and then observing observation $o \in \Omega$, we update the belief state. The updated belief state $b^{a,o}$ can be calculated as

$$b^{a,o}(s') = \frac{\Pr(o \mid s', a)}{\Pr(o \mid a, b)} \sum_{s \in S} T(s, a, s')b(s)$$

When working with these probabilistic processes, it is preferable to have full observability into the system to accurately calculate certain wanted properties. To remove this partial observability, we can transform a POMDP into a fully observable *belief MDP*.

Definition 3.14 (Belief MDP)**.** For a POMDP $\mathcal{M} = (M, \Omega, O)$ where $M = (S, s_I, A, T)$ as defined above, the associated belief MDP is a tuple (B, A, τ, ρ) where

- $B = \Pi(S)$, the set of belief states;

- A, the same set of actions;
- $\tau : B \times A \to \Pi(B)$, the transition function where

$$\tau(b, a, b') = \Pr(b' \mid a, b) = \sum_{o \in \Omega} \Pr(b' \mid a, b, o) \cdot \Pr(o \mid a, b)$$

Note that the belief MDP needs to have an initial belief state. This initial state distribution b_0 is sometimes also an extension of the original POMDP. If only the actual initial state is given as s_I, then we can simply calculate b_0 as $b_0(s_I) = 1$ and $b_0(s) = 0$ for all $s \in S \setminus \{s_I\}$. We also have to acknowledge that working with these belief representations of a given POMDP may yield a continuous, instead of a discrete, model.

Reward

If the POMDP is extended with a reward function R, the belief MDP will obtain a reward function ρ. If $R : S \times A \to \mathbb{R}$, then $\rho : B \times A \to \mathbb{R}$ where $\rho(b, a) = \sum_{s \in S} b(s) R(s, a)$.

Solving for optimal reward

Now we can also compute the value function. Let $Pr(o \mid a, b) = \sum_{s \in S} \sum_{s' \in O^{-1}(o)} T(s, a, s')b(s)$ be in the following function.

$$\begin{aligned} &V_0(b) = 0 \\ &V_n(b) = \max_{a \in A} \left[\rho(b, a) + \gamma \sum_{o \in \Omega} Pr(o \mid a, b) V_{n-1}(b^{a,o})\right] \end{aligned}$$

Chapter 4

Reward Controllers

The problem with history-based rewards is that we have to remember all the previous observations and only then calculate the associated reward, instead of simply calculating the reward per transition.

In this chapter we are going to take the history-based reward function and transform it into something more tangible. We are going to transform it into an abstract machine that keeps track of its history and rewards associated.

First we'll 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 sort of history-based reward function $R : \Omega^* \to \mathbb{R}$ which belongs to some POMDP $\mathcal{M}$. 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 $\pi \in \Omega^*$, we want that the state we end up in to contain the reward associated with π. This is in some sense the same as a Moore machine, except for the fact that instead of applying σ to every state we encouter, we only use σ on the last state obtained.

Definition 4.1. 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.

Note that when we discuss $|\mathcal{F}|$, we are discussing the number of states in the reward controller $\mathcal{F}$, so $|\mathcal{F}| = |N|$. 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 state. Which is why we we use the following definition, similarly as what we have defined for DFAs.

Definition 4.2. We define $\delta^* : N \times \Omega^* \to N$ where $\delta^*(n, w)$ denotes the state we end up after reading word `seq` starting from state n as follows

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

TO WRITE: since we let the base cost be zero, note that we cannot allow for minimizing policies. convert cost to profit, given a start-base.

Implementation

For the implementation of a Reward Controller, we have used the DFA construction as described in [9]. This DFA construction allows us to map a value for every state. For simple DFAs this would be a `0` for non-acceptance, and a `1` for acceptance. In our case, we can simply encode the appropriate reward in their states.

4.2 From a list of sequences

Let's 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. 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 is connected to zero.

We only want to obtain any of the rewards if their associated observation sequence has been observed in its entirety. 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 and we end up in a certain node n, we obtain the reward $\sigma(n)$ in that node. It's 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 Non-Markovian 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: $R : \Omega^* \to \mathbb{R}$
2:     $n_I \leftarrow$ new Node()                                ▷ initial node
3:     $n_F \leftarrow$ new Node()                                ▷ dump node
4:     path($n_I$) = $\lambda$
5:     $N \leftarrow \{n_I, n_F\}$
6:     for all seq = $o_1 o_2 \dots o_k$ in sequences do
7:         $n \leftarrow n_I$
8:         for $i \leftarrow 1, \dots, k$ do
9:             if $\delta(n, o_i)$ is undefined then
10:                $n' \leftarrow$ new Node()                     ▷ create new memory node
11:                path($n$) = $o_1 \dots o_i$
12:                $N \leftarrow N \cup \{n'\}$
13:                $\delta(n, o_i) \leftarrow n'$
14:            $n \leftarrow \delta(n, o_i)$                      ▷ update memory node
15:        $\sigma(n) \leftarrow R$(seq)                          ▷ set reward
16:    for all $n \in N$ do                      ▷ makes $\delta$ and $\sigma$ deterministic
17:        for all $o \in \Omega$ do
18:            if $\delta(n, o)$ is undefined then                ▷ useless transition
19:                $\delta(n, o) \leftarrow n_F$
20:        if $\sigma(n)$ is undefined then
21:            $\sigma(n) \leftarrow 0$
22:    return $(N, n_I, \Omega, \mathbb{R}, \delta, \sigma)$
```

We start by creating an initial node in Line 2 and a dump node in Line 3. The idea is that, since the reward controller is deterministic, if we need to determine the reward of a sequence that is (for example) longer than a known sequence (with reward), we don't want to end in the node in which the reward is encoded. Thus these zero-reward sequences are passed along to a node which will only consist of self-loops and will have a reward of zero encoded to them.

Then for every sequence which we are given, we walk through it. If we then come across a transition which isn't defined yet, 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 existed, we simply update the memory node. After we are done with reading the sequence, we simply encode the reward into the node itself in Line 15.

Then since the reward controller needs to be deterministic, we set the other undefined values. Every other transition that hasn't been made yet, will be transferred to the dump node as mentioned above 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 represents a directed acyclic graph. This indicates 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's 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* $\texttt{seq} \in \Omega^*$*, let* $r = R(\texttt{seq})$ *be its associated reward. Then* $\sigma(\delta^*(n_I, \texttt{seq})) = r$.

Proof. Given a sequence `seq`, we set n to be the node we end up in, i.e. $n = \delta^*(n_I, \texttt{seq})$.
Now 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.
If $n \in N \setminus \{n_F\}$, we can obtain the unique path to node n through $\texttt{path}(n)$. We know that this is equal to `seq`, so the associated reward is thus $R(\texttt{path}(n)) = R(\texttt{seq}) = r$. □

We observe that the number of memory nodes $|N|$ of the newly created reward controller $\mathcal{F}$ is bounded by $|\Omega|^k + 1$, where $k = \max\limits_{seq \in sequences} |seq|$.
We acknowledge that this can lead to quite large reward controllers. Please note that using state of the art automata learning, the sizing of the reward controller obtained for a number of sequences can be decreased.

Example

Say 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 the 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)

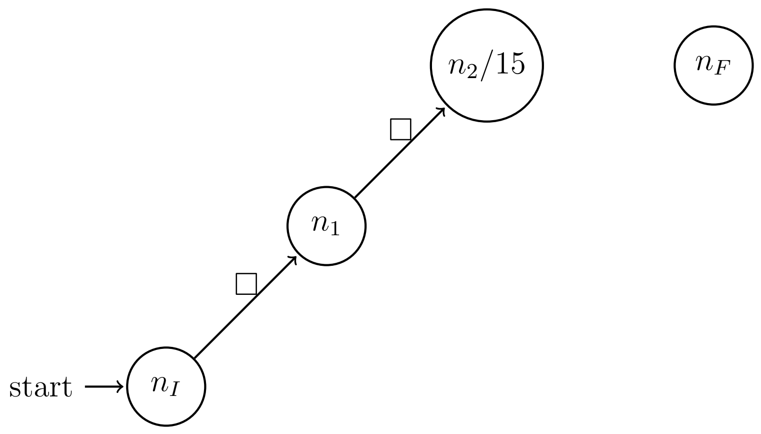


Figure 4.1: Reward controller after sequence (1)

After sequence (2)

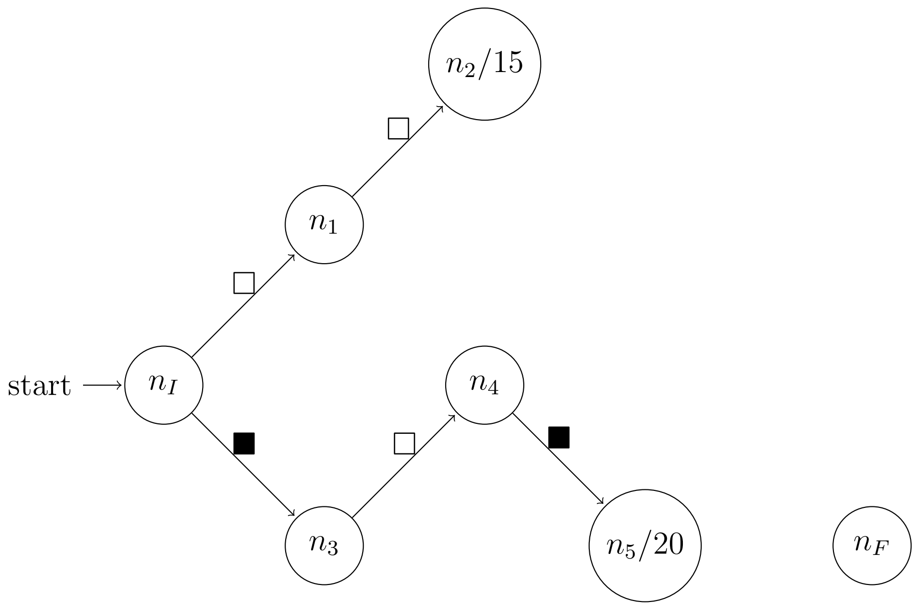


Figure 4.2: Reward controller after sequence (1) and (2)

After sequence (3)

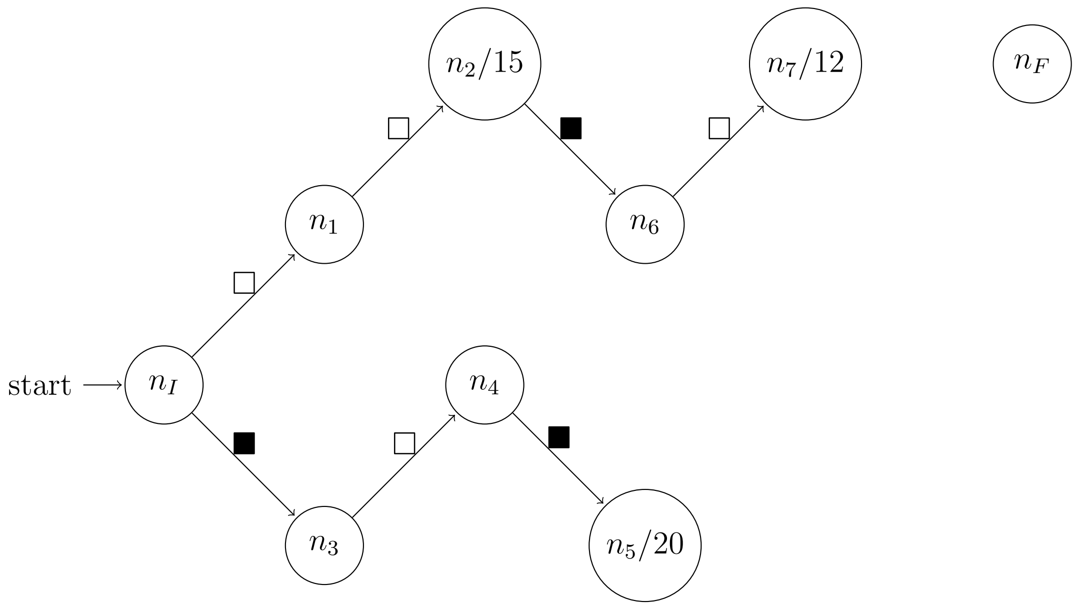


Figure 4.3: Reward controller after sequence (1), (2) and (3)

After sequence (4)

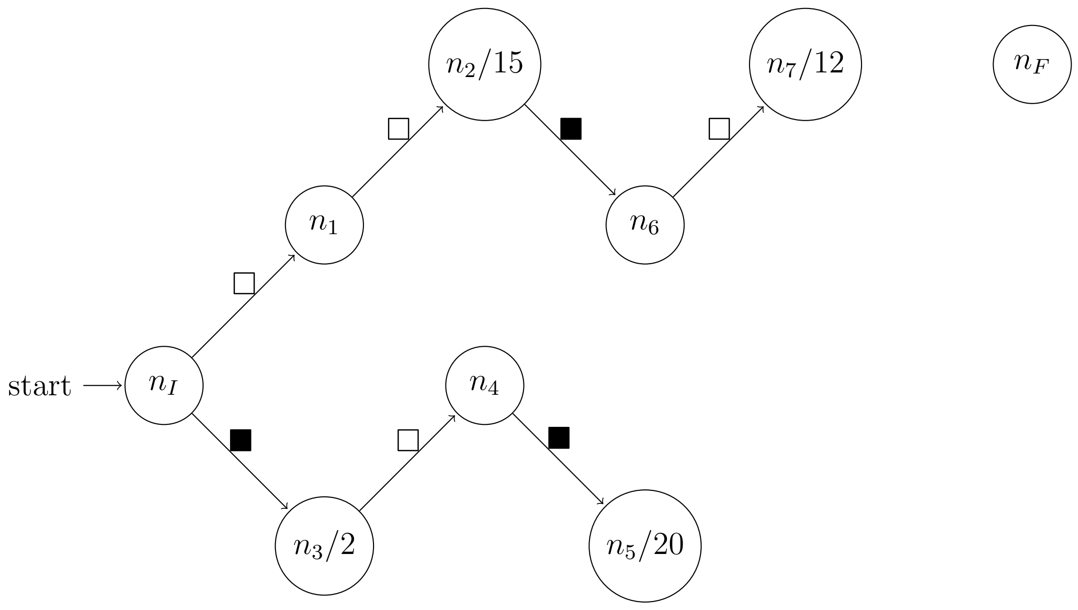


Figure 4.4: Reward controller after sequence (1), (2) and (3)

Finalized Reward Controller

Now we complete the reward controller by completing the rest of the transitions. 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 wasn't defined.

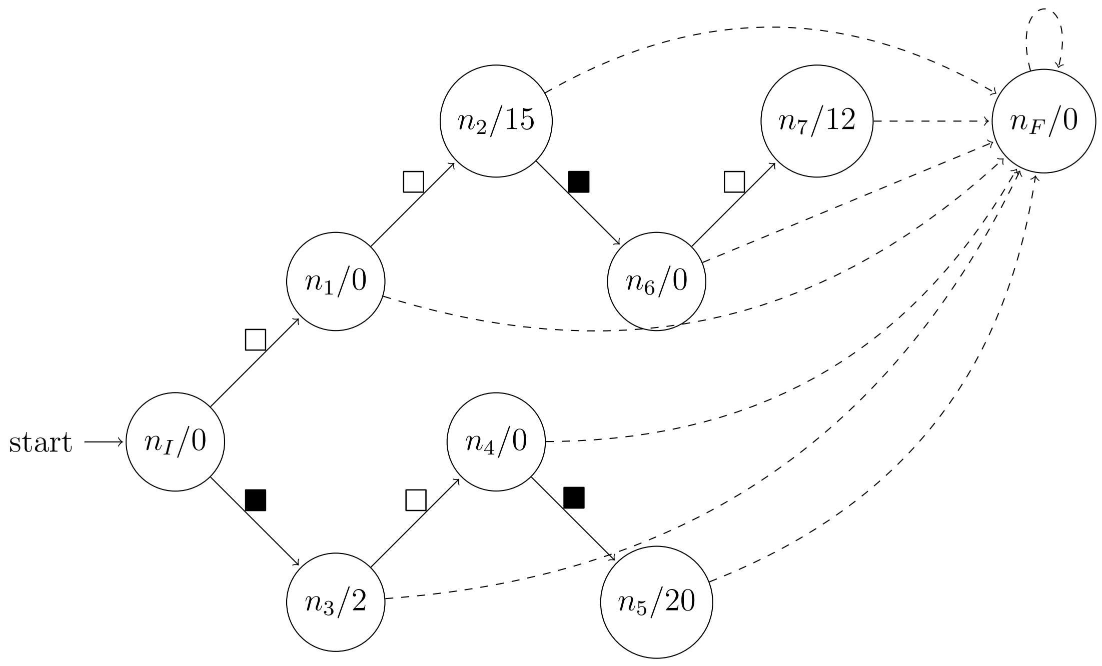


Figure 4.5: Final reward controller

Implementation

In `code/reward_controller_seq.py` we have the following:

`reward_controller_from_sequences(sequences,omega)`
Given a dictionary in the shape of $\{\texttt{seq}_1 : r_1,\ \texttt{seq}_2 : r_2, \dots,\ \texttt{seq}_n : r_n\}$ together with the set Ω representing the input alphabet, it will return a reward controller as described in Algorithm 1.

4.3 From regular expressions

TO WRITE: introduction into why using regular expressions for rewards

Given a number of regular expressions over observations defined as $e_1, e_2, \dots, e_n$ together with their respective rewards $r_1, r_2, \dots, r_n \in \mathbb{R}$. Let us define a reward function R that maps the regular expression to their respective reward, in other words $R(e_i) = r_i$.

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 would be is to create a DFA that is generated by the regular expression given. This can be done through simply turning the regular expression into a Non-Deterministic Finite Automaton (with λ-transitions) and then turning that into a DFA or using other known methods[3]. All that is left for a single regular expression is to keep track of the rewards associated to their final states.

So given the n regular expression, 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. And then per construction we have that $L(D_i) = L(e_i)$.

Note that since we want to obtain a reward controller, 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 `seq` $\in \Omega^*$ gets accepted by D_i, we have to make sure that the state it ends up in - i.e. the final state(s) - has the reward encoded in its state(s). This is done by the following definition.

Definition 4.5. Let $R_A : Q_1 \cup Q_2 \cup \cdots \cup Q_n \to \mathbb{R}$ be a 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. In other words,

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

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 4.6. 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
- $\delta(\langle q_1, q_2, \ldots, q_n \rangle, a) = \langle \delta_1(q_1, a), \delta_2(q_2, a), \ldots, \delta_n(q_n, a) \rangle$
- $F = \{\langle q_1, q_2, \ldots, q_n \rangle \mid \exists i \in \{1, 2, \ldots, n\} : q_i \in F_i\}$

Lemma 4.7. *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 4.6. Then we $L(D) = L(D_1) \cup L(D_2) \cup \ldots L(D_n)$.*

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 obtained product DFA together with the associated rewards of the states.

Definition 4.8. Given a (product) DFA $N = (Q, q_0, \Omega, \delta, F)$ and the associated reward function R_A, we define the induced reward controller $\mathcal{F} = (N, n_I, \Omega, \mathbb{R}, \delta_{\mathcal{F}}, \sigma)$ as follows

- $N = Q$
- $n_I = q_0$
- $\delta_{\mathcal{F}} = \delta$
- $\sigma : Q \to \mathbb{R}$ where $\sigma(\langle q_1, q_2, \ldots, q_n \rangle) = \sum_{i=1}^{n} R_A(q_i)$

Note that the 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 then 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 $\texttt{seq}$.

Lemma 4.9. *Given $e_1, e_2, \ldots, e_n$ a sequence of regular expression together with their associated rewards $r_1, r_2, \ldots, r_n$, let D be the product automaton as defined in Definition 4.6 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 4.8 given D. We say that for all possible words $\texttt{seq} \in \Omega^*$ the following holds:*

$$\sigma(\delta^*(n_I, \texttt{seq})) = \sum_{e \in \{e_i \mid \texttt{seq} \in L(e)\}} R(e_i)$$

Proof.

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

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 4.8. For Equation (4.3) we use that fact that in Definition 4.5 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 4.5 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 $i \in \{1, \dots, n\}$, we rewrite the term in Equation (4.7). Finally in Equation (4.8) we simply rewrite the term under the sum. $\square$

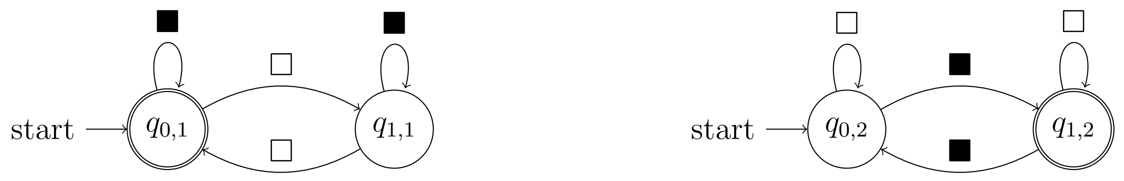
Logically, the size of the reward controller obtained through a series of n automata is bounded by $|D_1||D_2|\dots|D_n|$, where D_i is the automata obtained through the regular expressions e_i. When implementing this, make sure to minimize the automata whenever possible to decrease the sizing of the reward controller.

implementatio
rele-
vant?

Example

Let's say we are given 2 regular expressions. One is that an even number off $\square$ gives a reward of 10 and the other states that an odd number of $\blacksquare$ 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 $\square$

(b) DFA for regular expression for odd number of $\blacksquare$

Figure 4.6

Then we create the product automaton as defined in Definition 4.6. The result can be seen in Figure 4.7.

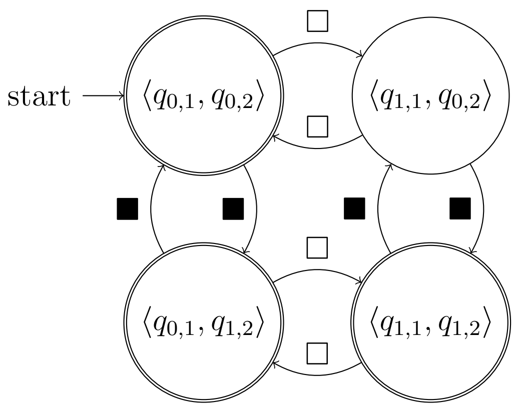

start
$\langle q_{0,1}, q_{0,2}\rangle$
$\langle q_{1,1}, q_{0,2}\rangle$
$\langle q_{0,1}, q_{1,2}\rangle$
$\langle q_{1,1}, q_{1,2}\rangle$
□
■

Figure 4.7: Product DFA for both regular expressions

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

$$\begin{aligned}
R_A(q_{0,1}) &= 10 \\
R_A(q_{1,1}) &= R_A(q_{0,2}) = 0 \\
R_A(q_{1,2}) &= 15
\end{aligned}$$

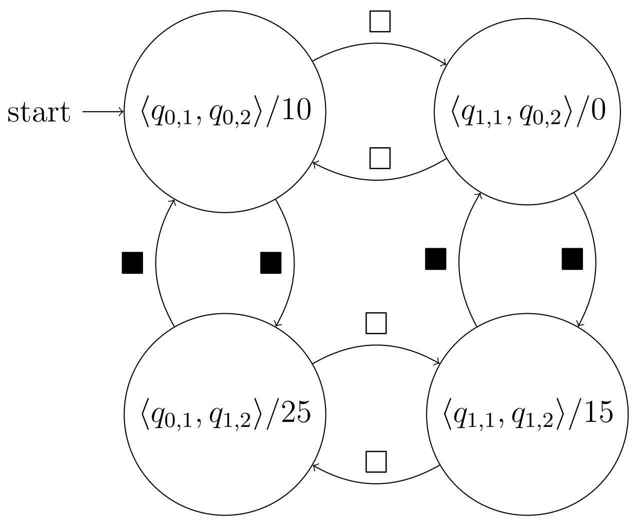

start
$\langle q_{0,1}, q_{0,2}\rangle/10$
$\langle q_{1,1}, q_{0,2}\rangle/0$
$\langle q_{0,1}, q_{1,2}\rangle/25$
$\langle q_{1,1}, q_{1,2}\rangle/15$
□
■

Figure 4.8: Reward Controller for R

Implementation

We used [10] as a base for creating a DFA from a given regular expression. The regular expression needs to have the following grammar (this can also be seen in `code/regex-to-dfa/grammar/RegEx.g4`):

```
prog : (regex newline)*;

regex : regex '*'       #kleene-star
  | regex regex         #concatenation
  | regex '|' regex     #alternation
```

```
   | ID                   #identifier
   | 'λ'   #epsilon
   | '(' regex ')'        #parenthesis
   ;

newline : '\n';

ID: [a-zA-Z0-9];
WS: [\t\r ]->skip;
```

In `code/reward_controller_regex.py` we have the following:

`rename(D)`
Transform the given reward controller D into one that ensures that the states are labeled with numbers from 0 to $|D| - 1$.

`regex_to_dfa(regex, omega)`:
Given a regular expression conform to the syntax as defined above and the input language Ω over which it is defined, we transform it into a DFA.

`union(machines, rewards)`:
Given a list n of `machines` $(D_1, D_2, \ldots, D_n)$ and a list or n rewards $(r_1, r_2, \ldots, r_n)$, we create the induced product DFA according to Definition 4.6. Then we (create and) return the induced reward controller as defined in Definition 4.8.

`reward_controller_from_regex(info,omega)`::
Given n regular expressions together with their associated reward in dictionary $(\{e_1 : r_1, e_2 : r_2, \ldots e_n : r_n\})$, together with the input language Ω specified, we return the reward controller representing the information given.

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, together with the original POMDP. 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. Firstly, 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. In Section 5.5 we talk about on how to limit the computational process by only allowing sequences up to a certain length. In Section 5.6 we talk a bit about the implementation regarding this new induced POMDP.

5.1 Extended POMDP

Given a POMDP with has a history-based reward function, we want to obtain the related reward at any certain moment. But there is not a action in the model that ensures that the model stops and we can 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. The extended POMDP for a given POMDP $\mathcal{M} = (M, \Omega, O)$ where $M = (S, s_I, A, T_M)$ is a new POMDP $\widetilde{\mathcal{M}} = (\widetilde{M}, \Omega', O')$ where

- $\widetilde{M} = (S', s_I, A', T_{M_t})$, the hidden MDP where:
  - $S' = S \cup \{s_F\}$, the finite set of states;
  - $A' = A \cup \{\texttt{end}\}$, the finite set of actions;
  - $T_{M_t} : S' \times A' \to \Pi(S')$, the probabilistic transition function defined as:

$$T_{M_t}(s, a, s') = \begin{cases} 1 & \text{if } s' = s_F \text{ and } a = \texttt{end} \\ T_M(s, a, s') & \text{otherwise} \end{cases}$$

- $\Omega' = \Omega \cup \{o_F\}$
- $O' : S' \to \Omega'$ where

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

We now also need to adjust the reward function for the extended POMDP, to accomodate for the new information.

Definition 5.2. 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

$$\widetilde{R}(o_1 o_2 \dots o_n, \texttt{end}) = R(o_1 o_2 \dots o_n)$$

5.2 Induced POMDP

We will now combine the reward controller $\mathcal{F}$ - representing the history-based reward function - with the given related PODMP to obtain an induced POMDP where we map the memory into the system. This ensures that we don't have to keep the observation sequence in memory.

Definition 5.3. 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;
  - $T_{M_{\mathcal{F}}} : S_{\mathcal{F}} \times A \to \Pi(S_{\mathcal{F}})$, 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 and the set of memory nodes N. We only allow a transition between the new states if they align properly with regards to the transitions of the reward controller $\mathcal{F}$. We always look at the observation of the state we reach when transitioning between states. This is also the reason why the initial state $s_{I,\mathcal{F}}$ is not $\langle s_I, n_I \rangle$, but instead $\langle s_I, \delta(n_I, O(s_I)) \rangle$. This is because when we start with this process, we always start in a state. will always look at the observation of the **next** state.

Now we can extend this induced POMDP $\mathcal{M}_{\mathcal{F}}$ as presented in Definition 5.1, yielding $\widetilde{\mathcal{M}_{\mathcal{F}}}$, with the adjusted reward function in which we obtain the related reward when we enter the final state through action `end`.

Definition 5.4. The reward function $\mathcal{R} : S'_{\mathcal{F}} \times A \to \mathbb{R}$ 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 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.

TO WRITE: size and complexity

5.3 Example

Let us observe the POMDP as seen in Figure 5.1 over which we have the history-based reward function $R(\texttt{even number of } \square) = 10$. The corresponding reward controller can be seen in Figure 5.2.

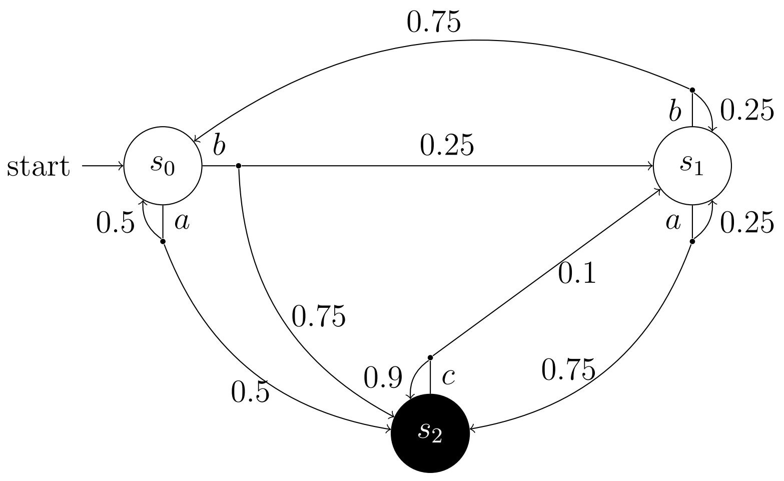


Figure 5.1: POMDP with $\Omega = \{\square, \blacksquare\}$

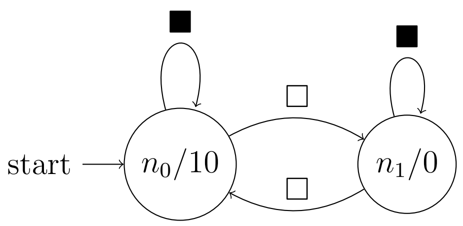
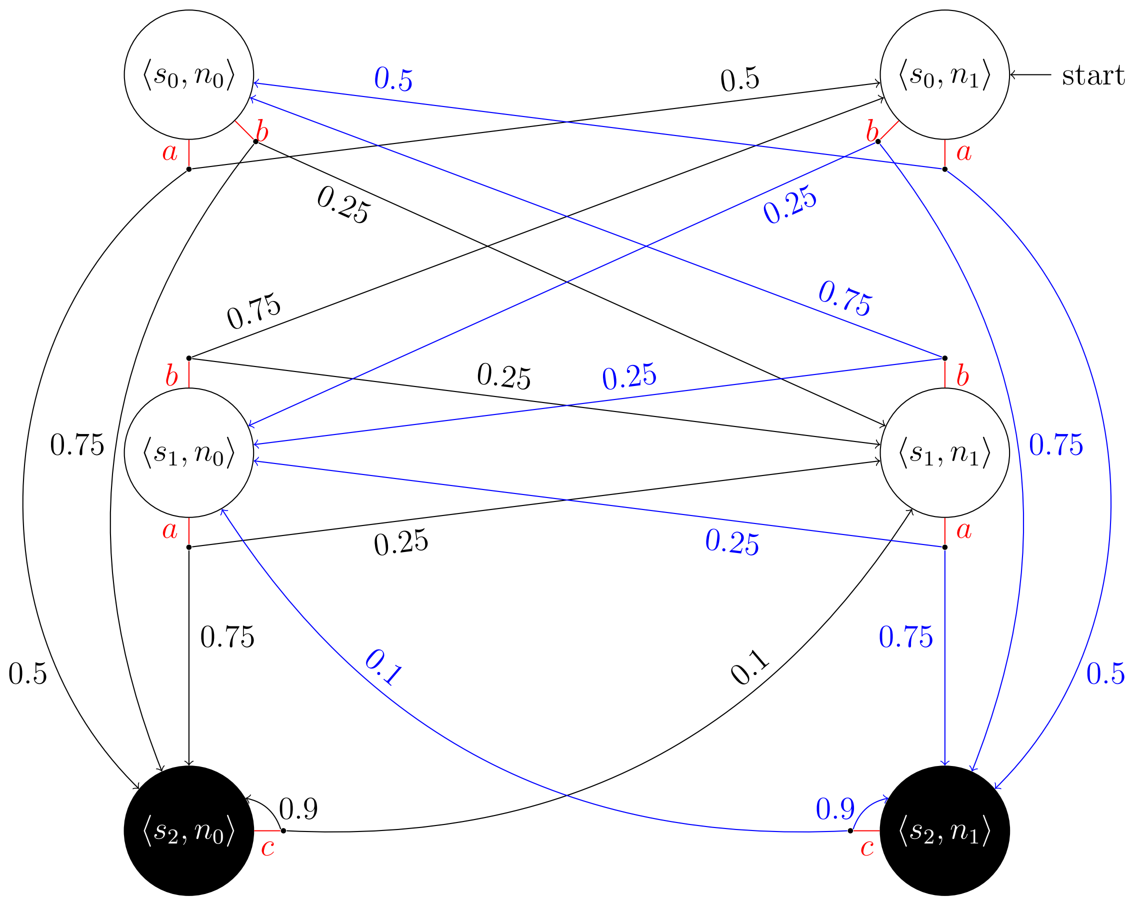


Figure 5.2: Reward Controller for R

The induced POMDP corresponding with Definition 5.3 can be seen in Figure 5.3. If we `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 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 $\square$.



$\langle s_0, n_0 \rangle$
$\langle s_0, n_1 \rangle$
start
$\langle s_1, n_0 \rangle$
$\langle s_1, n_1 \rangle$
$\langle s_2, n_0 \rangle$
$\langle s_2, n_1 \rangle$
a
b
c
0.5
0.25
0.75
0.1
0.9

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

ensure for better read-ability

5.4 Optimal policy

TO WRITE: rework 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.

$$\begin{tikzcd}
\mathcal{M} \arrow[r, "\text{Definition 4.8}", "\text{or Algorithm 1}"'] \arrow[rr, dashed, bend left] \arrow[d, "\text{Definition 5.1}"'] & \mathcal{F} \arrow[r, dashed, "\text{Definition 5.3}"] & \mathcal{M}_{\mathcal{F}} \arrow[d, "\text{Definition 5.1}"] \\
\widetilde{\mathcal{M}} & & \widetilde{\mathcal{M}_{\mathcal{F}}}
\end{tikzcd}$$

Figure 5.4: Overview

If we can obtain an optimal policy for $\widehat{\mathcal{M}_{\mathcal{F}}}$ we automatically obtain the optimal policy for $\widehat{\mathcal{M}}$. This can then be easily transformed into an optimal policy for $\mathcal{M}$.

The optimal policy of $\widehat{\mathcal{M}_{\mathcal{F}}}$ optimal policy π for $\widetilde{\mathcal{M}_{\mathcal{F}}}$ also works for $\widetilde{\mathcal{M}}$. something with same observation sequences, for all actions.

TO WRITE: rewrite this, that it's the sum over all rewards (still works since the only non-zero reward is the final transition

Lemma 5.5. *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 5.4. Then for all $\langle s, n\rangle \in S_{\mathcal{F}}$*

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

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

Proof.

$$\begin{aligned} R(o_1 o_2 \dots o_n) &= \sigma(\delta * (n_I, o_1 o_2 \dots o_n))\text{Lemma 4.4 and Lemma ??} \\ &= \sigma(n) \\ &= \widetilde{\mathcal{R}_{\mathcal{F}}}(\langle s, n\rangle, \texttt{end})\text{Definition 5.4} \end{aligned}$$

Note that furthermore the only requirement of s is that $O(s) = o_n$. □

Note that the original POMDP $\mathcal{M}$ does not have the action `end`, which does exist for every extended POMDP used for obtaining the optimal policy.

5.5 Limiting the observation sequence

However, when we try to calculate the probability for which we end up in s_F ($P[\mathtt{F} s = s_F]$), we notice that this probability is not equal to 1, because there is no absolutele certainty that `end` will ever be performed and thus no certainty that we end up in s_F. Luckily, we can enforce that the model only allows observation sequences up to a certain natural number T, such that for every observation sequence $o_1 o_2 \dots o_n$ we know that $n \leq T$.

Definition 5.6. We can extend the underlying 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 underlying MDP (S', s'_I, A, T'_M) where

- $S' = S \times \{0, \dots, T\}$
- $s'_I = \langle s_I, 0\rangle$
- $T'_M : S' \times A \to \Pi(S')$ 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 reward function $R : S' \times A \times S' \to \mathbb{R}$ will only look at the states of S and will be transformed into

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

In other words, we add a simple counter that keeps track of the number of actions allowed. At any point we are allowed to go to the final state, but when the counter reaches T, we force the model to enter the final state. This method enforces the process to always finish in the final state.

Now since that we know that $P[\texttt{F}\, s = s_F] = 1$, we can calculate the expected maximum reward for when we enter that state. In other words, we can use model checking tools like `PRISM` to calculate $R_{max}[\texttt{F}\, s = s_F]$.

TO WRITE: size and complexity

5.6 Implementation

In `code/POMDP.py` we have the folowing

`__init__(M, Omega, O, prism)`
We either initialize a POMDP with M, Ω, O, where M is a MDP or with a `prism`-file. When only given the `prism`-file as argument, we build the model with the help of stormpy[6]. Then we obtain all information about the hidden MDP M, the observation space and the observation function from this model.

In `code/induced_POMDP.py` we have the folowing functions

`__init__(prism, R, regex, T)`
Given the `prism`-file of the original POMDP, reward function R, `regex` indicating wether R is defined over regular expression (and thus having the value `true`) or over sequences (having the value `false`) and the counter T as seen in definition Definition 5.6. It then creates the induced POMDP as defined in Definition 5.3. This induced POMDP is then extending according to Definition 5.1. Finally, to be able to compute nice properties we finally limit the model using Definition 5.6.

`create_prism_file()`
After inializing the induced POMDP with the constructor, we can then obtain the information of said POMDP. This function transform this POMDP into a `prism`-file. This also includes the `rewards` which are defined in Definition 5.6.

After we have executed the function `create_prism_file()`, we can run `storm` with the following property

```
Rmax=?  [F "end"]
```

to obtain the maximum expected reward.

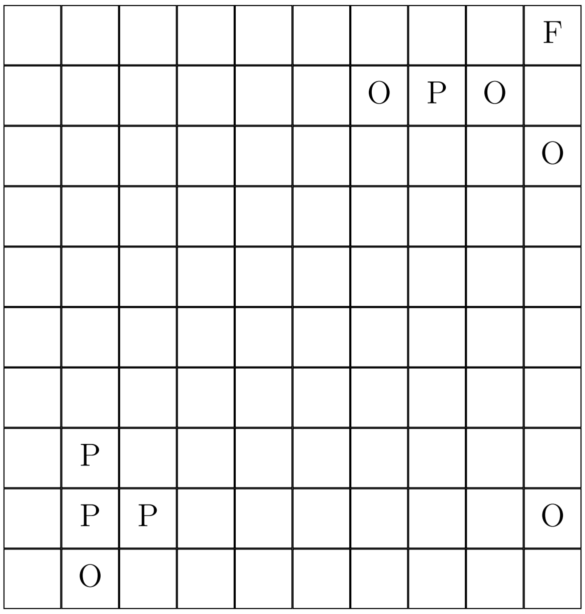
Chapter 6

Case Study

6.1 Problem

Based on a grid world found in [5], we have a 10×10 grid, in which there are five obstacles (O), where an agent needs to find the exit (F).

Let us have a Roomba in a five by five grid, 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 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 north, east, south or west. See Figure 6.1 for an overview of this problem



									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.

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 finish without driving over any small item. Thus, let our history-based reward function be:

Reaching the goal, without encountering any obstacles : 1

This can be represented in the following regular expression:

`start notbad* goal` : 1

allowing us with the following reward controller:

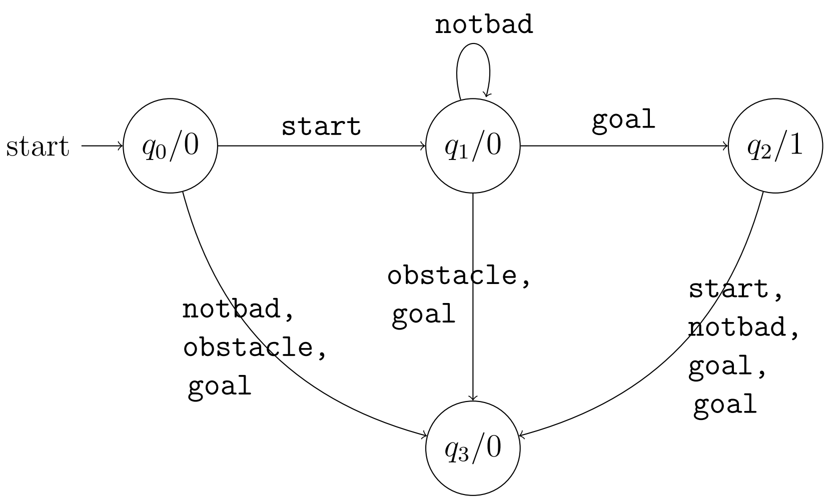


Figure 6.2: Reward Controller for reinforced learning

Logically speaking, there will not occur a transition after the `goal` has been observed by designation of the model. However, since the reward controller is a deterministic automaton, we specify this.

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

Another way to tackle the problem of reaching the charging station without hitting any obstacle, is to give a penalty for observing an obstacles. A simple way to approximate this problem would be to simply specify in the `prism`-program that every time you observe an obstacle, you get a penalty of 50 for example. To find a policy for this, we ask to minimalize the involved costs.

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. To approximate this, let's take a look at the following approach. So 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.

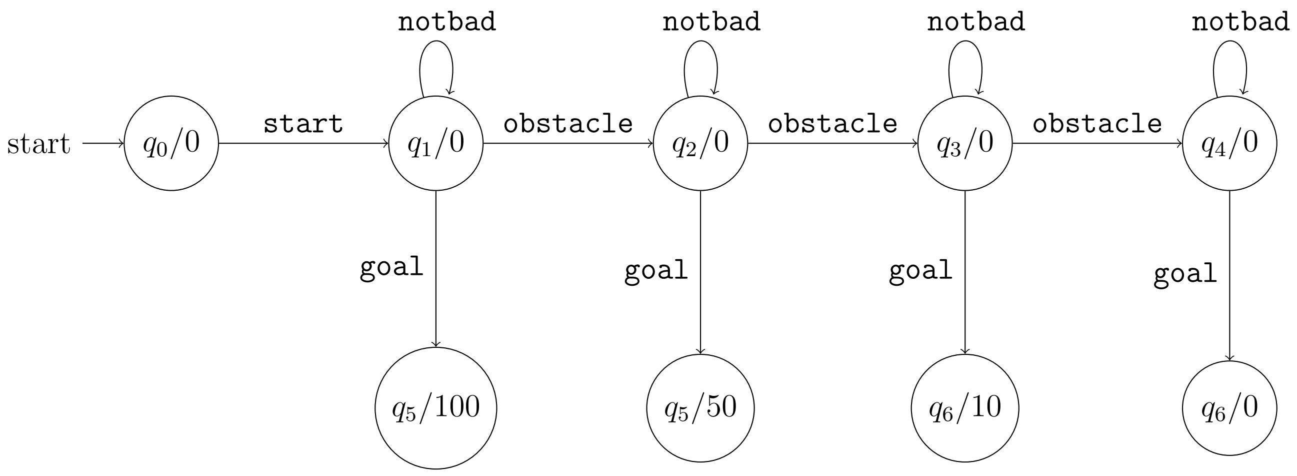


Figure 6.3: Reward Controller for broken obstacles

In Figure 6.3 some transitions are missing. Whenever a transition occurs that isn't specified it will go to a dump state, one that has a zero reward encoded.

6.2 Results

For testing this scenario, we have to pick the maximum observation sequence length T as we have seen in Definition 5.6. We have run the benchmark toolchain for creating policies for POMDPs[4] on a system which runs on `Ubuntu 20.04.3 LTS`. The system has a `Intel Core i3-6006U` processor and 4GB of RAM.

This tool solves the POMDP given a specific property that we want to check. For our problem we want to find the maximum expected reward after reading a sequence up to length T, given the reward function as stated above. The tool finds this value and then finds a policy that complies with this value. A pMC will be generated for finding an optimal policy. The size of this is represented in number of states and number of transitions.

The empty slots depict the fact that the size of the pMC was too big to be solved. For $N = 5$ we observe that the maximum expected reward had been reached already when $T = 50$.

move tables to appendix?

For $N = 10$ we can see that the maximum expected reward has been reached for when $T = 35$. This can be decreased if we allow for a certain error margin.

The sizing of the pMC that is needed to solve for the policy for optaining the maximum expected reward seems to be linear, in terms of states as wel ass transitions. Since the solver for finding the optimal expected reward is a bit more than linear, it would be pertinent to find the smallest T possible, given a certain error margin for the expected maximum reward.

Chapter 7

Conclusion

TO WRITE: conclusion and future work

any history-based reward function can lead to extreme growth in the final induced pomdp.

the reward controller defined in 4 currently only looks at sequence of observations, but this can also be adjusted to a sequence of actions, states, observations or a combination of those.

7.1 Future Work

TO WRITE: mentioned in 4, use state of the art automata learners to decrease the size of the final pomdp

optimizing the creation of the extended induced pomdp, a lot of states are created that are obsolete (especially if you have a placement action in your pomdp, see case study)

finding optimal T

Bibliography

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

[2] 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.

[3] 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.

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

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

[6] S. Junges and M. Volk. moves-rwth/stormpy. `https://github.com/moves-rwth/stormpy`, 2020.

[7] Oktay Karabağ, Ayse 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, 03 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] M. Vazquez-Chanlatte. dfa. `https://github.com/mvcisback/dfa`, 2021.

[10] K. V. Vinayaka. regex-to-dfa. `https://github.com/OpenWeavers/regex-to-dfa`, 2018.

[11] 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.