Radboud University Nijmegen

Faculty of Science

History-based Rewards for POMDPs

-

Thesis MSc Computing Science

Supervisor:
dr. Nils Jansen

Author:
Serena Rietbergen

Second reader:
- -

-

Contents

Abstract

Chapter 1

Introduction

Motivating Example

Problem Formulation

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

Contribution

Structure

first preliminaries, then definitions chapter 4 we present reward controllers, i.e. give a structure on how to model the reward function. in chapter 5 we combine the original pomdp with the created reward controller to obtain an markov model again.

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 ϵ denote the empty word, so $|\epsilon| = 0$.

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 $supp(\psi) = \{s \in S \mid \psi(s) > 0\}$.

TO WRITE: random variable, expected value

Chapter 3

Background

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. 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

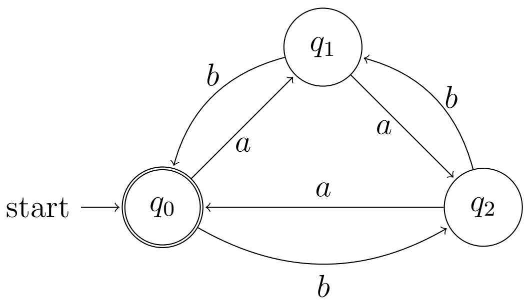


Figure 3.1: DFA over $\Sigma = \{a, b\}$ which accepts words if the number of a's and b's are equal modulo 3.

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

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

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

3.1.2 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 [2].

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.

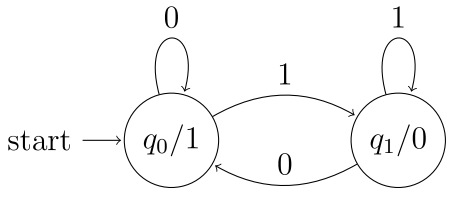
Example

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.

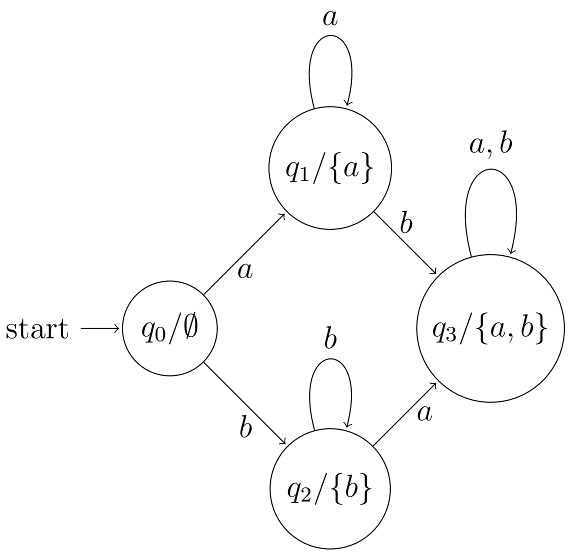
In Figure 3.2a we have for $\Sigma = O = \{0, 1\}$ a machine that inverts a given sequence. The inverted sequence will however also be preceded by a 1 per construction. For example, when we pass through the sequence 1110, we obtain 10001.

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 $s_0, s_1, \dots, s_{n-1}, s_n \in S$:

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

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.

3.2.1 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

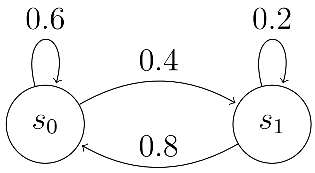


Figure 3.3: A simple Markov chain

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

3.2.2 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* π of a MDP is realization of the stochastic process performed by the MDP denoted by the finite sequence $s_1 a_1 s_2 a_2 \ldots s_{n-1} a_{n-1} s_n \in (S \times A)^* \times S$. To obtain the last state of a trajectory we can use the following

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

Example

write a nice example

Rewards

We can extend MDPs with a *reward function* R which assign a reward - usually in $\mathbb{R}$ - 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 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;
- $R : (S \times A)^* \to \mathbb{R}$ - which looks at the finite (sub)trajectory without the last state, or;
- $R : (S \times A)^* \times S \to \mathbb{R}$ - 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 maximum reward (or minimize the costs involved). However, just obtaining the optimized expected 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 $\sigma : (S \times A)^* \times S \to \Pi(A)$, which maps a trajectory π to a probability distribution over all actions.

We call a policy *memoryless* if the function only considers $last(\pi)$ in deciding the actions.

TO WRITE: this only works for simple reward functions

TO WRITE: where does gamma come from

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

3.2.3 Partial observability

TO WRITE: Introduce pomdp

Definition 3.9 (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 - $O^{-1}(o) = \{s \in| O(s) = o\}$ - 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 $(\Omega \times A)^* \times \Omega$. The set of all possible finite observed trajectories of 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

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

Rewards

TO WRITE: pomdp with reward, different kind of rewards

Policy

TO WRITE: since we now only know the observation we change the policy

Definition 3.10. An observation-based strategy of a POMDP $\mathcal{M}$ is a function $\sigma : ObsSeq^{\mathcal{M}} \to \Pi(A)$ such that $supp(\sigma(O(\pi))) \subseteq A(last(\pi))\ \forall \pi \in (S \times A)^* \times S$.

Solving for optimal reward

TO WRITE: refer to belief mdp

3.2.4 Belief MDP

Definition 3.11 (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.

TO WRITE: introduction to beflief update

Definition 3.12 (Belief update)**.** Given the current belief state b, then after performing action $a \in A$ and then observing observation o, we update the belief state. The updated belief state $b^{a,o}$ can be calculated as

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

TO WRITE: connection to belief mdp

Definition 3.13 (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 set of actions;
- $\tau : B \times A \times B$, the transition function where

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

TO WRITE: note that belief mdp are continuous time and not discrete

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 $\rho(b, a) = \sum\limits_{s \in S} b(s) R(s, a)$.

Solving for optimal reward

$Pr(o \mid a, b) = \sum\limits_{s \in S} \sum\limits_{s' \in O^{-1}(o)} T(s, a, s') b(s)$

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

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 $(N, n_I, \Omega, \mathbb{R}, \delta, \lambda)$, 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 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 π starting from state n as follows

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

Implementation

`https://github.com/mvcisback/dfa`.

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 $\pi_1, \pi_2, \dots, \pi_n$ together with their associated real valued rewards $r_1, r_2, \dots, r_n$.

Definition 4.3. Given the observation sequences $\pi_1, \pi_2, \dots, \pi_n$ 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

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

In R we simply connect the observation sequence π_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 state 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 state n, we obtain the reward $\sigma(n)$ in that state. 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: $\quad n_I \leftarrow$ `new Node()` $\triangleright$ initial node
3: $\quad n_F \leftarrow$ `new Node()` $\triangleright$ dump node
4: $\quad \texttt{path}(n_I) = \epsilon$
5: $\quad N \leftarrow \{n_I, n_F\}$
6: $\quad$ **for all** $\pi = o_1 o_2 \ldots o_k$ in sequences **do**
7: $\qquad n \leftarrow n_I$
8: $\qquad$ **for** $i \leftarrow 1, \ldots, k$ **do**
9: $\qquad\quad$ **if** $\delta(n, o_i)$ is undefined **then**
10: $\qquad\qquad n' \leftarrow$ `new Node()` $\triangleright$ create new memory node
11: $\qquad\qquad \texttt{path}(n) = o_1 \ldots o_i$
12: $\qquad\qquad N \leftarrow N \cup \{n'\}$
13: $\qquad\qquad \delta(n, o_i) \leftarrow n'$
14: $\qquad\quad n \leftarrow \delta(n, o_i)$ $\triangleright$ update memory node
15: $\qquad \sigma(n) \leftarrow R(\pi)$ $\triangleright$ set reward
16: $\quad$ **for all** $n \in N$ **do** $\triangleright$ makes δ and σ deterministic
17: $\qquad$ **for all** $o \in \Omega$ **do**
18: $\qquad\quad$ **if** $\delta(n, o)$ is undefined **then** $\triangleright$ useless transition
19: $\qquad\qquad \delta(n, o) \leftarrow n_F$
20: $\qquad$ **if** $\sigma(n)$ is undefined **then**
21: $\qquad\quad \sigma(n) \leftarrow 0$
22: $\quad$ **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 state 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 state 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 states, so per Definition 4.3 we encode those to zero in Line 21.

We observe that the number of memory nodes $|N|$ of the newly created reward controller $\mathcal{F}$ is bounded by $|\Omega|^k + 1$, where $k = \max_{seq \in sequences} |seq|$.

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.

let maarten read this

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

Proof. Let us state that after reading π, we end up in state n, i.e. $n = \delta^*(n_I, \pi)$. 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 `path`(n). We know that this is equal to π, so the associated reward is thus $R(\texttt{path}(n)) = R(\pi) = r$. □

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)

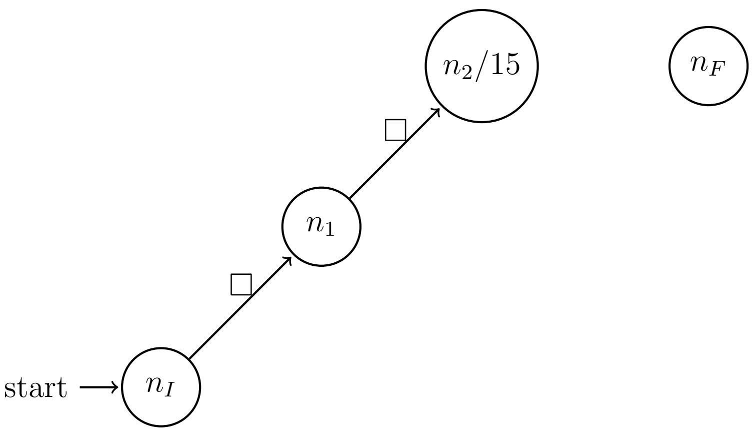


Figure 4.1: Reward controller after sequence (1)

After sequence (2)

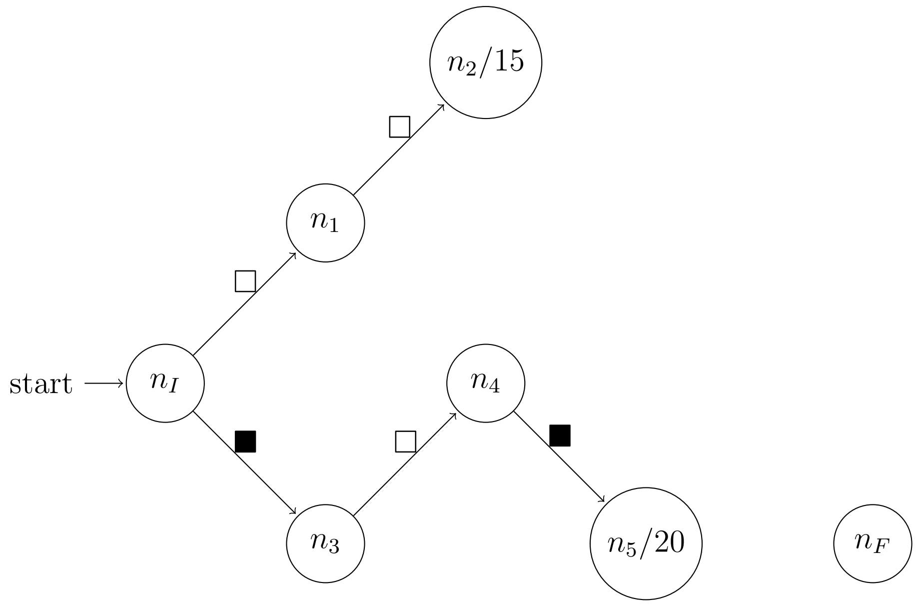


Figure 4.2: Reward controller after sequence (1) and (2)

After sequence (3)

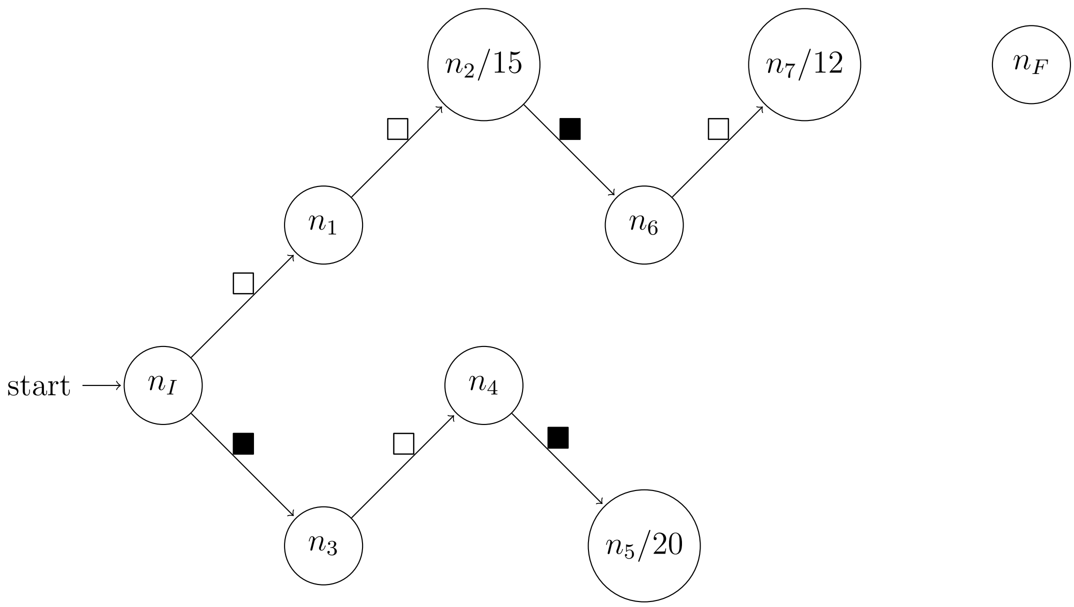


Figure 4.3: Reward controller after sequence (1), (2) and (3)

After sequence (4)

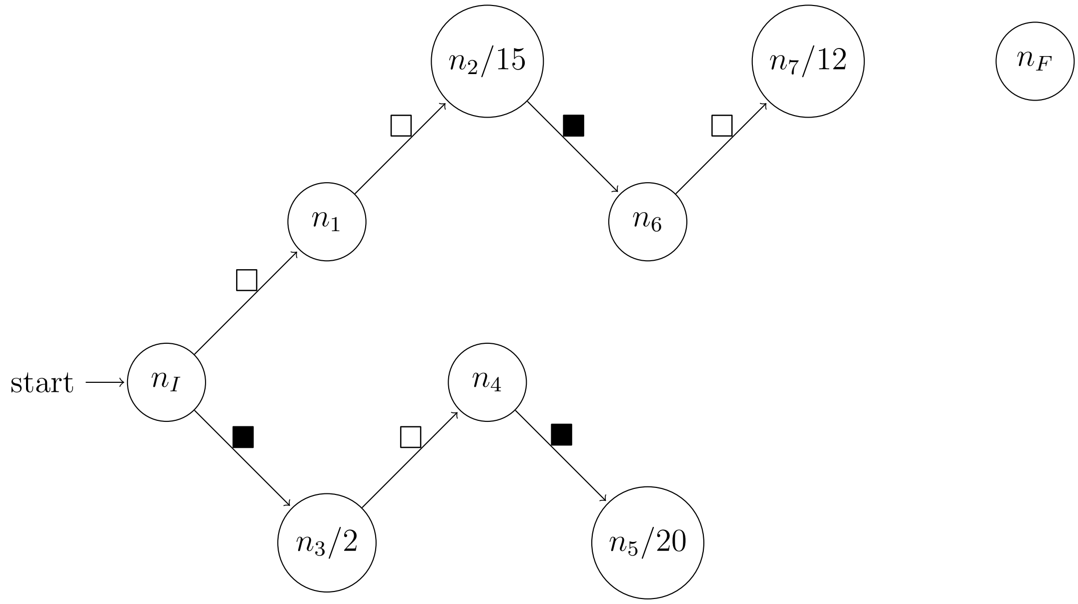


Figure 4.4: Reward controller after sequence (1), (2) and (3)

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 λ wasn't defined.

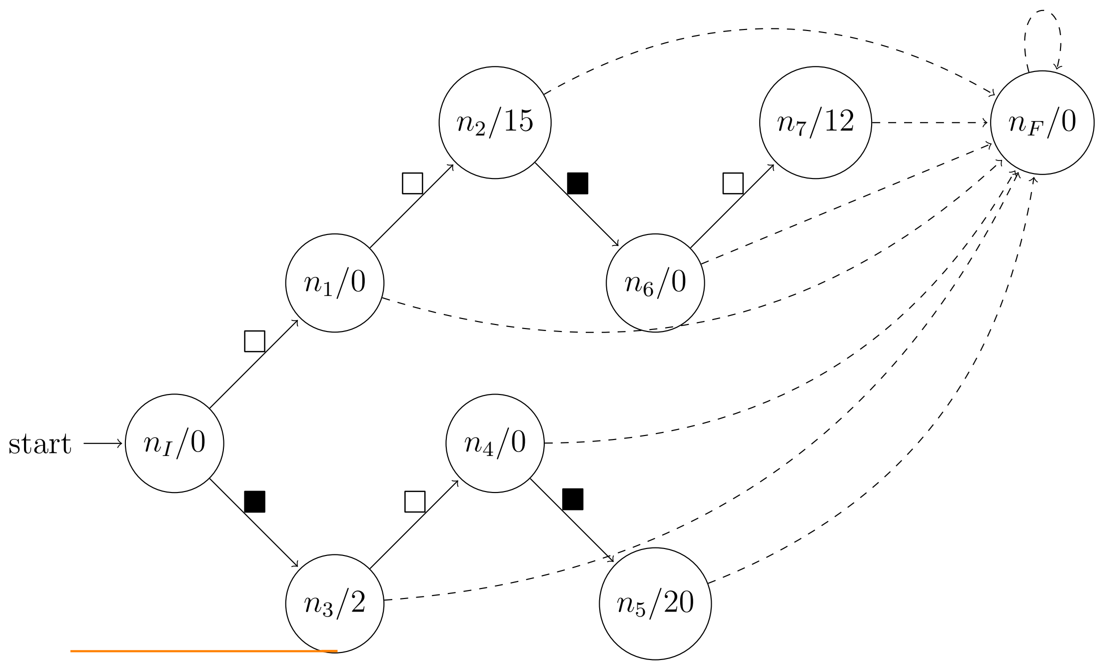


Figure 4.5: Final reward controller

fill this in pls ty

Implementation

4.3 From regular expressions

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 ϵ-transitions) and then turning that into a DFA or using other known methods[1]. 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 $pi \in \Omega^*$ 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 $R_A : Q_1 \cup Q_2 \cup \dots \cup Q_n \to \mathbb{R}$ be a function that maps any state q of all the state spaces of $D_1, D_2, \dots, D_n$ 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 $D_1, D_2, \dots, D_n$ where $D_i = (Q_i, q_{0,i}, \Sigma, \delta_i, F_i)$ is a tuple $D = (Q, q_0, \Sigma, \delta, F)$ where

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

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 $\pi \in \Omega^*$ 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 $\pi \in \Omega^*$ the reward controller obtains the combination of rewards depending on the final state after having read π.

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 $\pi \in \Omega^*$ the following holds:*

$$\sigma(\delta^*(n_I, \pi)) = \sum_{e \in \{e_i | \pi \in L(e)\}} R(e_i)$$

Proof.

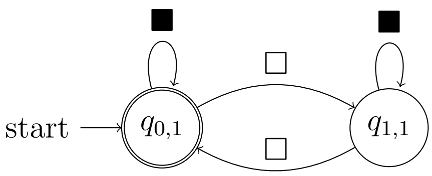
$$
\begin{aligned}
\sigma(\delta^*(n_I, \pi)) &= \sigma(\langle q1, q2, \ldots, q_n \rangle) && (4.1)\\
&= \sum_{i}^{n} R_A(q_i) && (4.2)\\
&= \sum_{\substack{i \in \{1,\ldots,n\}\\ q_i \in F_i}} R_A(q_i) && (4.3)\\
&= \sum_{\substack{i \in \{1,\ldots,n\}\\ q_i \in F_i}} R(e_i) && (4.4)\\
&= \sum_{\substack{i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

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 π. 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 $i \in \{1, \ldots, n\}$, we rewrite the term in Equation (4.7). Finally in Equation (4.8) we simply rewrite the term under the sum. □

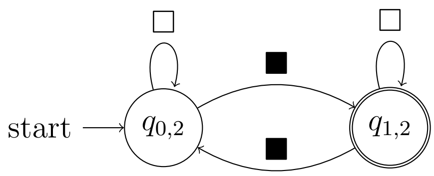
Example

Let's say we are given 2 regular expressions. One is that an even number off □ gives 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 4.6. The result can be seen in Figure 4.7.

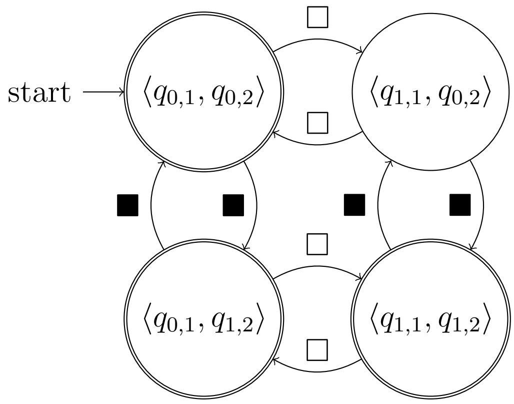


Figure 4.7: Product DFA for both regular expressions

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

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

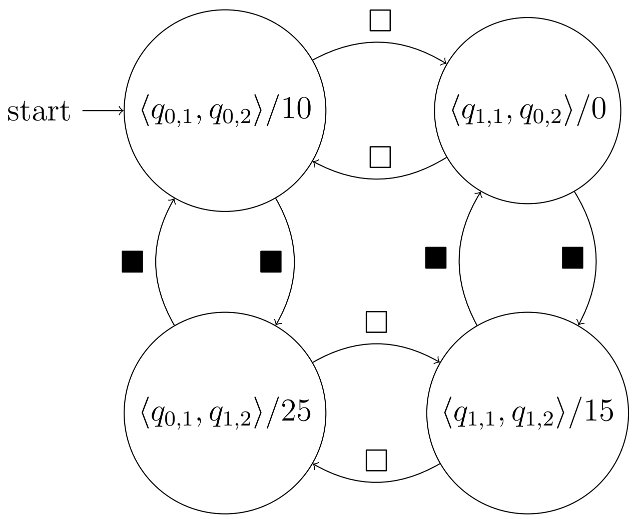


Figure 4.8: Reward Controller for R

Implementation

talk about this

Chapter 5

Obtaining policy

find a better chapter name

TO WRITE: introduction

TO WRITE: please not that all the text here is placeholder, just notes that need to be processed

The resulting POMDP is a product construction between the original POMDP and the reward controller representing the history-based reward function.

Since the reward in encoded in the states themself in $\mathcal{F}$, we dont want to just simple encode them in the POMDP in the state as well. If we were to do this, then we'd get the reward every time we passed over the state. We only want to obtain the relevant reward when we are *finished* with the process. This is why we used the extra action `end` to mark the end of the process. Then, when we wish the process to end, we simply execute the action `end` and will then obtain the reward that was encoded in the relevant state from $\mathcal{F}$ where we finished upon. This new state s_F only contains deterministic loops, ensuring that the process ends there.

5.1 Definition

Definition 5.1. The induced POMDP for reward controller $\mathcal{F} = (N, n_I, \Omega, \mathcal{R}, \delta, \lambda)$ on a POMDP $\mathcal{M} = (M, \Omega, O)$ where $M = (S, s_I, A, T_M)$ is a tuple $\mathcal{M}_\mathcal{F} = (M_\mathcal{F}, \Omega', O')$ where

- $M_\mathcal{F} = (S', s'_I, A', T_{M_\mathcal{F}}, \mathcal{R})$, the hidden MDP where:
  - $S' = S \times N \cup \{s_F\}$, the finite set of states;
  - $s'_I = \langle s_I, \delta(n_I, O(s_I))\rangle$, the initial state;
  - $A' = A \cup \{\texttt{end}\}$, the finite set of actions;
  - $T_{M_\mathcal{F}} : S' \times A' \to \Pi(S')$, the probabilistic transition function defined as:

$$T_{M_\mathcal{F}}(s, \texttt{end}, s_F) = 1 \text{ for all } s \in S'$$

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

  - $\mathcal{R} : S' \times A' \times S' \to \mathbb{R}$ where

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

- $\Omega' = \Omega \cup \{o_F\}$, the observation spate
- $O' : S' \to \Omega'$, the observation function where

$$O'(s) = \begin{cases} O(s') & \text{if } s = \langle s', n \rangle \\ o_F & \text{if } s = s_F \end{cases}$$

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.

5.2 Implementation

TO WRITE: the python part, where we simple combine the information of the reward controller together with the pomdp (without reward) together to create a new pomdp

remove the end action from prism and code it into the pomdp in python. check if this is possible.

TO WRITE: the transformation to prism where we then add the extra `end` actions with the last state added.
- limit to T, which needs to be passed along
- for the end action, we only need to observation

Bibliography

[1] 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.

[2] 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.