Learning Optimal Scheduling for Time Uncertain Settings

Resume

Peter Gjøl Jensen and Jakob Haahr Taankvist

Department of Computer Science, Aalborg University,
Selma Lagerlöfs Vej 300, 9220 Aalborg Øst, Denmark

Duration Probabilistic Automata (DPAs) and Timed Priced Markov Decision Processes (PTMDPs) introduced in this work are two different classes of timed games with uncertainty, where the controller play against a stochastic environment. We here present a method which learns strategies for the controller on DPAs and PTMDPs. There are a number of different criteria one can optimize against, in this we focus on synthesizing strategies which optimizes towards minimizing the expected cost at some reachability target. The method developed is based on reinforced learning and we see that it is able to learn near-optimal strategies for Duration Probabilistic Automatons. For the larger class of PTMDPs the method is in large also applicable.

The main learning algorithm consists of a number of steps. We develop a number of different methods for two of these steps. We then experimentally explore the different methods in terms of the scheduler they synthesize, the time and the memory used for synthesizing this scheduler. We see that the methods in large are equally good, and the performance on the individual methods depends on the models of the experiments used.

We also show that the methods presented outperform previously known automated for tools synthesizing schedulers for Duration Probabilistic Automata with an order of magnitude improvement in running time, while still obtaining the same schedulers down to a difference of 0.5 in the decision boundaries. All of the methods presented have been implemented in the tool Uppaal. This enables us to also synthesize strategies which are constrained by a Uppaal-Tiga strategy, giving some worst case guarantees for the synthesized scheduler. Using the Uppaal implementation we can thus synthesize schedulers for cost-bounded reachability-objectives for games while also providing a (near-) optimal expected cost.

Learning Optimal Scheduling for Time Uncertain Settings

Peter Gjøl Jensen and Jakob Haahr Taankvist

Department of Computer Science, Aalborg University,
Selma Lagerlöfs Vej 300, 9220 Aalborg Øst, Denmark

Abstract. We present a method which learns strategies on Markov decision processes with time and price. The method will synthesize strategies, optimized towards minimizing the expected cost. The method is based on reinforced learning and is able to learn near-optimal strategies for Duration Probabilistic Automatons. The method is in large also applicable to the larger class of Priced Time Markov Decision Processes.
We develop a number of methods for different steps of the main learning algorithm, and empirically investigate their effect on the synthesized strategies. We also show that the methods presented outperform previously known automated for tools synthesizing schedulers for Duration Probabilistic Automata with an order of magnitude improvement in running time, while still obtaining the same schedulers down to a difference of 0.5 in the decision boundaries. All of the methods presented have been implemented in the tool Uppaal. This enables us to synthesize strategies for cost-bounded reachability-objectives for games while also providing a (near-) optimal expected cost.

1 Introduction

Planning any project is in it self a major task, requiring tedious allocation of resources and correct estimates of time requirements. A good project planner is able to foresee and solve races for resources between tasks while scheduling and adapting the schedule according to new information that is observed while the project is executed. This foresight is largely based on experience and a good intuition. However, even a skilled project planner can have a hard time scheduling a project in the best possible way. Factoring in the stochastic nature of the real world does not make the task any simpler. The complex interleaving of chance makes it inherently hard to adapt the schedule in an optimal way. In [16] it was shown, that an optimal scheduler under uncertainty is computable for Duration Probabilistic Automata (DPA). In [13] we showed that a near to optimal scheduler can be approximated using machine learning technics for finding a scheduler for such a DPA. We here expand and elaborate on this method of learning.

1.1 The Scheduling Problem

The problem of providing an optimal scheduler is not trivial and the question of optimality is different at different levels of abstraction. The simplest version

of the scheduling problem consists of several fixed-time tasks. Under these conditions, there are tools that can provide an optimal scheduler with respect to time [2] as well as with respect to to some cost/value using Priced Time Automata [7].

However, not all environments are controlled enough to provide fixed-time guarantees for all tasks. Consider the example in in Figure 1, a computer with a single printer has two time uncertain processes. Process *P1* opens a large file on a local media and has to print it and store changes made to it. Concurrently, process *P2* opens a fairly small file over a network connection, prints it and stores it locally. Due to parameters such as disk-access time, caches, network-delay and network congestion, the simple task of opening a file becomes time uncertain. Furthermore, to print a file, a process has to have exclusive (non pre-emptive) access to the printer.

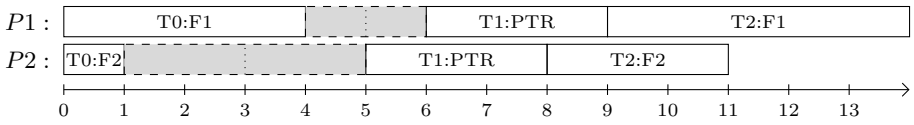

Fig. 1: The description of processes *P1* and *P2* using a shared printer (*PTR*) for printing two different files (*F1*,*F2*). The shaded areas indicate time uncertain completion of tasks *T0* in both processes, the task can complete anywhere in the shaded area. File *F1* is stored locally while *F2* is fetched via network, hence the difference in time variation.

This gives ground to Time Uncertain Schedulers, where the exact running time of tasks is uncertain. Optimal Time Uncertainty Schedulers can be found using Timed Games [11, 18], where optimal means a scheduler which minimizes (or maximizes) the worst case completion time. In contrast to fixed-time scheduling where the optimal strategy does not depend on the specific execution, the strategy of a Time Uncertain Scheduler does.

While Time Uncertain Optimal Schedulers provide a strategy guaranteed to minimize the worst case, the worst case is often not frequently occurring, making the proposed scheduler less than optimal on average. In a realistic setting, the expected completion time is of more concern than the worst case. To the best of our knowledge, there exists only one algorithm to provide an optimal scheduler with regards to expected completion time, this was provided by Kempf et al. [16], but unfortunately this algorithm does not scale well.

We present an approximating approach using reinforced learning and scheduler evaluation to approximate Time Uncertain Optimal Schedulers w.r.t expected completion time. The proposed solution is based on the work on guided scheduler optimization by Henriques et al. [15], where reinforced learning in combination with Statistical Model Checking (SMC) is used to verify Bounded

Linear Temporal Logic properties on discrete finite Markov Decision Processes (MDPs).

We also show in [13] that given this algorithm and the algorithms for worst case analysis from UPPAAL-TIGA, we can also optimize the scheduler under worst case guarantees. This was first suggested in [10]. We will introduce two models, DPAs and PTMDPs. In context of DPAs we will use the term scheduler, and in context of PTMDPs we will use the term strategy. The two terms are equivalent.

1.2 Related Work

Our synthesis problem – aiming at optimal expected cost subject to worst-case time bounds – extends the notion of *beyond* worst-case synthesis in [10] introduced for finite state MDPs, with consideration of minimizing expectation of mean-payoff (shown to be in NP∩coNP) as well reachability cost (shown to be NP-hard).

The DPA formalisms considered in [16] is a proper subclass of PTMDP proposed in this work. In [16] exact methods for synthesizing strategies with minimal expected completion time are given and implemented. However, no worst-case guarantees are considered. As we shall demonstrate our reinforcement learning method produces identical solutions and with an order of magnitude time-improvement.

[17] uses a version of Kearns algorithm to find a memoryless scheduler for expecting reward, however with no implementation provided, and no real-time consideration.

Our use of statistical model checking for learning optimal strategies of PTMDPs extends that of [15] from finite-state MDPs to the setting of timed game automata based, infinite state MDPs requiring the use of new symbolic strategies.

Finally, statistical model checking has been used for confluent MDPs in [8].

2 Duration Probabilistic Automata

We present the DPA model by first defining individual tasks, then by defining collections of tasks in a Simple Duration Probabilistic Automaton (SDPA) and finally by presenting a DPA as a collection SDPAs.

A task is a single atomic unit, which cannot be split into smaller parts, and must have exclusive access to some resource to complete the task. We define a task as

Definition 1 (Task). *A task is a four-tuple* (a, b, R, φ)*, where*

- $a, b \in \mathbb{Z}_{\geq 0}, a \leq b$ *are minimum and maximum possible durations of the task, respectively,*
- R *is a resource used by the task, and*
- φ *is a probability mass distribution function over* $[a, b]$*.*

We always assume φ to be an uniform distribution. For a task $t = (a, b, R, \varphi)$ we refer to the respective elements as $(a_t, b_t, R_t, \varphi_t)$.

A series of tasks that need to be completed in a specific sequential order is a process and can be modeled as an SDPA. The SDPA contains information about the progress of tasks and contains states to wait in between execution of two tasks. We define an SDPA formally as:

Definition 2 (Simple Duration Probabilistic Automaton). *A Simple Duration Probabilistic Automaton (SDPA) is a tuple $(T, next, t_1, x)$, where*

- *T is a set of tasks,*
- *$t_1 \in T$ is the initial task, and*
- *$next : T \to (T \cup \{\bot\}) \setminus \{t_1\}$ is a transition function from any task to the successor task or to the final state $\bot$ if there are no remaining tasks. next is a one-to-one mapping.*
- *$\overline{t}$ is the complement-task for each $t \in T$.*
- *x is a clock.*

A state in an SDPA $S = (T, next, t_1, x)$ is a pair $(\tilde{t}, r)$ where $\tilde{t} \in \{t, \overline{t} \mid t \in T\} \cup \{\bot\}$ is a discrete state and $r \in \mathbb{R}_{\geq 0}$ is the valuation of the clock x. The initial state in S is the pair $(\overline{t_1}, 0)$ and the state space $SP(S)$ of S is the set of all states.

The complemented task named $\overline{t}$ denotes a state *waiting to start* t. We define for a task t that $a_{\overline{t}} = 0$, $b_{\overline{t}} = \infty$, $R_{\overline{t}} = \emptyset$. The complement task $\overline{t}$ is a pseudo task allowing the SDPA to postpone executing a task for some arbitrary amount of time.

We can then define the semantics of an SDPA. These are defined as a transition system, using the following rules:

$$\frac{}{(\tilde{t}, r) \xrightarrow{d}_s (\tilde{t}, r + d)} r + d \leq b_{\tilde{t}}, d \in \mathbb{R}_{\geq 0}, \tilde{t} \in \{t, \overline{t} \mid t \in T\} \cup \{\bot\} \tag{1}$$

$$\frac{}{(t, r) \xrightarrow{done}_s (\overline{s}, 0)} t \in T, s \in T \cup \{\bot\}, next(t) = s, a_t \leq \nu \leq b_t \tag{2}$$

$$\frac{}{(\overline{t}, r) \xrightarrow{go}_s (t, 0)} t \in T \tag{3}$$

A DPA is a set of SDPAs where tasks are executed in parallel and share a common resource pool. We formally define a DPA:

Definition 3 (Duration Probabilistic Automaton). *A Duration Probabilistic Automaton (DPA) D is a set $\mathcal{S} = \{S_1, S_2, \ldots, S_N\}$ of N SDPAs where,*

- *the state space of the DPA D is $SP(D) = SP(S_1) \times SP(S_2) \times \cdots \times SP(S_N)$ excluding states with conflicting resources, i.e. excluding states $\{(\tilde{t^1}, x^1), \ldots, (\tilde{t^N}, x^N)\}$ where $\exists 1 \leq i, j \leq N, i \neq j . R_{\tilde{t^i}} \cap R_{\tilde{t^j}} \neq \emptyset$.*

– *the initial state of the DPA D is* $\{(\overline{t_1^1}, 0), (\overline{t_1^2}, 0), \ldots, (\overline{t_1^N}, 0)\}$ *where* t_1^i *is the initial task in* S_i.
– *the set of clocks for a DPA D is defined as* $X = \{x^1, \ldots, x^N\}$.
– *a clock valuation* ν *is a function* $\nu : X \to \mathbb{R}_{\geq 0}$.

Before we define the semantics of a DPA we define the plus operator and the less-than comparator on clock valuations, we sometimes denote a clock valuation as a set of pairs of clocks and real numbers s.t. $\nu = \{(x, 4), (y, 1), (z, 5)\}$, if $\nu(x) = 4, \nu(y) = 1$ and $\nu(z) = 5$:

Definition 4 (Plus Operator for Clock Valuation and Number). *The plus operator for a clock valuation* ν *and a number* $d \in \mathbb{R}$*:*

$$\nu + d = \{(x, r + d) \mid (x, r) \in \nu\}.$$

we then define the less-than comparator of two clock valuations:

Definition 5 (Less-Than for Clock Valuations). *Given clock valuations* ν_1 *and* ν_2 *defined over the same set of clocks,* $\nu_1 < \nu_2$ *if:*

$$\forall (x, r) \in \nu_1 . \exists (x, r') \in \nu_2 . r < r'.$$

The semantics of a DPA are defined as a transition system as well, using the following rules:

$$\frac{(\tilde{t^i}, r^i) \xrightarrow{d}_s (\tilde{t^i}, r^i + d) \text{ for all } 1 \leq i \leq N}{\{(\tilde{t^1}, r^1), \ldots, (\tilde{t^N}, r^N)\} \xrightarrow{d} \{(\tilde{t^1}, r^1 + d), \ldots, (\tilde{t^N}, r^N + d)\}} \tag{4}$$

$$\frac{(t^j, r^j) \xrightarrow{done}_s (\overline{s}, 0)}{\{(\tilde{t^1}, r^1), \ldots, (t^j, r^j), \ldots, (\tilde{t^N}, r^N)\} \xrightarrow{done^j} \{(\tilde{t^1}, r^1), \ldots, (\overline{s}, 0), \ldots, (\tilde{t^N}, r^N)\}} \tag{5}$$

$$\frac{(\overline{t^j}, r) \xrightarrow{go}_s (t^j, 0), \quad \bigcup_{1 \leq i \leq N, i \neq j} \tilde{t_R^i} \cap t_R^j = \emptyset}{\{(\tilde{t^1}, r^1), \ldots, (\overline{t^j}, r^j), \ldots, (\tilde{t^N}, r^N)\} \xrightarrow{go^j} \{(\tilde{t^1}, r^1), \ldots, (t^j, 0), \ldots, (\tilde{t^N}, x^N)\}} \tag{6}$$

To control the behavior of the DPA, we create a scheduling policy Ω which decides what action to take in a given state of the DPA. A DPA under a specific scheduler results in a Markov Chain.

Definition 6 (Scheduling Policy). *Given a DPA D with the set of SDPAs* $\mathcal{S} = \{S_1, S_2, \ldots, S_N\}$*, a scheduling policy is a function* $\Omega : SP(D) \to \{go^i \mid 1 \leq i \leq N\} \cup \{\mathbf{w}\}$ *such that for every* $i \in [1, N], \Omega(q, \nu) = go^i$ *only if* S_i *has a waiting task in* q *and* $\Omega(q, \nu) = \mathbf{w}$ *only if at least one task is running in* q.

The intuition of the scheduler is that either Ω returns go^i and a task is started or Ω returns $\mathbf{w}$ for some time until Ω again returns go^j. With the model defined, we can now define the problem.

3 Optimal Scheduling of DPAs

In this section we start by presenting the notion of an optimal scheduler and continue by elaborating the results of Kempf et al. [16] with regards to different types of schedulers. The definition used in this section are inspired by the definitions by Kempf et al. [16].

3.1 Optimal Scheduling

While Definition 1 of a task states φ to be a uniform distribution over an interval, the interleaving of tasks result in a shift in the probability distribution. To handle this, we define a shift similar to Kempf et al. [16] such that we normalize the probability distribution:

Definition 7 (Shift). *A shift x of the probability distribution φ for a task t, written $\varphi_{/x}$ is defined as*

$$\varphi_{/x}[\tau] = \varphi[\tau + x] \qquad if\ 0 \leq x \leq a_t \tag{7}$$

$$\varphi_{/x}[\tau] = \varphi[\tau + x] \cdot \frac{b_t - a_t}{b_t - x} \qquad if\ a_t < x < b_t \tag{8}$$

With shift defined, we can begin to reason about the probability that some task finishes before others in a DPA. As with shift, Kempf [16] et al. provided a similar definition. Based on this, we define the probability ρ for an SDPA s with a running task, that s is the first to terminate its current task.

Definition 8 (Task Termination Probability). *Given a DPA D, a state $(q, \nu) \in SP(D)$ and SPDAs with running tasks $\{s_1, .., s^i, .., s_n\}$, the probability that s^i terminates at time τ and all other running tasks terminate at $> \tau$ is given by*

$$\rho^i(q, \nu)[\tau] = \varphi^i_{/x^i}[\tau] \prod_{i' \neq i} \varphi^{i'}_{/x^{i'}}[> \tau] \tag{9}$$

This leads us to the definition of stochastic time-to-go for a DPA under a scheduler. From Definition 8 we know the probability that some task will terminate, and given our scheduler Ω we know which task will be started in the next state. From this, we can construct a formula for the probability distribution for a given DPA D under Ω terminates.

Definition 9 (Stochastic Time-To-Go). *Given a DPA D and a scheduler Ω, the global probability distribution for a global state (q, ν) is given by function μ defined as*

$$if\ \forall p \in q, p = \bot \qquad \mu(q,\nu,\Omega)[\tau] = \begin{cases} 1 & if\ \tau = 0 \\ 0 & if\ \tau > 0 \end{cases} \tag{10}$$

$$if\ (q,\nu+\tau) \overset{\Omega(q,\nu+\tau)}{\longrightarrow} (q',\nu')$$

$$\mu(q,\nu,\Omega)[\tau] = \sum_{i \in Active} \int_0^\tau \rho^i(q,\nu)[\tau'] \cdot \mu(q',\nu',\Omega)[\tau-\tau']d\tau' \tag{11}$$

From Definition 9 we can derive the expected time-to-go under a scheduler

Definition 10 (Expected Time-To-Go).

$$E(q,\nu,\Omega) = \int_0^\infty \tau \cdot \mu(q,\nu,\Omega)[\tau]d\tau \tag{12}$$

Using Definition 10, we can create a notion of the optimal scheduler Ω, as this is the scheduler that given any clock-valuation ν and state q is as least as good as any other possible scheduler Ω'. We define the optimal scheduler formally

Definition 11 (The Optimal Scheduler). *The stochastic optimal scheduler Ω is defined as*

$$\forall \Omega'.E(q,\nu,\Omega) \leq E(q,\nu,\Omega') \tag{13}$$

3.2 Scheduler Types

In a probabilistic setting, it is important to distinguish between a static and an adaptive scheduling policy [1]. A static scheduling policy is not influenced by the clock-valuations i.e. a scheduling policy is based solely on the discrete state of the system. As it cannot adapt to the actual task durations, this scheduler type cannot always describe an optimal scheduler. This can be seen in the example in Figure 2 which contains two determined runs over the DPA from the example in Figure 1. An optimal scheduler for this DPA is for the second task in *P1* to use the printer first, *unless* the first task in *P2* completes in 1 time units, in which case *P2* should use the printer first. But because the actual completion times of tasks cannot be considered in a static scheduling policy, a static scheduling policy cannot describe an optimal scheduling of the DPA.

Another important distinction is between lazy and non-lazy schedulers. It was shown by Abdedda et al. [19] that the class of non-lazy schedulers contains the optimal scheduling. We use the definitions of laziness by Kempf et al. [16].

Definition 12 (Laziness). *A scheduling policy Ω is lazy at (q,ν) if $\Omega(q,\nu+\tau) = i$ and $\Omega(q,\nu+\tau') = w$ for every $\tau' \in [0,\tau)$. A schedule is non-lazy if no such (q,ν) exists.*

They continue by showing that

Theorem 1 (Non-Lazy Optimal Schedulers). *The optimal value E can be obtained by a non-lazy scheduler.*

This intuitively means that if it pays off for a task to wait, it will do so until some task releases some resources (i.e. a task finishes).

Due to the non-blocking nature of tasks with no resource requirements, we can introduce another theorem which reduces the complexity of the generated scheduler. Theorem 2 states that if there is a task t without any resource requirements waiting to be started, it is always optimal to start t.

Theorem 2. *The optimal value of E can be obtained by a scheduler which always immediately starts waiting tasks which do not need any resources.*

Proof (Theorem 2). Consider an optimal scheduler Ω which chooses to wait in the state (q, ν) that has some waiting task t from the SDPA S which does not need any resources. We can construct another optimal scheduler Ω' which starts t in the state (q, ν) and in any other state (q', ν'), $\Omega'(q', \nu') = \Omega(q', \nu')$.

Clearly, as Ω is optimal from every state in the state space and with every clock valuation ν, Ω' is optimal for all other states than (q, ν). This is due to the fact that Ω and Ω' only depends on the current state of the DPA (they are memoryless).

Starting t in (q, ν) cannot effect any other SDPA as it uses no resources. The SDPA containing t can only meet an earlier deadline than under Ω, as t will now be executed sooner. This means that the expected running time under Ω' can only be shorter than or equal to the running time under Ω. And as Ω is optimal, so is Ω'. □

Theorem 3. *Given an optimal scheduler Ω and two states (q, ν) and (q, ν') s.t. $\nu = \{(x_1, r_1), \ldots, (x_n, r_n)\}$ and $\nu' = \{(x_1, r_1 + d_1), \ldots, (x_n, r_n + d_n)\}$ where $d_1, \ldots, d_n \in \mathbb{R}_{\geq 0}$ It holds that $E(q, \nu, \Omega) \geq E(q, \nu', \Omega)$.*

In the states the discrete states are the same, and all tasks in (q, ν') are at least as progressed as in (q, ν). It then holds that the expected time to go for (q, ν) is at least as big as for (q, ν') under an optimal scheduler.

From Kempf et al. [16] we have the following lemma:

Lemma 1. *Let q be a state and let ν and ν' be two clock valuations which are identical except for some clock $x \in X$, $\nu'(x) = \nu(x) + d$ and let Ω be an optimal scheduler. Then $E(q, \nu', \Omega)$ is at least as good as $E(q, \nu, \Omega)$, that is, $E(q, \nu', \Omega) \leq E(q, \nu, \Omega)$.*

Proof (Theorem 3). Given an optimal scheduler Ω and a state (q, ν) s.t. $\nu = \{(x_1, r_1), \ldots, (x_n, r_n)\}$ and a state (q, ν') s.t. $\nu' = \{(x_1, r_1 + d_1), \ldots, (x_n, r_n + d_n)\}$ we now prove that $E(q, \nu', \Omega) \leq E(q, \nu, \Omega)$

The following holds due to Lemma 1:

$$E(q, \nu, \Omega) \geq E(q, \nu_1, \Omega) \geq E(q, \nu_2, \Omega) \geq \cdots \geq E(q, \nu_n, \Omega)$$

Where

$$
\begin{aligned}
\nu_1 &= \{(x_1, r_1 + d_1), (x_2, r_2), \ldots, (x_n, r_n)\}, \\
\nu_2 &= \{(x_1, r_1 + d_1), (x_2, r_2 + d_2), \ldots, (x_n, r_n)\}, \\
&\vdots \\
\nu_n &= \{(x_1, r_1 + d_1), (x_2, r_2 + d_2), \ldots, (x_n, r_n + d_n)\}
\end{aligned}
$$

And as $\nu_n = \nu'$ Theorem 3 is true.

□

Finally, schedulers can be preemptive or non-preemptive. In preemptive schedulers, tasks may be preempted after being started. This is often not applicable to real processes which may be non-preemptable e.g. due to side effects. For this reason we will focus only on non-preemptive schedulers.

As the aim of this thesis is to approximate optimal scheduling, we will in consider only non-preemptive non-lazy adaptive scheduling policies and refer to such simply as *scheduling policies* for DPAs.

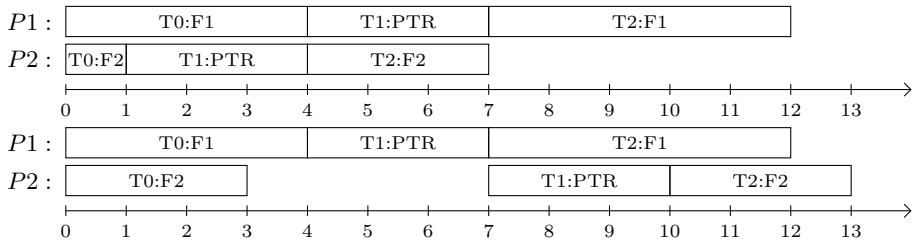

Fig. 2: Example of two timeliness showing determined runs for the processes shown in Figure 1. No static scheduler can be optimal in both runs.

4 Priced Timed Markov Decision Processes

In [13], attached and currently under review, we describe methods for solving the scheduling problem. In the paper we introduce the more general model of Priced Timed Markov Decision Processes (PTMDPs).

Example Consider the PTMDP of Figure 3 modeling a process consisting of a sequence of two uncontrollable steps, `r, d`, with a possible control action, `a, b, w` being taken after the first step. The first step `r` is taken between 0 and 100 time-units according to a uniform distribution[1] as can be seen by the invariant

[1] following the stochastic semantics for timed automata components applied in UPPAAL SMC.

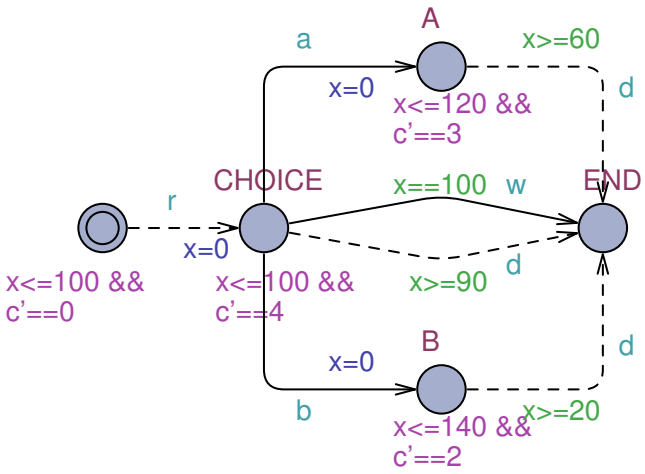

Fig. 3: Example of an PTMDP from [13].

`x<=100` and the absent guard, and with cost-rate `c'==0`. In the next step, the controller may suggest to play any of the time-action pairs $(d, \texttt{a})$, $(d, \texttt{b})$ with $d \leq 100$ or $(100, \texttt{w})$. These will be in competition with the uniformly distributed choices of the environment $(e, \texttt{d})$ with $e \in [90, 100]$. It is clear that in terms of worst-case time, the best choice for the controller is $(100, \texttt{w})$ with 200 as worst-case overall time. In contrast, the worst choice for the controller is $(100, \texttt{b})$ with 340 as worst-case time.

Another case is that we want to minimize the expected value of the clock c. This means that it would be stupid to wait for some time in `CHOICE` and then do a or b. This is due to the fact that x is reset when taking a or b, which means that no matter when we take the a or b transitions we will stay in the `A` or `B` location the same amount of time.

Lets now look into the different choices:

Choose to do a As argued if the controller chooses to do a it should *not* wait in the `CHOICE` location. This means that the c clock will only progress in the `A` location. In the `A` location only the environment can act. When $x \geq 60$ the environment can go to the `END` location, and there is an invariant such that it always holds that $x \leq 120$. Thus the environment must go to `END` when $60 \leq x \leq 120$. We assume that the environment chooses the delay over a uniform distribution. As $c' = 3$ this means that the expected value of c in `END` is $3 \cdot \left(\frac{120-60}{2} + 60\right) = 270$.

Choose to do b This choice is very similar to choosing a however here the environment must go to `END` when $20 \leq x \leq 140$ and $c' = 2$. Thus in this case the expected value of c in `END` is $2 \cdot \left(\frac{140-20}{2} + 20\right) = 160$.

Choose to wait 100 and then do w In this case we choose to wait in the `CHOICE` location the environment has to go to `END` when $90 \leq x \leq 100$ and

$c' = 4$. The controller can only choose w when $x = 100$, in theory there is a race between the controller and the environment. However the only time the controller can act is when $x = 100$, and then the controller will go to `END`. In the same situation the only choice for the environment is also to go to `END` as well. Thus the expected value for c in `END` can be calculated in the same way as in the two previous cases, and is $4 \cdot \left(\frac{100-90}{2} + 90\right) = 380$.

This means that the best strategy for minimizing the expected cost, i.e. the final value of `c`, is to choose `b`.

Now assume that the `END` location must be reached within an upper time-bound of 210. The on-the-fly method of UPPAAL-TIGA (exploiting early termination) may (in fact will) produce the strategy which deterministically chooses $(100, \mathtt{w})$. This clearly meets the given upper time-bound, and yields an expected reachability cost of $4 * 95 = 380$. The most permissive strategy guaranteeing the time-bound `210` (also obtainable by UPPAAL-TIGA) will have the choice depend on the time-point t when `CHOICE` is reached: if $t > 90$ only $(100, \mathtt{w})$ is a legal choice; if $70 < t \leq 90$ also $(d, \mathtt{a})$ with $d \leq 90 - t$ are legal choices, and finally if $t \leq 70$ also $(e, \mathtt{b})$ with $e \leq 70 - t$ are legal. The strategy with minimal expected reachability cost while guaranteeing the time-bound 210, will (obviously) deterministically make the "cheapest" legal choice for a given value of t, i.e.

- $(100, \mathtt{w})$ for $t > 90$,
- $(0, \mathtt{a})$ when $70 < t \leq 90$,
- and $(0, \mathtt{b})$ when $t \leq 70$.

This yields 204 as minimum expected value; We know that the value of t will be uniformly distributed. This means that 70% of the time we will do $(0, \mathtt{b})$, 20% of the time we will do $(0, \mathtt{a})$ and 10% of the time we will do $(100, \mathtt{w})$. We know the expected price of the different choices from above. Thus we can simply take the sum of the products of the price and the probability of an action. This will give us the expected price:

$$0.7 \cdot 160 + 0.2 \cdot 270 + 0.1 \cdot 380 = 204.$$

Thus we now have the most cost optimal strategy under the constraint that we have to reach the `END` location in 210 time units.

4.1 Priced Timed Markov Decision Processes

In this section we formalize the notion of PTMDPs. This section is analogous to Sections 2 and 3 but for the more general model of PTMDPs.

We first introduce Priced timed games and then use this in the definition of PTMDPs.

Priced Timed Games Priced Timed Games [20] are two-player games played on (priced) timed automata [3, 6]. Here we recall the basic results. Let $X = \{x, y, ...\}$ be a finite set of clock. We define $\mathcal{B}(X)$ as the set of clock constraints over X generated by grammar: $g, g_1, g_2 ::= x \bowtie n \mid x - y \bowtie n \mid g_1 \wedge g_2$, where $x, y \in X$ are clocks, $n \in \mathbb{N}$ and $\bowtie \in \{\leq, <, =, >, \geq\}$.

Definition 13. *A* Priced Timed Automaton *(PTA)* $\mathcal{A} = (L, \ell_0, X, \Sigma, E, R, Inv)$ *is a tuple where* L *is a finite set of locations,* $\ell_0 \in L$ *is the initial location,* X *is a finite set of non-negative real-valued clocks,* Σ *is a finite set of actions,* $E \subseteq L \times \mathcal{B}(X) \times \Sigma \times 2^X \times L$ *is a finite set of edges,* $P : L \to \mathbb{N}^X$ *assigns a price-rate to each location, and* $Inv : L \to \mathcal{B}(X)$ *sets an invariant for each location.*

The semantics of a PTA $\mathcal{A}$ is a *priced transition system* $S_{\mathcal{A}} = (Q, q_0, \Sigma, \to)$, where the set of states Q consists of pairs (ℓ, v) with $\ell \in L$ and $v \in \mathbb{R}^X_{\geq 0}$ such that $v \models Inv(\ell)\}$, and $q_0 = (\ell_0, 0)$ is the initial state. Σ is a finite set of actions, and $\to \subseteq Q \times (\Sigma \cup \mathbb{R}_{\geq 0}) \times \mathbb{R}_{\geq 0} \times Q$ is the priced transition relation defined separately for action $a \in \Sigma$ and delay $d \in \mathbb{R}_{\geq 0}$ as:

- $(\ell, v) \xrightarrow{a}_0 (\ell', v')$ if there is an edge $(\ell \xrightarrow{g,\alpha,r} \ell') \in E$ such that $v \models g$, $v' = v[r \mapsto 0]$ and $v' \models Inv(\ell')$,
- $(\ell, v) \xrightarrow{d}_p (\ell, v + d)$, where $p = P(\ell) \cdot d$, $v \models Inv(\ell)$ and $v + d \models Inv(\ell)$.

Thus, the price of an action-transition is 0, whereas the price of a delay transition is proportional to the delay according to the price-rate of the given location. We shall assume that $S_{\mathcal{A}}$ is *deterministic* in the sense that any state $q \in Q$ has at most one successor q^α for any action or delay $\alpha \in (\Sigma \cup \mathbb{R}_{\geq 0})$. A *run* of a PTA $\mathcal{A}$ is an alternating sequence of priced action and delay transitions of its priced transition system $S_{\mathcal{A}}$: $\pi = q_0 \xrightarrow{d_0}_{p_0} q'_0 \xrightarrow{a_0}_0 q_1 \xrightarrow{d_1}_{p_1} q'_1 \xrightarrow{a_1}_0 \cdots \xrightarrow{d_{n-1}}_{p_{n-1}} q'_{n-1} \xrightarrow{a_{n-1}}_0 q_n \cdots$, where $a_i \in \Sigma$, $d_i, p_i \in \mathbb{R}_{\geq 0}$, and q_i is a state (ℓ_{q_i}, v_{q_i}). We denote the set of runs of $\mathcal{A}$ as $Exec_{\mathcal{A}}$, and $Exec^f_{\mathcal{A}}$ ($Exec^m_{\mathcal{A}}$) for the set of its finite (maximal) runs. For a run π we denote by $\pi[i]$ the state q_i, and by $\pi|_i$ ($\pi|^i$) the prefix (suffix) of π ending (starting) at q_i. For a finite run π, $C(\pi)$ denotes its total accumulated cost $\sum_i^n p_i$. Similarly $T(\pi)$ denotes the total accumulated time $\sum_i^n d_i$. An infinite run π is said to be cost-divergent provided $\lim_{n\to\infty} \sum_i^n p_i = +\infty$. We say that $\mathcal{A}$ is (cost-) non-Zeno provided every infinite run is time-(cost-)divergent.

Definition 14. *A* Priced Timed Game $\mathcal{G}$ *(PTG) is a PTA whose actions* Σ *are partitioned into controllable (*Σ_c*) and uncontrollable (*Σ_u*) actions.*

We note, that for PTAs and PTGs with $P(\ell) = 1$ in all locations ℓ, we obtain standard timed automata (TA) and timed games (TG). Given a (P)TG $\mathcal{G}$, a set of goal-locations $G \subseteq L$ and a cost- (time-) bound $B \in \mathbb{R}_{\geq 0}$, the (G, B) *cost-(time-) bounded reachability control problem* for $\mathcal{G}$ consists in finding a *strategy* σ that will enforce G to be reached within accumulated cost (time) B. The formal definition of this control problem is based on definitions of *strategy* and *outcome*.

Definition 15. *A* strategy σ *over a PTG* $\mathcal{G}$ *is a partial function from* $Exec^f_{\mathcal{G}}$ *to* $\mathcal{P}(\Sigma_c \cup \{\lambda\}) \setminus \{\emptyset\}$ *such that for any finite run* π *ending in state* $q = last(\pi)$*, if* $a \in \sigma(\pi) \cap \Sigma_c$*, then there must exist a transition* $q \xrightarrow{a} q'$ *in* $S_{\mathcal{G}}$*.*

Given a PTG $\mathcal{G}$ and a strategy σ over $\mathcal{G}$, the outcome $Out(\sigma)$ is the subset of $Exec_{\mathcal{G}}$ defined inductively by $q_0 \in Out(\sigma)$, and:

- If $\pi \in Out(\sigma)$ then $\pi' = \pi \xrightarrow{e} q' \in Out(\sigma)$ if $\pi' = Excel_{\mathcal{G}}$ and either one of the following three conditions hold:
  1. $e \in \Sigma_u$, or
  2. $e \in \Sigma_c \cap \sigma(\pi)$ and $e \in \sigma(\pi)$, or
  3. $e \in \mathbb{R}_{\geq 0}$ and for all $e' < e$, $last(\pi) \xrightarrow{e'} q'$ for some q' st $\sigma(\pi \xrightarrow{e'} q') \ni \lambda$.

Let (G, B) be a cost- (time-) bounded reachability objective for $\mathcal{G}$. We say that a maximal, finite run π is *winning* w.r.t. (G, B), if $last(\pi) \in G \times \mathbb{R}_{\geq 0}$ and $C(\pi) \leq B$. A strategy σ over $\mathcal{G}$ is a *winning strategy* if all runs in $Out(\sigma)$ are winning (w.r.t. (G, B)).

A *memoryless strategy* σ only depends on the last state of a run, e.g. whenever $last(\pi) = last(\pi')$, then $\sigma(\pi) = \sigma(\pi')$. For unbounded reachability and safety objectives for TGs, memoryless strategies suffices [20], For TGs with an additional clock `time`, which is never reset (here named *clocked* TGs), memoryless strategies even suffices for time-bounded reachability objectives.

The notion of strategy in Def. 15 is non-deterministic, thus inducing a natural order of *permissiveness*: $\sigma \preceq \sigma'$ iff $\sigma(\pi) \subseteq \sigma'(\pi)$ for any finite run π. *Deterministic* strategies – returning singleton-sets for each run – are least permissive. For safety objectives – being maximal fixed-points – strategies are closed under point-wise union, yielding (unique) *most permissive strategies*. For TGs being non-Zeno, time-bounded reachability objectives *are* safety properties.

Theorem 4. *Let* $\mathcal{G}$ *be a non-Zeno, clocked TG. If a time-bounded reachability objective* (G, T) *has a winning strategy, then it has (a) deterministic, memoryless winning strategies, and (b) a (unique) most permissive, memoryless winning strategy* $\sigma^p_{\mathcal{G}}(G, T)$*.*

The tool UPPAAL-TIGA [5] provides on-the-fly, symbolic (zone-based) algorithms for computing both types of memoryless safety strategies for TGs. For PTGs, the synthesis problem for cost-bounded reachability problems is in general undecidable [9].

Priced Timed Markov Decision Processes The definition of outcome of a strategy in the previous Section assumes that an environment behaves completely antagonistically. We will now assume a randomized environment, where the choices of delay and uncontrollable actions are stochastic according to a (delay,action)-density function for a given state.

Definition 16. *A* Priced Timed Markov Decision Process *(PTMDP) is a pair* $\mathcal{M} = \langle \mathcal{G}, \mu^u \rangle$*, where* $\mathcal{G} = (L, \ell_0, X, \Sigma_c, \Sigma_u, E, R, Inv)$ *is a PTG, and* μ^u *is a*

family of density-functions, $\{\mu_q^u : \exists \ell \exists v.q = (\ell, v)\}$*, with* $\mu_q^u(d,u) \in \mathbb{R}_{\geq 0}$ *assigning the* density *of the environment aiming at taking the uncontrollable action* $u \in \Sigma_u$ *after a delay of* d *from state* q*.*

In the above definition, it is tacitly assumed that $\mu_q^u(d,u) > 0$ only if $q \xrightarrow{d,u}$ in $\mathcal{G}$. Also, we shall *wlog* for time-bounded reachability objectives assume that $\sum_u(\int_{t\geq 0} \mu_q^u(t,u)dt) = 1$ [2]. In case the environment wants to perform an action deterministically after an exact delay d, μ_q^u will involve the use of Dirac delta function (see [12]).

The presence of the stochastic component μ^u makes a PTMDP a de facto infinite state Markov decision process. Here we seek strategies that will minimize the expected accumulated cost of reaching a given goal set G.

Definition 17. *A* stochastic strategy μ^c *for a PTMDP* $\mathcal{M} = \langle \mathcal{G}, \mu^u \rangle$ *is a family of density-functions,* $\{\mu_q^c : \exists \ell \exists v.q = (\ell, v)\}$*, with* $\mu_q^c(d,c) \in \mathbb{R}_{\geq 0}$ *assigning the* density *of the controller aiming at taking the controllable action* $c \in \Sigma_c$ *after a delay of* d *from state* q*.*

Again it is tacitly assumed that $\mu_q^c(d,c) > 0$ only if $q \xrightarrow{d,c}$ in $\mathcal{G}$. Now, a PTMDP $\mathcal{M} = \langle \mathcal{G}, \mu^u \rangle$ and a stochastic strategy μ^c defines a race between the environment and the control strategy, where the outcome is settled by the two competing density-functions. More precisely, the combination of $\mathcal{M}$ and μ^c defines a probability measure $\mathbb{P}_{\mathcal{M},\mu^c}$ on (certain) sets of runs.

For $\ell_i \in L$ and $I_i = [l_i, u_i]$ with $l_i, u_i \in \mathbb{Q}$, $i = 0..n$, we denote the *cylinder set* by $\mathcal{C}(q, I_0\ell_0 I_1 \cdots I_{n-1}\ell_n)$ consisting of all maximal runs having a prefix of the form: $q \xrightarrow{d_0} \xrightarrow{a_0} (\ell_1, v_1) \xrightarrow{d_1} \xrightarrow{a_1} \cdots \xrightarrow{d_{n-1}} \xrightarrow{a_{n-1}} (\ell_n, v_n)$ where $d_i \in I_i$ for all $i < n$. Providing the basis for a Sigma-algebra, we now inductively define the probability measure for such sets of runs[3]:

$$\mathbb{P}_{\langle \mathcal{G},\mu^u\rangle,\mu^c}\big(\mathcal{C}(q, I_0\ell_0 I_1 \cdots I_{n-1}\ell_n)\big) =$$
$$\sum_{p\in\{u,c\}} \sum_{\substack{a\in\Sigma_p \\ \ell_q \xrightarrow{a} \ell_1}} \int_{t\in I_0} \mu_q^p(t,a) \cdot \Big(\int_{\tau > t} \mu_q^{\overline{p}}(\tau)d\tau \Big) \cdot \mathbb{P}_{\langle \mathcal{G},\mu^u\rangle,\mu^c}\big(\mathcal{C}((q^t)^a, \mathcal{C}(I_1 \cdots I_{n-1}\ell_n)\big)\, dt$$

The above definition requires a few words of explanation: the outermost sums divide into cases according to who wins the race of the first action (c or u), and which action a the winner will perform. Next, we integrate over all the legal delays the winner may choose according to the given interval I_0 using the relevant density-function. Independently, the non-winning player ($\overline{p}$) must choose a larger delay; hence the product of the probability that this will happen. Finally, the probability of runs according to the remaining cylinder $I_1\ell_1, \cdots, I_{n-1}\ell_n$ from the new state $(q^t)^a$ is taken into account.

[2] For a time-bounded reachability objective (G, T), we may without affecting controllability assume that each location has each action (controllable or uncontrollable) action enabled after T.

[3] with the base case, e.g. $n = 0$, being 1

Now let $\pi \in Exec^m$ and let G be as set of goal locations. Then $C_G(\pi) = min\{C(\pi|_i) : \pi[i] \in G\}$ denotes the accumulated cost before π reaches G[4]. Now C_G is a random variable, which for a given stochastic strategy, μ^c, will have expected value given by the following Lesbegue integral:

$$\mathbb{E}^{\mathcal{M}}_{\mu^c}(C_G) = \int_{\pi \in Exec^m} C_G(\pi) \mathbb{P}_{\mathcal{M},\mu^c}(d\pi)$$

Now, we want a (near-optimal) stochastic strategy μ^o that minimizes this expected value, subject to guaranteeing T as a worst-case reachability time-bound – or alternatively – subject to μ^o being a stochastic refinement ($\prec$ [5]) of the most permissive time-bounded reachability strategy $\sigma^p(G,T)$ for $\mathcal{M}$. That is $\mathbb{E}^{\mathcal{M}}_T(C_G) = \inf\left\{ \mathbb{E}^{\mathcal{M}}_{\mu^c}(C_G) \mid \mu^c \prec \sigma^p(G,T) \right\}$. We note that letting μ^c range over deterministic strategies σ^d suffices in attaining $\mathbb{E}^{\mathcal{M}}_T(C_G)$.

5 Learning Strategies for PTMDPs

The algorithm used to learn will generate an approximation of the optimal strategy. The algorithm has five main phases, and one optional. All these can be seen in Figure 4. We will now shortly describe each of these steps. In the following sections, Section 9 and Section 10, we will take a more in depth look at the *filtering* and *learning* phases.

Simulation In the initial step of the algorithm, we use UPPAAL SMC to make a batch of runs which we can then learn on. In the first round the runs will be generated over a uniform strategy. In the subsequent iterations we use the stochastic strategy from the learning step of the previous iteration. The result of this phase is a set of runs Π. Given that we have learned a most permissive strategy using UPPAAL-TIGA, the simulation is restricted to always respect this strategy.

Filtering When filtering, we choose the *best* runs. We will look into different definitions of *best* runs in Section 9. The set of *best* runs are then send to the Learning phase.

Learning In this step we use the set of *best* runs for learning which actions led to these runs. Thus the intuition is that if all the *best* runs did action a in location l, it is likely to be a good action to do. The result of this step is a stochastic strategy, which is a density function μ^c_q that for a given state q, action a and delay d gives a probability to take a after d in q.

Currently we have three different methods for doing the learning; Co-Variance matrices, Logistic Regression and Splitting. We elaborate these methods in Section 10.

[4] Note that $C_G(\pi)$ will be infinite in case π does not reach G. However, this case will never happen in our usages.

[5] $\mu^c \prec \sigma$ iff $\mu^c_q(d,a) > 0$ only if $\lambda \in \sigma(q^e)$ for all $e < d$ and $a \in \sigma(q^d)$.

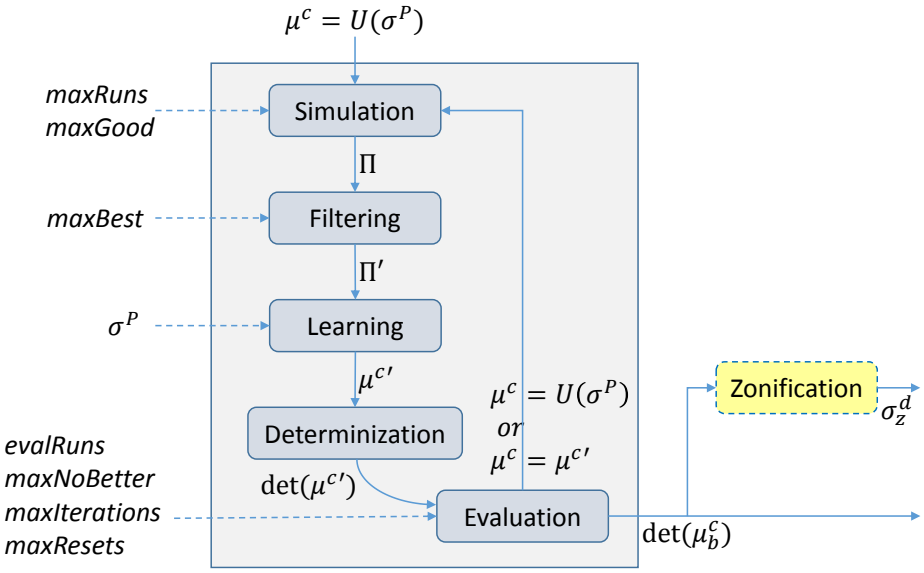

Fig. 4: Approximation of the optimal strategy using reinforced learning from [13]. In the article everything on the figure is explained, in this work we will only explain the most essential parts.

Determinization As we know that the optimal strategy is deterministic, we here determinize the strategy. This simply means that we always choose the action, delay pair which have the highest probability. Note that in the next iteration we do *not* learn from runs generated under the determined strategy but under the stochastic strategy.

Evaluation In this step we evaluate the generated deterministic strategy. In each iteration of the algorithm we save the best strategy seen so far. The strategies is evaluated using UPPAAL SMC. When we have not synthesized a better strategy than the one we have for *maxNoBetter* iterations, we restart the algorithm with the uniform strategy (but remembering the current best strategy). After *maxResets* resets, we terminate the algorithm, and report the best candidate found across the different resets. There are also other termination criteria build into this step, which we will not elaborate on here.

Zoneification (optional) In this step we translate the strategy such that it is expressed in terms of zones. This step makes it possible to e.g. do model checking on the model under the learned strategy. Currently this step is not implemented. The splitting method generates strategies which are essentially expressed as zones. This means that it is theoretically trivial to do model checking on a model under a strategy generated by the splitting method. However in practice this is currently not implemented in UPPAAL.

Model checking under a strategy could e.g. give us the worst possible time to completion under the synthesized strategy.

6 The Learning Algorithm in Practice

In [13] we conducted a set of experiments. We saw in these experiments that the three methods mentioned in Section 5 preformed equally well on the experiments chosen. We also saw that the algorithm is able to learn strategies which are significantly better than the uniform strategy. Both when unconstrained and when constrained by a strategy generated by UPPAAL-TIGA.

We will now take a more in debt look at how the algorithm works when run on the PTMDP from Figure 3 constrained by the UPPAAL-TIGA strategy which guarantees that we reach `END` in 210 time units.

We here focus on the Logistic Regression learning method. In Figure 5 we have plotted runs going through the `CHOICE` location, under strategies generated in different iterations.

Figure 5.a We here see the runs generated under the uniform strategy, when constrained by the UPPAAL-TIGA strategy. We can see that the algorithm never chooses to do $(0, \mathtt{b})$ when $t > 70$ which is exactly what is specified by the UPPAAL-TIGA strategy. The same holds for $(0, \mathtt{a})$ and $t > 90$. We also see that the expected cheapest action is clearly $(0, \mathtt{b})$ then $(0, \mathtt{a})$ and then $(100, \mathtt{w})$ again in accordance with the analysis from Section 4.

Figure 5.b We see that the algorithm learned that it should never do $(100, \mathtt{w})$ if there is any other choice, we also see that the density of $(0, \mathtt{a})$ where $0 < t < 70$ is dropping.

Figures 5.b, 5.c, 5.d, 5.e and 5.f We can observe that the density of $(0, \mathtt{a})$ in $0 < t < 70$ is gradually dropping as the algorithm gains more experience through the learning iterations and the different strategies synthesized.

Figure 5.f In this we see that the algorithm has learned that when $0 < t < 60$ it should do $(0, \mathtt{b})$, when $70 < t < 90$ it should do $(0, \mathtt{a})$, and when $t > 90$ it should do $(100, \mathtt{w})$.

There is still some uncertainty when $60 < x < 70$. However after determinizing the strategy we see that the algorithm has learned that the limit should be exactly 70, thus the algorithm has learned the exact optimal strategy.

Thus we see that the general algorithm behaves as expected, and that the algorithm converges towards the optimal strategy through the iterations. We now look into whether the theorems about DPAs presented in Section 2 holds for PTMDPs.

7 Theorems in Relation to PTMDP

As DPAs are a proper subclass of PTMDPs as argued in Appendix A, it is natural to ask whether Theorems 1 and 3 hold for PTMDPs too. Theorem 2

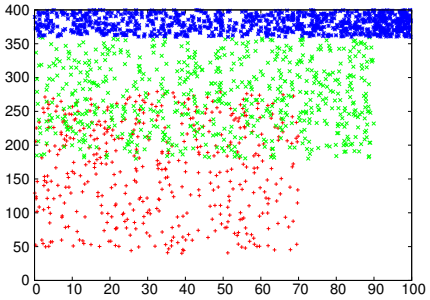

Fig. 5.a: The runs generated in simulation under the uniform strategy.

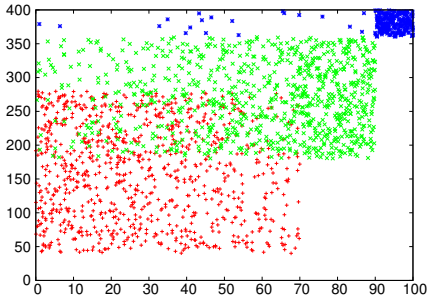

Fig. 5.b: The runs generated in the simulation under the strategy from the second iteration.

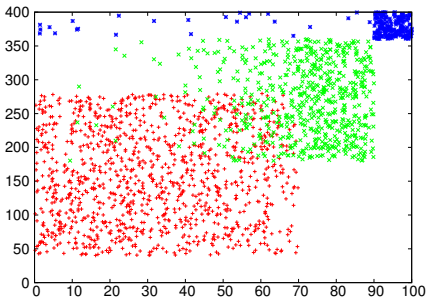

Fig. 5.c: The runs generated in the simulation under the strategy from the fourth iteration.

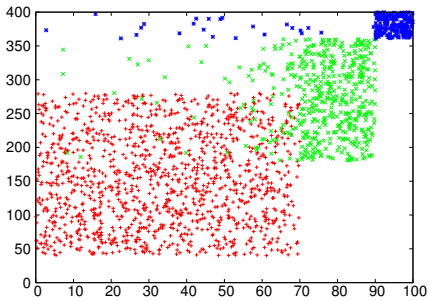

Fig. 5.d: The runs generated in the simulation under the strategy from the sixth iteration.

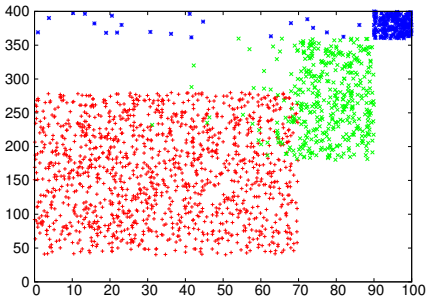

Fig. 5.e: The runs generated in the simulation under the strategy from the eighth iteration.

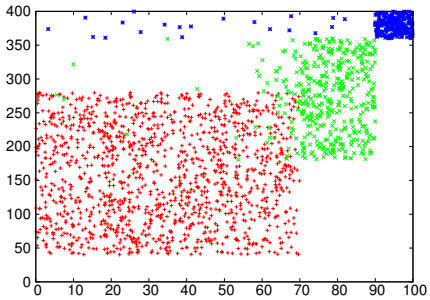

Fig. 5.f: The runs generated in the simulation under the strategy from the tenth iteration.

Fig. 5: Runs generated over the PTMDP from Figure 3 in Section 4 using the filtering method from Section 9.3 and the Logistic Regression method. The runs shown are the runs going through the `CHOICE` location. The horizontal axis shows the valuation of the *time* clock. The vertical axis shows the time to done from the point where the run entered the `CHOICE` location.
The colors denote the choice made by the strategy. Blue mean means $(100, \mathtt{w})$, green means $(0, \mathtt{a})$ and red means $(0, \mathtt{b})$.

does not make much sense in the context of PTMDPs as they have no notion of resources. In this section we will give two counter examples to the two theorems, thus proving they do not hold for the more general model.

7.1 Lazyness

In Figure 6 we give an example of a PTMDP where the optimal strategy is lazy when we want to minimize the value of c in the *END* location. The only location where we should make a choice is in the *CHOICE* location where free to be as $c' = 0$. The behavior in places A, B and C is completely deterministic, because of that the following is clear:

Choose $(d, a), d \in \mathbb{N}_{\geq 0}$ We delay d time units and then choose a. Then the value of c is 2 when entering the *END* location.

Choose $(d, b), d \in \mathbb{N}_{\geq 0}$ Delay d then choose b. The value of c is then 1 when entering the *END* location.

Choose $(d, c), d \in \mathbb{N}_{\geq 0}$ Delay d then choose c. The value of c is then 2 when entering the *END* location.

This means that the optimal strategy for *CHOICE* is:

$\nu(x) \leq 2$ Delay d s.t. $1 < \nu(x) + d \leq 2$ then do b.
$\nu(x) > 2$ Do c.

In the first case the optimal strategy requires we wait and then do an action. Thus the strategy is lazy, which means that Theorem 1 does not hold for PTMDPs in general.

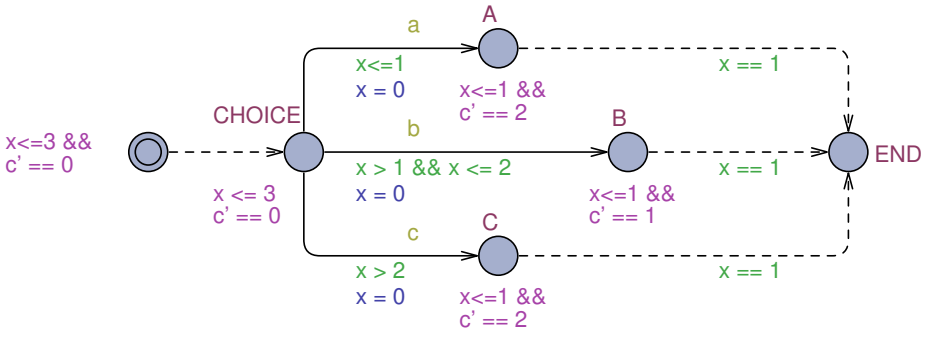

Fig. 6: PTMDP where the optimal strategy is lazy, and Theorem 3 does not hold.

7.2 Delayed states are always better

In the model in Figure 6 Theorem 3 does not hold either.

Given the optimal strategy μ^c we can see that for the two states

$$q = (\{CHOICE\}, \{x = 1.5, c = 0\})$$

and

$$q' = (\{CHOICE\}, \{x = 3, c = 0\})$$

that $\mu^c_q(0, b) = 1$ has an expected value of 1 and $\mu^c_{q'}(0, c) = 2$ which yields a value of 2. This is a counter example to the theorem as the more progressed state q' has a higher expected cost.

In the current algorithm we assume Theorem 1 is true, and in the new filtering method we present in Section 9 we shall assume Theorem 3 to be true. Thus for some PTMDPs we are not guaranteed to learn the optimal strategy. However, for some PTMDPs the best non-lazy strategy can still be an improvement compared to the uniform strategy, e.g. for the PTMDP in Figure 3 the optimal strategy is actually lazy[6]. In the remainder of this work, we concentrate on DPAs, the sub-class of the PTMDPs for which the theorems hold.

8 DPA Optimization

The DPA model is a subclass of the PTMDPs, as argued in Appendix A. This means that there are some optimizations we can only do safely when working with DPAs. These optimizations could make the learning algorithm work better. We suggest the following optimizations:

Immediate Start According to Theorem 2 starting a task with no resource-requirements can never be worse than not starting it. Therefore we restrict all our schedulers s.t. if there exists a task ready to be started with no resource-requirements, we immediately start this task. Intuitively this can lead to many different schedulers, trivially starting all the waiting tasks with no resource-requirements one by one, will, regardless of order, lead to the same state.

Resetting of clocks The "progress" of a waiting task can have no effect on the scheduler. To give the learning algorithms the least noisy data, we always assume that the clocks of the waiting states are zero. This makes these clocks insignificant in the learning algorithms. Any learning algorithm should recognize that as the clock always has the same value, no matter which decision was made, its value is insignificant. This could possibly be extended to all non active clocks for the more general class, these clocks can be found via static analysis [4].

It is easy to see that the values of clocks of SDPAs waiting to start a task is never important for the scheduler, as the value of this clock does not restrict any transitions, and it is reset when entering the next discrete state in the SDPA.

[6] However for the model in Figure 6 the algorithm is not able to learn anything.

9 Filtering

An important part of the learning algorithm is the filtering step. We will in this section explore different methods for filtering. In the filtering step the goal is to select the runs from which we should learn. These runs should be "good" runs, but should also give information about as much of the state space as possible.

The naïve approach to filtering is to simply taking the set of *maxBest* runs. In Section 9.1 we describe this naïve approach for filtering. In Section 9.4 we describe the filtering method used in [13]. This method takes the discrete part of the state into account. In Section 9.4 we present a filtering approach which also takes the continuous part of the state into account.

9.1 Stateless Filtering

The naïve filtering step simply returns the set of runs which takes the shortest time. To define this properly, we define the function time to end.

Definition 18. *Given a run generated under a lazy strategy*

$$\pi = (q_0, \nu_0) \xrightarrow{d_0} (q_0, \nu_0 + d_0) \xrightarrow{a_0} (q_1, \nu_1) \xrightarrow{d_1} (q_1, \nu_1 + d_1) \xrightarrow{a_1} \dots \xrightarrow{a_{n-1}} (q_n, \nu_n)$$

over a DPA D and a state $(q_k, \nu_k) \in \pi$, the function timeToEnd gives the time which passed from the point the run was in the state (q_k, ν_k), To the point where the run is in (q_n, ν_n) which is the end state. Formally:

$$timeToEnd(\pi, (q_k, \nu_k)) = \sum_{i=k}^{n-1} d_i$$

thus given a set of runs Π the filter returns the set Π' of runs s.t.

$$|\Pi'| = maxBest$$
$$\forall \pi' \in \Pi'. \nexists \pi \in \Pi \setminus \Pi'. timeToEnd(\pi', (q_0, \nu_0)) > timeToEnd(\pi, (q_0, \nu_0))$$

This filtering method returns a set of runs. This will not be the case in the methods explained in Section 9.3 and Section 9.5.

9.2 Learning on Stateless Filtered Runs

While the Stateless Filtering method intuitively seems sufficient, we here show that this is not always the case.

We will show this using the PTMDP in Figure 7 where we would like to minimize the value of the clock x in the `END` location.

In the `START` location the environment has two possible actions, each of which will be chosen with probability $\frac{1}{2}$; it can go to the `A` location or the `CHOICE` location. If we go to `A`, x will be 0 in the `END` location. If we go to the `CHOICE`

location the controller has to choose between actions `b` and `c`. If the controller chooses `b`, x will be 1 in the `END` location. If the controller chooses `c`, x will be 100 in the end location.

Thus the only place we need a strategy is for the `CHOICE` location. And the optimal strategy is clearly to do `b`.

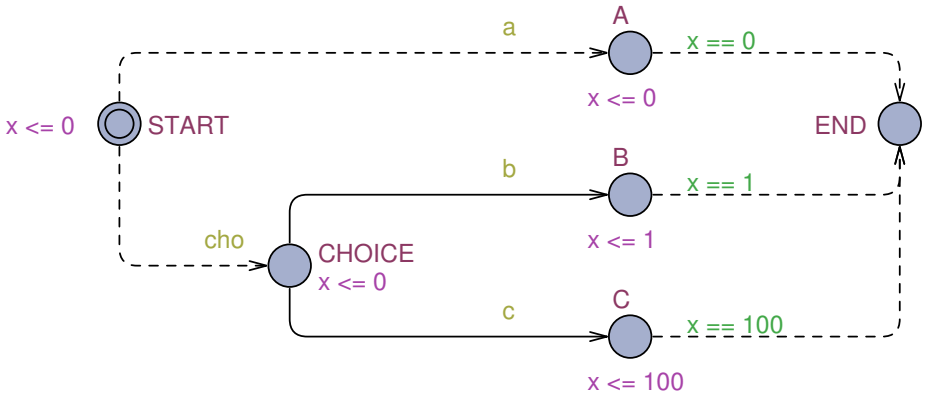

Fig. 7: PTMDP example where the Stateless Filtering method from Section 9.1 does not work.

Under the uniform strategy the expected time to completion from `START` is 25.25, and under the optimal strategy the expected time to completion from `START` is 0.5. If we use the filtering method from Section 9.1 the following will happen:

- Half the runs go through `A`, and half go through `CHOICE`.
- The runs going through `A` will have time 0 and the runs going through `CHOICE` will either have time 1 or 100.
- Assuming we choose the best half of the runs, we will choose all the runs going through `A`, and none of the runs going through `CHOICE`.

This means that the learning step will not get any runs going through `CHOICE` and thus will not learn anything about this location. As a consequence, all learning methods will suggest that we use the uniform strategy in this location, and thus we will get an expected time to completion which is 25.25 under the learned strategy.

Even though for simplicity, we here use the PTMDP formalism, a similar issue can occur in a DPA. If we look at the partial DPA in figure 8 we see that if `A0` stops when $\nu(x^A) < 5$ we will be in the discrete state $\overline{\texttt{A1}}, \texttt{B0}$ and if `A0` stops when $\nu(x^A) > 5$ we will be in the discrete state $\overline{\texttt{A1}}, \overline{\texttt{B0}}$. As intuitively runs which stop when $\nu(x^A) < 5$ are shorter than runs which stop when $\nu(x^A) > 5$ the Stateless filtering method will give more information for the discrete state $\overline{\texttt{A1}}, \texttt{B0}$, than for the discrete state $\overline{\texttt{A1}}, \overline{\texttt{B0}}$.

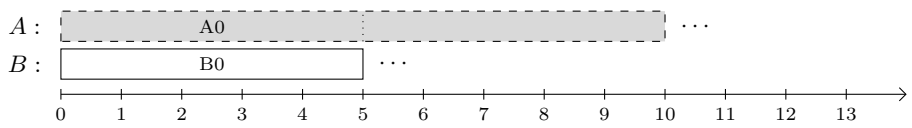

Fig. 8: A part of a DPA showing how the environment is able to decide which discrete state the DPA will be in.

9.3 Discrete State Filtering

We suggest to have a higher granularity when selecting the data to learn on. Instead of returning a set of runs, we will instead return a function $\mathbb{F}$ which goes from a discrete state q to a set of action valuation pairs.

This set will represent the actions which were advantageous in the concrete discrete state, regardless of how we got to this discrete state. We can define the returned set of action, valuation using the set of runs Π_q:

$$|\Pi_q| = maxBest$$
$$\forall \pi' \in \Pi_q . \nexists \pi \in \Pi \setminus \Pi_q . timeToEnd(\pi', (q, \nu)) > timeToEnd(\pi, (q, \nu'))$$

where ν and ν' are valuations. $\mathbb{F}(q)$ is then simply the valuation of the runs when they were in q and the action they chose in q. Note that this definition is similar to the one from Section 9.1. The difference is that here we only consider the part of the run executed after q. In Section 9.1 we considered the whole run.

If we look at the PTMDP in Figure 7 we will see that we will get action valuation pairs for the `CHIOCE` location as well as for the `A` location. This means that the learning step now has the information it needs to learn the optimal strategy and lower the expected time to go to 0.5.

9.4 Learning on Discrete State Filtered Runs

In the above we argue that we should consider both the discrete part of the state and the expected time to run when evaluating whether an *action*, *valuation*-pair should be used for learning or not. We here provide a case, in which this refined method is still not sufficient.

In Figure 9 we give an example of a DPA where it is important to also include the continuous part of the state when doing the filtering.

We have run our learning algorithm on the DPA in Figure 9. We used Logistic Regression, 2000 runs in each simulation, and used 200 $(action, valuation)$ pairs for each discrete state to learn on. We have illustrated the development of the algorithm for one discrete state, namely the one where $A0$ is running and $B0$ has finished running; and thus is in $\overline{B1}$. Each of the plots in Figure 10 shows a set of runs, the horizontal axis shows $\nu(x^A)$, the vertical axis shows the time to done from this state. And the colors denote the choice made.

We will now, based on Figure 10, give an in depth explanation of the development of the scheduler for this particular state as the algorithm iterates.

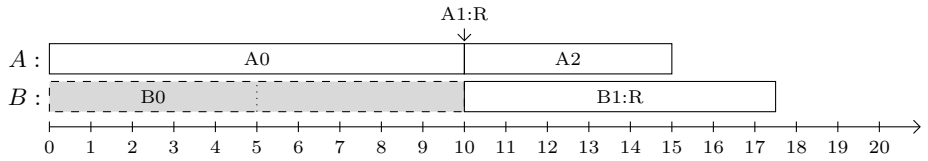

Fig. 9: A DPA, the first process first has a completely deterministic task `A0`, which do not use any resource, then has the task `A1` which takes no time, but needs resource R and then in the end it has a task `A2`. The second has two tasks, the first can end between 0 and 10, and the second always takes 7.5. The optimal scheduler in state $(\{A0, \overline{B1}\}, \nu)$ is go_2 if $\nu(x^A) < 5$ else **w**.

Figure 10.a We here see the runs generated under the uniform scheduler. We see two green lines and one red line. The red line consist of the runs which choose to start `B1` and the green lines consist of the runs that waited until `A0` was done, as described in the caption of the figure. However there are two green lines. The upper being the runs which in the later configuration $\{\overline{\texttt{A1}}, \overline{\texttt{B1}}\}$ chose to start `B1` after `A0` finished, the lower line being those who chose to start `A1` instead.
From the plot it is clear that to minimize the expected time to completion, we should never choose to start `B1` after waiting (the upper green line). It is also clear that we should choose to start `B1` if $\nu(x^A) < 50$ and we should wait till `A0` finishes, and then do `A1` if $\nu(x^A) > 50$.

Figure 10.b Here we see the runs from Figure 10.a which were selected for learning for this discrete state. What is interesting to see is that no points are selected for $\nu(x_{\texttt{A}}) < 25$. We can also see that some red points was selected for $\nu(x_{\texttt{A}}) > 50$ even though that it is clear the green points are the best here.

Figure 10.c We here see that we have learned that we should not choose `B1` after waiting, as the upper green line is much less dense. We also see that the distribution over $\nu(x_{\texttt{A}})$ of whether to wait or start `B1` is completely random, thus we have not learned anything about this discrete state.

Figure 10.d Here we can see that as the upper green line has disappeared we will not choose any red points for learning at all. This is due to the fact that we choose the 10% best runs. This also means that we only learn on runs for $\nu(x_{\texttt{A}}) > 55$.

Figure 10.e This is a set of runs generated under the best stochastic strategy found, the strategy is to almost always wait, and then start `A1`. However this is not the optimal strategy, as discussed above. The reason we have learned this is clearly due to the filtering which has made sure we only learned the optimal strategy for $\nu(x^A) > 50$ - This is then assumed to be the best strategy for the whole discrete state.

9.5 Continuous State Filtering

As we now know that the filtering is still not fine grained enough, we can refine our filtering method further. We chose to refine the method explained in Sec-

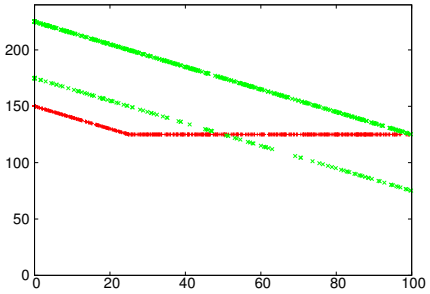

Fig. 10.a: The runs generated in simulation under the uniform scheduler.

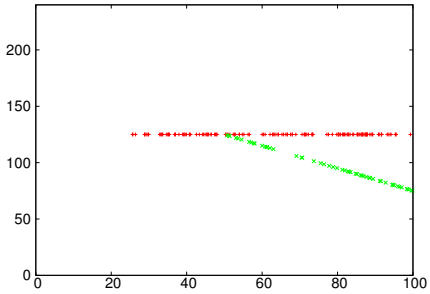

Fig. 10.b: The runs chosen for learning from Figure 10.a.

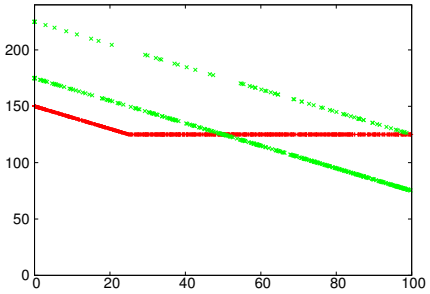

Fig. 10.c: The runs generated in the simulation under the scheduler from the first iteration.

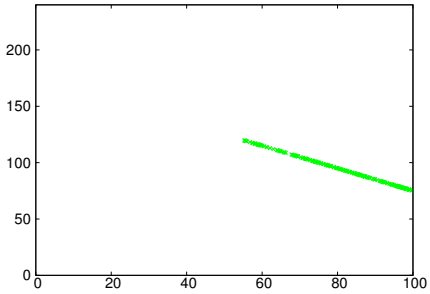

Fig. 10.d: The runs chosen for learning from Figure 10.c.

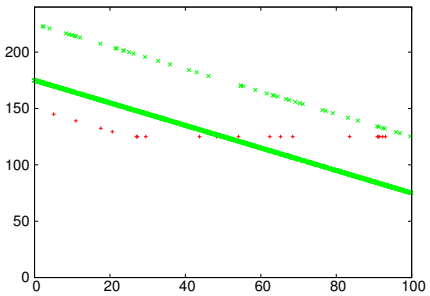

Fig. 10.e: The runs generated in the simulation under the scheduler from the second iteration. This was the best scheduler found.

Fig. 10: Runs generated over the DPA seen in Figure 9 using the filtering method from Section 9.3. The runs shown are the runs going through the state where A is doing $A0$ and B has just finished $B0$ and is waiting in $\overline{B1}$. The horizontal axis shows the valuation of the clock for process A, $\nu(x^A)$ when B finished $B0$. The vertical axis shows the time to done from the point where B finished $B0$. The colors denote the choice made by the scheduler. Red means that the scheduler choose to start $B1$, green means the scheduler decided to *wait* until $A0$ was done.

tion 9.2 by exploiting Theorem 3; if the values of the clocks are more progressed, the expected time to end is smaller. For the optimal scheduler, this means that if we have large clock valuations, we should choose runs which have low time to completion, and if we have small clock values we may choose runs which have a higher time to completion. We therefore propose a sweeping algorithm, shown in Algorithm 2, for choosing the set of action valuation pairs, given a set of runs. The algorithm will be run for each discrete state q, the results will then be aggregated to the function $\mathbb{F}$ as defined in Section 9.2.

Before looking into the algorithm, we first define an ordering $\preceq$ of (action, valuation, time) triples used internally in Algorithm 2. The ordering is defined in Algorithm 1.

Input: Two action, valuation, time triples; $(a, \nu, time)$ and $(a', \nu', time')$.
Result: `True` if $(a, \nu, time) \preceq (a', \nu', time')$ `False` otherwise

```
1 foreach x ∈ X do // We always check the clocks in the same order.
2 |   if ν(x) < ν'(x) then
3 |   |   return True;
4 |   else if ν(x) > ν'(x) then
5 |   |   return False;
6 if time ≤ time' then
7 |   return True;
8 else
9 |   return False
```

Algorithm 1: Ordering $\preceq$ of action, valuation, time triples. The ordering is used in Algorithm 2. Note that the *action* in the triples does not effect the ordering. The reason the action is in the input of the algorithm is that this makes it easier to describe Algorithm 2.

We can now describe Algorithm 2. The idea is to find the triples in av for which it holds that there is not another triple for which the valuations of all clocks are smaller, and time is also smaller. This is the property checked in Line 11. These triples are considered good for learning on due to Theorem 3; this will be more clear when we explore how the sweep filtering preforms on the example in Figure 9.

In Lines 1-6 of the algorithm we simply reformat the runs into the tuples used internally in the algorithm. In Line 7 we sort the input in preparation for the sweep. The sorting is to make it possible to only sweep over the points once, and still generate the front we are searching for. In Lines 10-16 we then find the front. We also evaluate the points not on the front using the points from the front.

As mentioned we check in Line 11 if the triple is on the front, we do this by checking if there is another triple in the front for which all clock valuations and time are smaller. It is enough to check the triples in the front due to the sorting. Let us consider the `else` branch of the `if` first. In this branch we will add the triple to the front as we did not find a better triple in the front. We then add

Input: A set of runs Π and a discrete state q.
Result: A set of action, valuation pairs.

$av = \emptyset$; `// The raw action valuation pairs. And time to completion`
foreach $\pi \in \Pi$ **do**
  if π *contains* $(q, \nu) \xrightarrow{go^j}$, $go^j, 1 \leq j \leq N\}$ **then**
    put $(a, \nu, timeToEnd(\pi, (q, \nu)))$ into av;
  else if π *contains* $(q, \nu) \xrightarrow{d}$, $a \in \mathbb{R}_{\geq 0}$ **then**
    put $(\mathtt{w}, \nu, timeToEnd(\pi, (q, \nu)))$ into av;
sort av ascending according to the ordering $\preceq$;
$Front \coloneqq \emptyset$;
$Result \coloneqq \emptyset$;
foreach $(a, \nu, time) \in av$ **do**
  if $\exists (a', \nu', time') \in Front. \forall x \in X. \nu(x) > \nu'(x) \wedge time > time')$ **then**
    choose $(a', \nu', time') \in Front$ s.t.
    $\nexists (a'', \nu'', time'') \in Front. \forall x \in X. \nu'(x) > \nu''(x) \wedge time'' < time'$;
    add $(a, \nu, time - time')$ to $Result$;
  else
    add $(a, \nu, time)$ to $Front$;
    add $(a, \nu, 0)$ to $Result$;
sort $Result$ descending on time;
`/* We only return the action, valuation part of the tripe, the time is not relevant in the learning algorithm. */`
return the first *macBest* action, valuation pairs in $Result$;

Algorithm 2: Select the action valuation pairs to use in the learning step of the main algorithm.

the triple to the *Result* set, but with $time = 0$. This indicates that this point is good as no run going trough the same state had a better time to done for this or a strictly smaller valuation. In the other branch we find the triple from the front, which have the smallest time of the triples from the front, for which all clock values are smaller than the new triple.

We then add the new triple to *Result* but with time relative to the point from the front, thus the time will be measured in relation to the point from the front, exploiting Theorem 3.

9.6 Learning on Continuous State Filtered Runs

We will now look into how this new filtering method works on the example from Figure 9. In Figure 11 we have plotted runs in the same way as in Figure 10, the only difference is that we now use the Continuous State filtering method. We will not go through these plots as thoroughly as before, as they are largely self-explanatory.

In Figure 11.b we see how the filtering works on the runs from the uniform scheduler. We can see the effect from the changes to time done by the algorithm. We can also see that already now the runs we have selected are the best possible data for the filtering algorithm. This is due to the fact that the runs selected when $\nu(x^A) \leq 50$ are the red runs (which say start $B1$) and the runs selected when $50 \leq \nu(x^A)$ are the runs that waited. This is exactly what the optimal scheduler should do.

We can see in Figure 11.c that the scheduler we synthesized is very close to the optimal. After 13 iterations we end on the scheduler which generated the runs in Figure 11.e. The final strategy says that the boundary is at $\nu(x^A) = 52$ which is very close to the optimal $\nu(x^A) = 50$.

We have in this section taken an in depth look into the filtering part of the algorithm. The naïve idea is to simply filter on the expected time to go. However as we have shown this is in some examples not enough. We have also seen that we should consider the full state of the DPA, and not just the discrete state.

In Section 11 we will investigate the three different filtering methods experimentally. We now look into the learning step of the algorithm.

10 Strategies: Data structures, Algorithms and Learning

Crucial to our reinforcement learning algorithm in Figure 3 is the efficient representation and manipulation of control strategies. In Uppaal-Tiga, non-deterministic strategies are represented using zones, e.g. sets Z of valuations described by a guard in $\mathcal{B}(X)$. In a representation R, each location ℓ has an associated finite set of zone-action pairs $R_\ell = \{(Z_1, a_1), \ldots, (Z_k, a_k)\}$, where $a_i \in \Sigma_c \cup \{\lambda\}$. Now R represents the strategy σ_R where $\sigma_R((\ell, v)) \ni a$ iff $(Z, a) \in R_\ell$ for some Z with $v \in Z$. In Uppaal-Tiga R is efficiently implemented as a hash-table with the location ℓ as key.

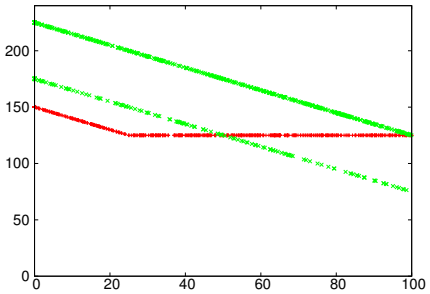

Fig. 11.a: The runs generated in simulation under the uniform scheduler.

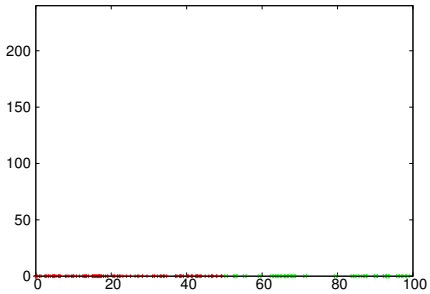

Fig. 11.b: The runs chosen for learning from Figure 11.a.

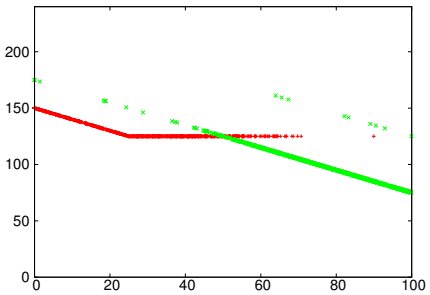

Fig. 11.c: The runs generated in the simulation under the scheduler from the first iteration.

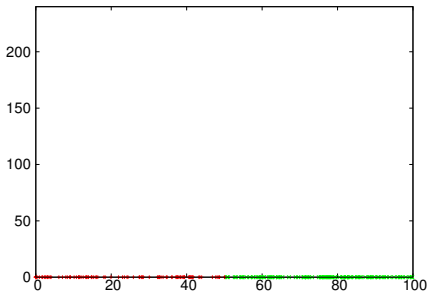

Fig. 11.d: The runs chosen for learning from Figure 11.c.

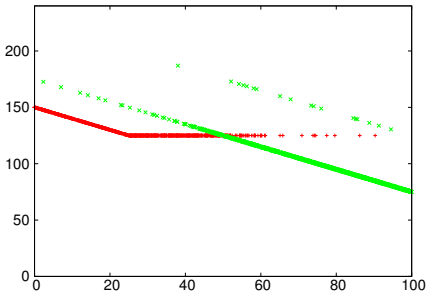

Fig. 11.e: The runs generated in the simulation under the scheduler from the 13^{th} iteration. This is the best scheduler found.

Fig. 11: Same plots as in Figure 10, but using the filtering method from Section 9.5 the reason the chosen runs all lie very low on the vertical axis is due to the filtering method, which will push the best runs to 0.

For stochastic strategies, we shall in the following restrict our attention to so-called *non-lazy* strategies1, μ^c, where the controller either urgently decides on an action, i.e. $\mu^c_q(d,a) = 0$ if $d > 0$, or prefer to wait until the environment makes a move, i.e. $\mu^c_{(\ell,v)}(d,a) = 0$ whenever $v(\texttt{time}) + d \leq T$ with T being the time-bound of the reachability property in question. We shall use $\texttt{w}$ to denote such an indefinite delay choice. Thus, for non-lazy stochastic strategies, the functionality may be recast as discrete probability distributions, i.e. $\mu^c_q : (\Sigma_c \cup \{\texttt{w}\}) \to [0,1]$. In particular, we note that any non-lazy, stochastic strategy can trivially be transformed to a deterministic strategy by always selecting the action with the highest probability.

In the following we introduce three different data structuring and learning algorithms for stochastic strategies. Given that memoryless strategies suffices, it suffices to learn a set of sub-strategies $\mu^c_\ell = \{\mu^c_q : \exists v.q = (\ell, v)\}$, where $\ell \in L$. The sub-strategies are then learned solely from a set of *(action,valuation)* pairs. Given a set of runs Π the relevant information for the sub-strategy μ^c_ℓ is given as In_ℓ:

$$In_\ell = \{(s_n, v) \in (\Sigma_c \cup \mathbb{R}) \times \mathbb{R}^X_{\geq 0} \mid (q_0 \xrightarrow{s_0}_{p_0} \ldots \xrightarrow{s_{n-1}}_{p_{n-1}} (\ell, v) \xrightarrow{s_n}_{p_n} \ldots) \in \Pi\}$$

This means that it suffices to describe a sub strategy. In Sections 10.1 and 10.2 we only describe methods for learning sub-strategies. In Section 10.3 we describe a method for learning a full strategy. As Sections 10.1 and 10.2 are taken directly from [13] the input is assumed to be a set of runs as described above. However it is trivial to change this input to be the function $\mathbb{F}$ described in Section 9.

10.1 Sample Mean and Covariance

For each controllable action c and location ℓ, we approximate the set of points representing clock valuations from which that action was successfully taken in ℓ by its sample mean and covariance matrix. Suppose we have N points corresponding to clock valuations $v_1, \ldots, v_N$. The sample mean vector $\overline{v}$ is the arithmetic mean, component-wise, for all the points: $\overline{v} = \frac{1}{N}\sum_{k=1}^{N} v_k$. The sample covariance matrix is defined as the square matrix $Q = [q_{ij}] = \frac{1}{N-1}\sum_{k=1}^{N}(v_k - \overline{v})(v_k - \overline{v})^T$.

Intuitively, if the sample covariance q_{ij} between two clocks x_i and x_j is positive, then bigger (resp. smaller) values of x_i correspond to bigger (resp. smaller) values of x_j. If it is negative, then the bigger (resp. smaller) values of x_i correspond to the smaller (resp. bigger) values of x_j. If it is zero then there is no such relation between the values of those two variables.

Note that the covariance matrix has size n^2 where n is the number of clocks but it is symmetric. Furthermore, for the matrix to be significant we need at least $n(n+1)/2$ sample points that correspond to the number of (potentially) different elements in the matrix, otherwise we default to using only the mean vector.

Distribution The purpose of this representation is to derive a distance from an arbitrary point to this "set" that is used to compute a weight for each controllable action. For a given valuation, such a distance $d(v)$ is evaluated as follows: $d(v)^2 = (u-\overline{v})^T Q^{-1}(u-\overline{v})$. If there are too few sample points then we default to using the Euclidian distance to the mean $\overline{v}$. The weight is then given by $w(v) = N \cdot e^{-d(v)}$. The weights for the different actions define a probability distribution.

Algorithm and Complexity When generating runs using SMC, controllable actions are chosen according to the represented distribution that is initialized to be uniform. The time complexity is $O(n^2)$, n being the number of clocks. For the learning phase, the covariance matrix is computed using the filtered "best" samples. Then we need to invert it (once) before the next learning phase. The time complexity is $O(n^3)$. This is done for every action.

10.2 Logistic Regression

We consider a sub strategy μ^c_ℓ where the only options are either to take a transition (a) or wait until the environment takes a transition (w) (the case with more options is addressed later). The goal is to learn the weights $\beta_0, \beta_1, \ldots, \beta_{|X|} \in \mathbb{R}$ to use in the logistic function: Equation 10.2.

$$f(v) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 \cdot v(x_1) + \cdots + \beta_{|X|} \cdot v(x_{|X|}))}},$$

where $x_1, \ldots, x_{|X|} \in X$. This function, combined with the learned weights $\beta_0, \beta_1, \ldots, \beta_{|X|}$, defines a stochastic sub-strategy s.t. $\mu^c_{(\ell,v)}(a) = f(v)$ and $\mu^c_{(\ell,v)}(\mathtt{w}) = 1 - f(v)$. Using Figure 12 we here give an intuition on how, given an input set In_ℓ, we learn the weights $\beta_0, \ldots, \beta_{|X|}$ (for details, see [14]). We assume that there exists only two options (a and $\mathtt{w}$) in the location ℓ, and (for simplicity and *wlog*) a single clock in the system. For each input $(s_n, v) \in In_\ell$:

- If $s_n = a$, construct a point at $(v(x), 1)$ where $x \in X$ is the clock. These are the triangles in Figure 12.
- Otherwise, construct a point at $(v(x), 0)$ where $x \in X$ is the clock. These are the circles in Figure 12.

We use L1-regularized logistic regression provided by LIBLINEAR [14] for fitting the function to the constructed points. The output of this process is the weights $\beta_0, \beta_1, \ldots, \beta_{|X|}$ and the result is shown in Figure 12. In the case of more than two options (e.g. if we also had an action b) we use the one-versus-all method. This method learns a function for each action[7].

Complexity The complexity of fitting the points using this method is $O(|In_\ell| + i)$ [21], where i is the number of iterations before the fitting algorithm converges thus for multiple actions, the complexity for learning is $O(c \cdot (|In_\ell| + i))$ where c is the number of options. We need to store $c \cdot |X|$ weights per location, this is the space complexity.

[7] If e.g. we have three actions, a, b and w, we will learn three functions, one which is a versus b and w, one which is b versus a and w, and one which is w versus a and b.

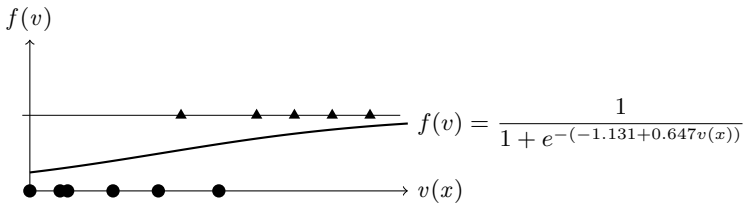

Fig. 12: Example of logistic regression with one clock x and two options a and w. For valuation v, $f(v)$ gives the probability of selecting action a (triangle) and $1-f(v)$ gives the probability of selecting action w (circle). The probabilities are equal at $v(x) = 1.747$ because $f(0.5) = 1.747$.

10.3 Splitting

In [13] we presented the splitting sub strategy, in this section we will elaborate on this method, and provide more details. Especially we will provide pseudocode for the algorithm as this was not included in [13] due to space constraints. Some paragraphs in this are slightly modified versions of paragraphs from [13].

As the output of filtering has changed, so has the input of this method. We have changed the method to comply with the changes.

Opposed to Sections 10.1 and 10.2 we in this section describe learning a full strategy instead of a sub-strategy.

We represent the strategy as a function $\mathbb{S}$ which goes from locations, l, to trees, t, $\mathbb{S}(l) = t$. An internal node in a tree is a four-tuple $(x, s, low, high)$, where *low* and *high* are either internal nodes or leaf nodes, $x \in X$ is the clock we split on and $s \in \mathbb{R}_{>0}$ is the discriminating value for the clock. A leaf node is a function W mapping actions, $a \in \Sigma_c \cup \{\mathtt{w}\}$ to weights, $W(a) \in \mathbb{R}_{>0}$, which can be normalized and then interpreted as probabilities. Figure 13 shows an example of a tree with a splitting for the clock valuation $\nu(x) = 2$.

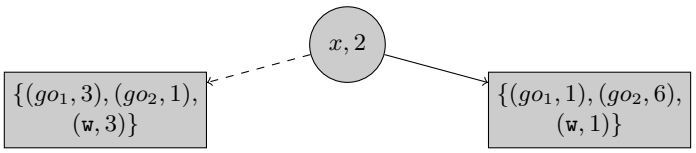

Fig. 13: A binary tree with a splitting on clock x and value 2.

For a given state (q, ν) we say that the state belongs to a leaf node W, if we by traversing the tree $\mathbb{S}(q)$ using the valuation ν reach W. Here W is represented by the pairs $(a, W(a))$ where $W(a) > 0$. This defines a stochastic strategy μ^c s.t. $\mu^c_{\ell,v}(a) = W(a) / \sum_{b \in \Sigma_c \cup \{\mathtt{w}\}} W(b)$ for all $a \in \Sigma_c \cup \{\mathtt{w}\}$. Initially, the tree consists of only a single leaf node assigning weight 1 to all actions. In each iteration of the learning algorithm presented in Section 4, a percentage of the leaf nodes are

split on one clock according to Algorithm 3 and 4 and then the leaf nodes are updated according to Algorithm 5.

When we get a function $\mathbb{F}$ from the filtering step to learn on, we will run the three algorithms. The general algorithm is:

Select nodes to split. Evaluate all the leaf nodes in the different trees, and find a set of leaf nodes which should be split. Each leaf is assigned a weight, and the leafs with the highest weights are the ones chosen for splitting. The weight assigned to a leaf is the number of good runs which did another action than the one which has the highest weight in the leaf. The selection of the leafs are done across the forest, thus leafs from different trees are compared. This is done in Algorithm 3.

Split the selected nodes. Choose one clock for each leaf selected in the previous step. Split the node on the selected clock. The clock is selected is the one which gives the biggest difference between the two resulting leafs. The formula for this difference is used in Line 11. Algorithm 4.

Update the tree. Update the strategy $\mathbb{S}$ using the function $\mathbb{F}$. Here we simply count the number of runs which did a specific action, and assign this count as the weight of the action. The weight from previous iterations is added to this count with a decay λ, done in Line 5 of Algorithm 5.

Input: The strategy $\mathbb{S}$ and a function $\mathbb{F}$ outputted from the filtering.
Result: A set of leaf nodes to be split in Algorithm 4.

```
1 toSplit := ∅;
2 foreach location l ∈ L do
3 |   foreach leaf node W ∈ 𝕊(l) do
4 |   |   Π := {(a, ν) ∈ 𝔽(l) | ν fits the constraints of W's ancestors};
5 |   |   a' := arg max_{a∈{go^j | 1≤j≤N}∪w}(W(a));
6 |   |   W.score := |{(a, ν) ∈ Π | a ≠ a'}|;
7 |   |   add W to toSplit;
8 sort toSplit descending on the scores;
9 return the first splitFrac leafs in toSplit;
```

Algorithm 3: Select nodes which will be split in Algorithm 4.

11 Experiments

In [13] we have already run experiments. We will not rerun these experiments but simply recall the results.

We see that for the DPAs from [16] our methods found the same schedulers as their analytic approach did, down to 0.5 time units on the decision boundaries. We also saw that the analytically approach used between 176 and 8547 seconds for the examples, whereas our approximation used less than 30 seconds for each

Input: A function $\mathbb{F}$ outputted from the filtering and a set of leaf nodes *toSplit* outputted from Algorithm 3.
Result: The trees are updated with the split leaf nodes.

foreach *leaf* $W \in toSplit$ **do**
  $\Pi \coloneqq \{(a, \nu) \in \mathbb{F}(l) \mid \nu$ fits the constraints of W's ancestors$\}$;
  $best \coloneqq -\infty$;
  `// Find the clock to split`
  foreach *clock* $x \in X$ **do**
    $min \coloneqq \min_{(a,\nu)\in\Pi}(\nu(x))$;
    $max \coloneqq \max_{(a,\nu)\in\Pi}(\nu(x))$;
    $mid \coloneqq \frac{max-min}{2} + min$;
    $\Pi_{left} \coloneqq \{(a, \nu) \in \Pi \mid \nu(x) \leq mid\}$;
    $\Pi_{right} \coloneqq \{(a, \nu) \in \Pi \mid \nu(x) > mid\}$;
    $w = \sum_{a' \in \{go^j \mid 1 \leq j \leq N\} \cup \{\mathtt{w}\}} (|\{((a, v) \in \Pi_{left} \mid a = a'\}| - |\{((a, v) \in \Pi_{right} \mid a = a'\}|)^2$;
    if $w > best$ **then**
      $best \coloneqq w$;
      $bestC = x$;
  `// Split on the chosen clock`
  $min \coloneqq \min_{(a,\nu)\in\Pi}(\nu(bestC))$;
  $max \coloneqq \max_{(a,\nu)\in\Pi}(\nu(bestC))$;
  $mid \coloneqq \frac{max-min}{2} + min$;
  $new \coloneqq (bestC, mid, W_1, W_1)$; `//` W_1 `denotes a leaf where all weights are` 1
  if $W = W.parent.left$ **then**
    $W.parent.left \coloneqq new$;
  else
    $W.parent.right \coloneqq new$;

Algorithm 4: Split the nodes selected in Algorithm 3.

Input: A strategy $\mathbb{S}$ and a function $\mathbb{F}$ outputted from the filtering.
Result: A set of leaf nodes to be split in Algorithm 4.

foreach *location* $l \in L$ **do**
  foreach *leaf node* $W \in \mathbb{S}(l)$ **do**
    $\Pi \coloneqq \{(a, \nu) \in \mathbb{F}(l) \mid \nu$ fits the constraints of W's ancestors$\}$;
    foreach *action* $a' \in \{go^j \mid 1 \leq j \leq N\} \cup \{w\}$ **do**
      $W(a) \coloneqq \lambda \cdot W(a) + |\{(a, v) \in \Pi \mid a = a'\}|$; `//` λ `is a parameter of decay`

Algorithm 5: Update the trees in the forest according to the new batch of runs.

example. Thus an order of magnitude faster. We also see that it depends on the example which of the learning methods is the best at finding the best scheduler. We see however that all the methods are capable of finding a scheduler which is significantly better than the uniform scheduler. We also see that the overhead of learning constrained by a UPPAAL-TIGA strategy is very low.

Lastly we note that the all the methods provide a reasonable improvement to the uniform scheduler and therefore we here only compare the different methods to each other, and not to the uniform scheduler.

We will now do a more in depth comparison of the filtering and learning methods. To evaluate the methods, we here choose a subset of all the parameters to evaluate. We will focus in the following parameters

- Comparison of the learning methods proposed in [13]; *Co-Variance Matrices*, *Splitting* and *Logistic Regression*.
- Comparison of the filtering methods from Section 9: *Stateless Filtering*, *Discrete State Filtering* and *Continuous State Filtering*.
- The effect of using a different fraction of runs for learning.

We conducted tests on the DPAs used by Kempf. et. al in [16] as well as randomly generated DPA's used in [13]. We also only investigate one parameter at a time to provide a better overview. For dimensions not reported, we always use the aggregated value across that dimension for the three reported values; mean, maximal and minimal.

In the experiments we set the parameters from Figure 4 as follows; $maxRuns := 2000$, $maxGood := 2000$, $maxbest := 1000$, $evalRuns := 2000$, $maxNoBetter := 10$, $maxIterations := 200$ and $maxResets := 7$. In the experiments investigating the effect of the relations between $maxGood$ and $maxbest$, we set $maxGood = 2000$ and $maxbest = 200$.

We will use UPPAAL SMC for evaluation of the synthesized schedulers, here we will evaluate the schedulers using UPPAAL SMC with `-E 0.005` option[8]. We will report running-time, memory consumption, scheduler performance. Due to the number of parameters, we will here only report representative results, while the full result-sets will be available online[9].

Values reported For stability, all experiments have been repeated 10 times. We therefore here report three values which are an aggregated result; The minimum, the maximum and the average of all repetitions. For each of the experiments we monitored the (by UPPAAL SMC estimated) expected time to completion given a scheduler as well as the time and memory consumption.

As our experiments are of varying size, we normalize and report the deviation of the result as a fraction of the average of all our measurements for a given experiment. If we did 4 repetitions of experiment A and got the values 20, 15, 5, 0 we would report these results as $max = 20$, $mean = 10$, $min = 0$ and then

[8] This will result in an evaluation of the strategy using 3688 runs.
[9] The results and the models are published at `http://goo.gl/qmigZW`

normalize to the values according to the mean. We say that an experiment with a normalized value of 1 is exactly on the mean, while an experiment with the normalized value of 1.1 is 10% worse.

The Naïve method For comparison, and to evaluate if our more complex learning methods provide an improvement, we also introduce a naïve method for learning. In this method, we simply, for each state, remember how many $(action, valuation)$ pairs we get from the filtering method, which chose some specific action. This value then represent our weights. This is what we refer to in Section 3.2 as a static scheduling policy, which is implemented as the filtering method in Section 10.3, but without splitting at any point in time, thus only using Algorithm 5.

11.1 DPAs by Kempf et. al.

In this section we look into how the proposed methods perform on the DPAs provided by Kempf. et. al.

The Learning Methods Effect on Synthesized Schedulers We investigate the different learning methods compared to each other. In Figures 14.a and 14.b we show sets of experiments comparing the different methods. The difference between the two figures is that in Figure 14.a we use half of the runs from the simulation for learning and in Figure 14.b we only use 10% of the runs. As we will see when comparing running times, this gives a slightly faster learning.

In Figure 14.a we see that for some examples the four methods perform equally well. For the others we see that the Naïve method is consistently worse. We also see that Splitting and Logistic Regression are almost always performing equally well. The results from the Co-Variance method are interesting, as the results vary a lot. This is because the method is more sensitive to which of the filtering methods we use than the other methods. We see that discrete filtering and stateless filtering gives the worst schedulers for the given experiment, while the continuous state filtering gives the best scheduler.

We see the same patterns in Figure 14.b, however it seems like the Naïve method is even more sensitive to the filtering methods when using less data in the learning step.

The Filtering Methods Effect on Synthesized Schedulers In the same way as in the previous section, we have compared the different filtering methods. In Figure 17.a we can see that when we use half of the runs there are almost no difference to which filter we use. However in Figure 17.b, when we only use 10% of the runs, the performance of the Stateless filtering clearly depends on which method for learning we use. The two remaining filtering methods seem largely unaffected by this change.

Speed of learning If we compare Figure 15.a and Figure 15.b, it is clear that the number of runs we use to handle in our filtering affect the running time. What we can also see, is that the Naïve learning method is the fastest of the four, often followed by the Co-Variance and Splitting methods. We note that the Linear Regression method is the slowest of all. If we compare Figure 18.a and Figure 18.b we see that the Continuous state filtering is noticeably slower when using the 50% best runs for learning. The remaining methods seem to behave as expected, with the 10%-experiments being the fastest.

Memory Consumption When looking at the memory consumption in Figure 16.a, it is clear that the Splitting method by far is the most memory hungry of all the methods using up to 2.5 times more memory than the average. We can also see that the remaining three methods are fairly on par, with the Naïve being the most memory efficient of the methods proposed. If we instead look at the memory consumption, depicted in Figure 19.a aggregated by the filtering method, we can see that the Continuous State-method requires slightly more memory on average. If we compare Figure 16.a with 16.b and Figure 19.a with 19.b, it is apparent that using a lower percentage of the runs also leads to a lower memory consumption. We can also see that the Continuous State Filtering method is an exception to this, as it needs to keep all runs in memory for off-line filtering. As the number of runs we generate does not change, but only the number of runs used in the learning, this is to be expected.

11.2 Randomly generated DPAs

Here we show the results of the experiments run on the randomly generated DPA's. The DPA's are generated in size varying from three to five processes and 3 to 10 tasks. All other parameters; duration of tasks, number of resources and resource-requirements pr. task have been selected at random. We use the same 10 generated DPAs for this test-set.

The Learning Methods Effect on Synthesized Schedulers As Figure 14.c shows, the variations in the synthesized schedulers are higher than in Section 11.1. The variations from the mean value are in the interval $[-0.9, 1.16]$. Looking more thoroughly at the data, we can see that for the two, in terms of state-space, largest experiments, Splitting is performing exceptionally bad. This indicates that the Splitting method is having trouble with stability, stretching the evidence it gets too thin. We can also observe that Logistic Regression in general is the best performing, while the Naïve method and the Co-Variance method are slightly worse, but often more dependent of the filtering method used.

The Filtering Methods Effect on Synthesized Schedulers From Figure 17.c it is obvious that the Discrete State filtering method is performing the best on average, followed by the Stateless filtering method. If we dig a bit deeper in the

data, we can observe that the Continuous State Filtering Method is often a close contender if combined with the Logistic Regression or the Co-Variance methods. We also observe that specifically for the largest examples the Continuous State filtering method is performing the worst.

Speed of learning Consistent with Section 11.1 we can in Figure 15.c see that Logistic Regression is the slowest of the methods, followed by the Co-Variance method. For the last two, there seems to be no clear winner. If we aggregate on the filtering-method instead, we can in Figure 18.c see that the Continuous State filtering method provides both the worst and the best running-times. Here we suspect the guiding through of the state space for the best running-times. As the complexity of the algorithm is much higher than the two other filtering methods, we expect this to explain the worst running times. We can also see that the Stateless filtering method is consistently worse than the Discrete State filtering method, employing that the Discrete state filtering method is better at guiding the search through the state-space.

Memory Consumption As in Section 11.1 we can in Figure 16.c see that the Splitting method by far is the most memory consuming method. We can also see, that as the size of the state-space grows, the significance of the memory consumption of the learning methods drop. Logistic Regression and the Naïve methods seems to have the lowest memory requirements which is also to be expected as their representation is the smallest. If we instead analyze Figure 19.c, we can see that the filtering methods in large have no major difference. The slight variations in the maximal consumption seem to consistently favor the Continuous State filtering method on the medium-sized examples. We expect that this is due to the more fine-grained exclusion of seemingly unfavorable actions, leading to a smaller state-space.

12 Conclusion

In this work we elaborated and extended the work done in [13]. Specifically, we exploited properties specific to the DPA-model for optimizations. We also elaborated on details left out in [13] due to space constraints. We specifically elaborated on the filtering methods used and the evolution of a scheduler during the learning algorithm as well as the Splitting method. Furthermore we proposed a novel filtering method to improve on the quality of the data used for learning as well as two naïve methods for learning and filtering respectively.

All of these methods and optimizations have been implemented as a part of the model-checker Uppaal, in such a way that different modules of the main learning-algorithm from Figure 4 can be easily changed.

In the final part of our work, we empirically investigated the performance of our implementation of the proposed methods. In general we found that our more refined learning methods, such as Logistic Regression and Splitting, can

have significant improvements over Co-Variance and Naïve methods. We also observed that the granularity of the filtering method preferred is dependent on the specific learning method. In general we did not see any significant improvements from using our novel Continuous State filtering method over the Discrete State Filtering method. On the contrary, Continuous State filtering did at times come with a penalty. This result was unexpected, and reveals that the test-cases might not be covering enough or complex enough to reap the benefits from the more advanced approach. We did however see that Stateless filter in most settings is outperformed by the two others. If we investigate the time and memory-consumption the filtering method used seems of minor significance of reasonably sized models, and is instead dependent on the learning methods. The memory consumption for the learning methods varying greatly, showing that the Splitting method spends significantly more memory on its tree structure. Concerning time, it became clear, on the larger experiments, that Logistic Regression comes with a severe performance penalty, possibly leaving room for implementation optimizations.

In general, the small difference between the performance of the schedulers synthesized indicate that the experiments in lack complexity.

12.1 Further Work

All learning methods presented in this work assumes Theorem 1 to be true. However as shown in Section 7.1 this theorem does not hold for PTMDPs. A learning method which does not rely on Theorem 1 to be true is obvious further work.

In the same way the Continuous State Filtering method currently relies on Theorem 3 which does not hold for PTMDPs either. We expect the method could be refined, and be better performing on this more general model with more complex problems. In relation to this, the optimizations we suggested and the non-lazyness theorem by [16] might hold for a larger subclass of PTMDP.

We also saw that our experiments seemed to lack complexity. Further work therefore involves constructing more advanced experiments as well as applying the method in real life scenarios.

Further work also includes making algorithms for zonifying the strategies generated by the different learning methods, as this will allow for model checking the model under a scheduler.

We have looked into two of the five steps of the algorithm. In the other steps we have either used already available tools or the most naive approach. Further work includes looking at each of these steps in more detail.

Furthermore, we have only concentrated on reachability for DPAs and PTMDPS. Further work includes optimizing for other properties, and possibly in other models, such as models with imperfect information.

13 Bibliographical Note

The work presented in this paper is largely based on work currently under review [13] made jointly with Alexandre David, Kim Guldstrand Larsen, Alex Leagy, Didier Lime and Mathias Grund Sørensen. This work is a continuation of our 8^{th} semester project, made jointly with Mathias Grund Sørensen. Both of these are attached to this thesis.

To make this thesis self-contained we have included sections from these other works, providing the main theory and models. The material presented in Section 1.1, Section 2 and Section 3 is directly from the 8^{th} semester project. Exceptions to this are Theorem 3 and Lemma 1. Section 4 is a summery of the work currently under review [13], Section 4.1 is taken directly from [13], the authors of this thesis had minor contributions to this. Section 10 is also taken directly from [13], however this section was mainly written by the authors of this paper together with Mathias Grund Sørensen. This is except Section 10.1 which the authors of this thesis only had minor contributions to. Section 10.3 is an extended version of a section from [13]. The authors of this paper also had only minor contributions to Appendix A.

The work in the sections not mentioned here are novel to this thesis.

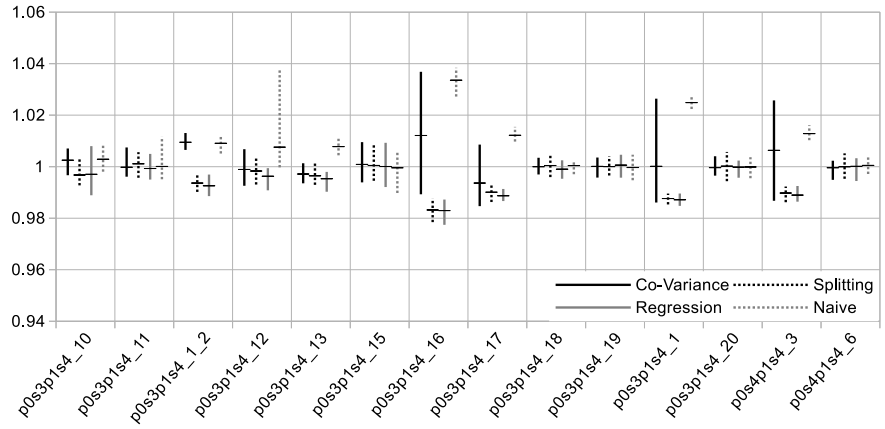

Fig. 14.a: The DPAs by Kempf el. al. [16] where we learn on half of the runs.

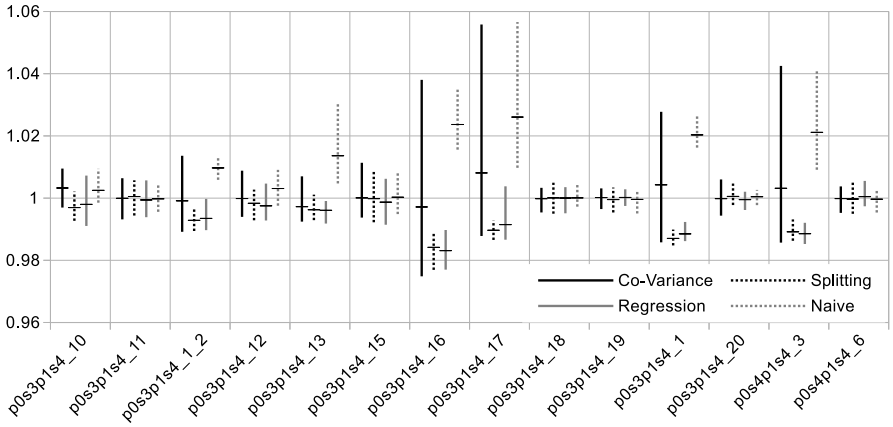

Fig. 14.b: The DPAs by Kempf el. al. [16] where we learn on 10% of the runs.

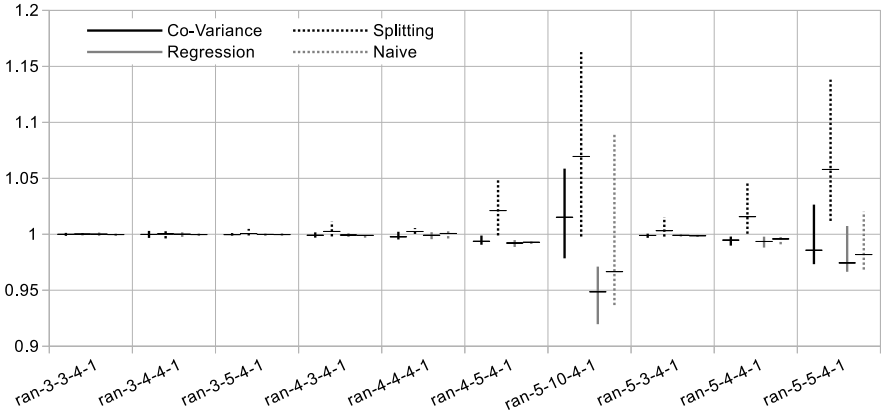

Fig. 14.c: Randomly generated DPAs where we learn on half of the runs.

Fig. 14: The expected time to go under the best scheduler found for the different learning methods.

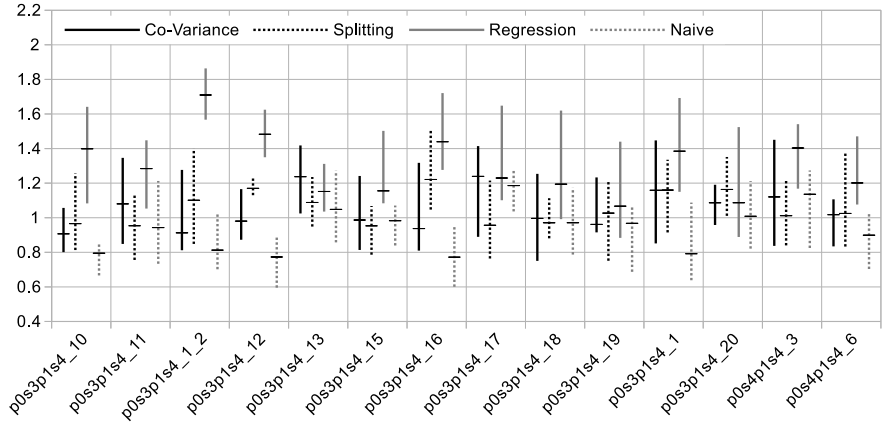

Fig. 15.a: The DPAs by Kempf el. al. [16] where we learn on half of the runs.

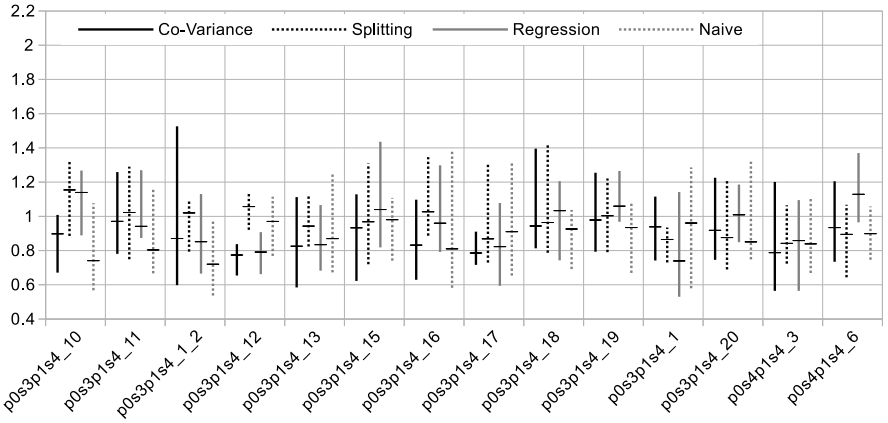

Fig. 15.b: The DPAs by Kempf el. al. [16] where we learn on 10% of the runs.

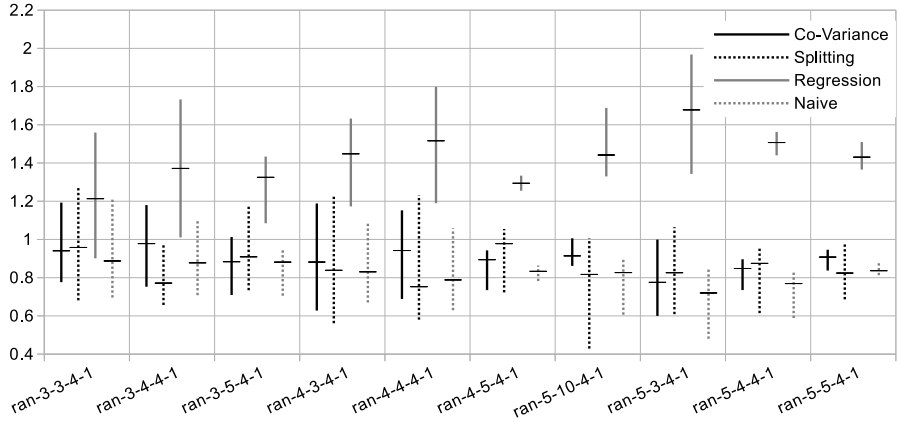

Fig. 15.c: Randomly generated DPAs where we learn on half of the runs.

Fig. 15: The time used by the algorithm for the different learning methods.

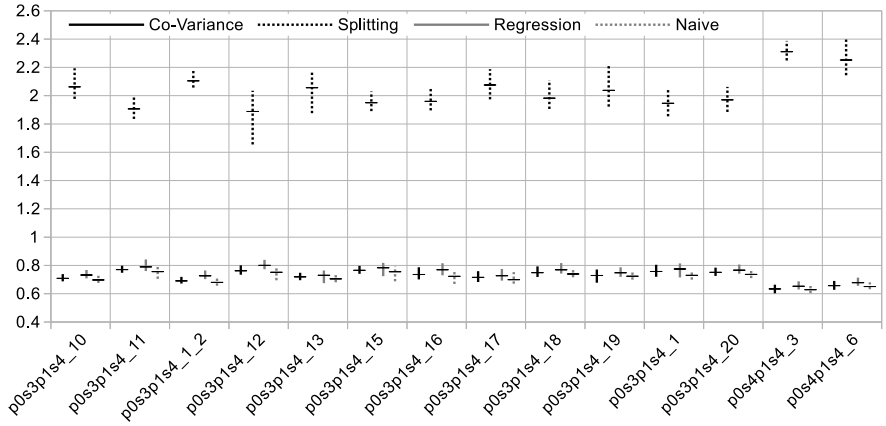

Fig. 16.a: The DPAs by Kempf el. al. [16] where we learn on half of the runs.

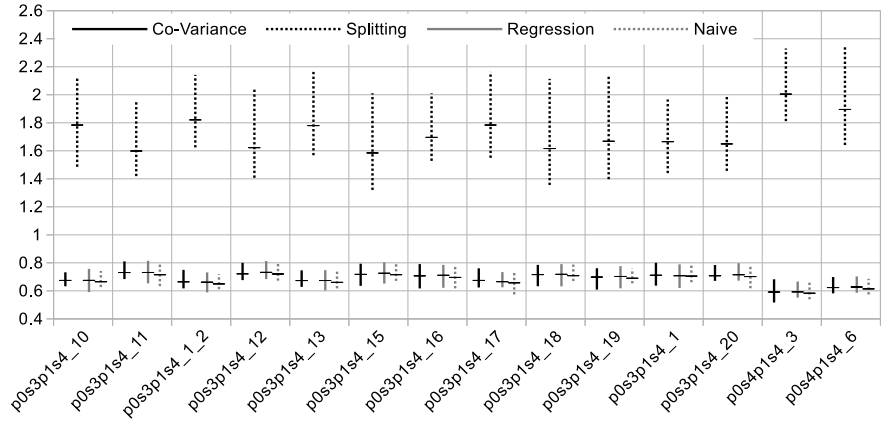

Fig. 16.b: The DPAs by Kempf el. al. [16] where we learn on 10% of the runs.

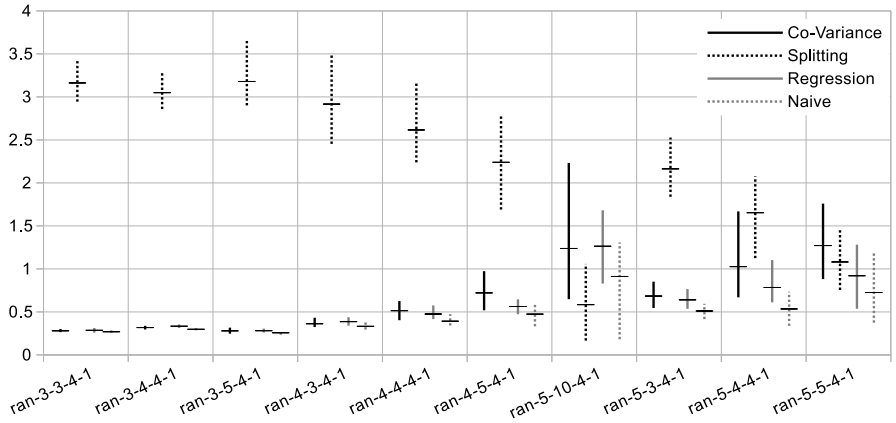

Fig. 16.c: Randomly generated DPAs where we learn on half of the runs.

Fig. 16: The memory used by the algorithm for the different learning methods.

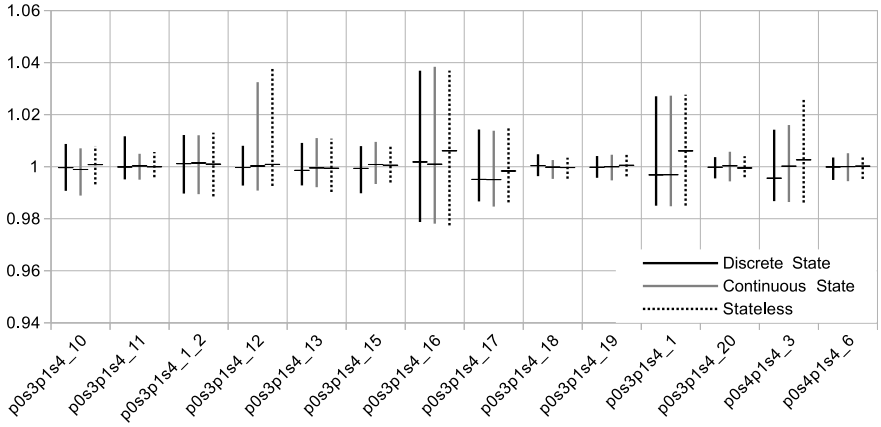

Fig. 17.a: The DPAs by Kempf el. al. [16] where we learn on half of the runs.

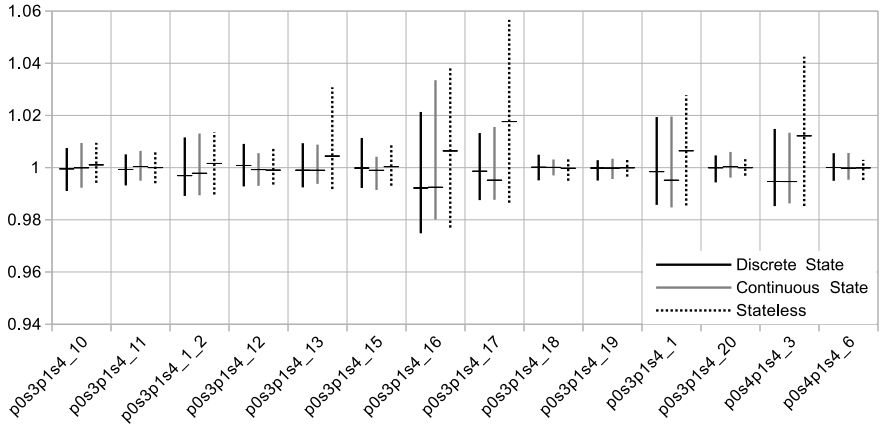

Fig. 17.b: The DPAs by Kempf el. al. [16] where we learn on 10% of the runs.

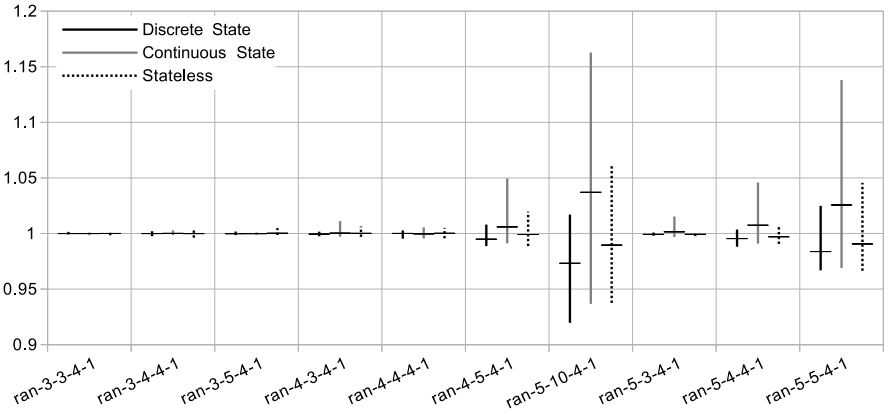

Fig. 17.c: Randomly generated DPAs where we learn on half of the runs.

Fig. 17: The expected time to go under the best scheduler found for the different filtering methods.

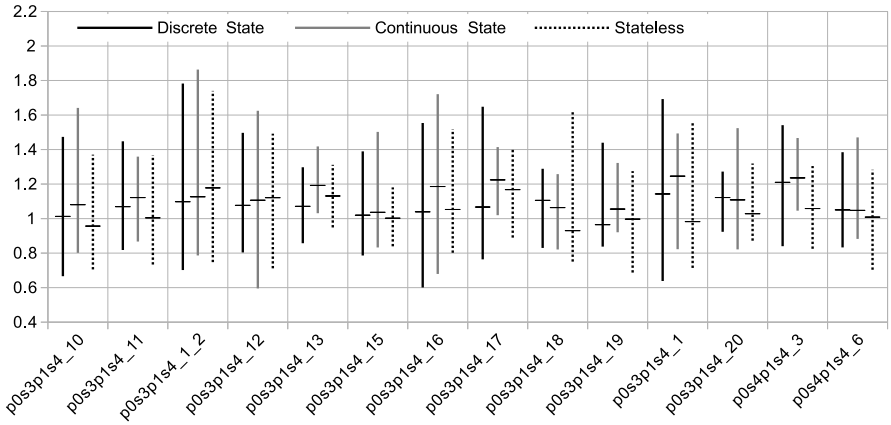

Fig. 18.a: The DPAs by Kempf el. al. [16] where we learn on half of the runs.

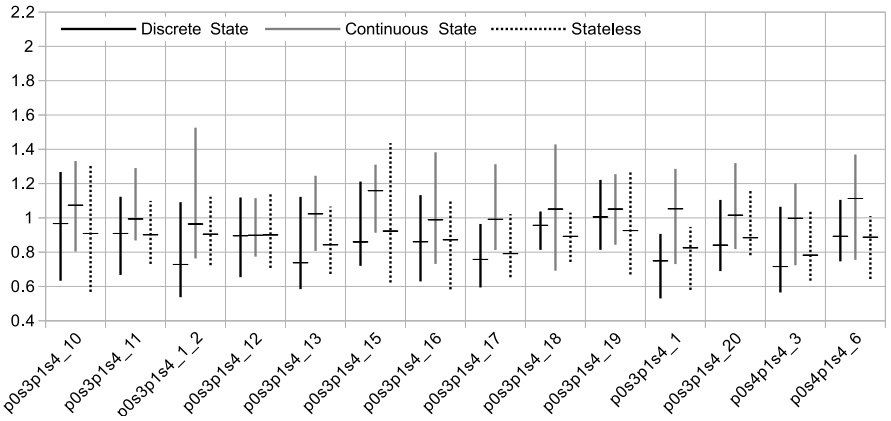

Fig. 18.b: The DPAs by Kempf el. al. [16] where we learn on 10% of the runs.

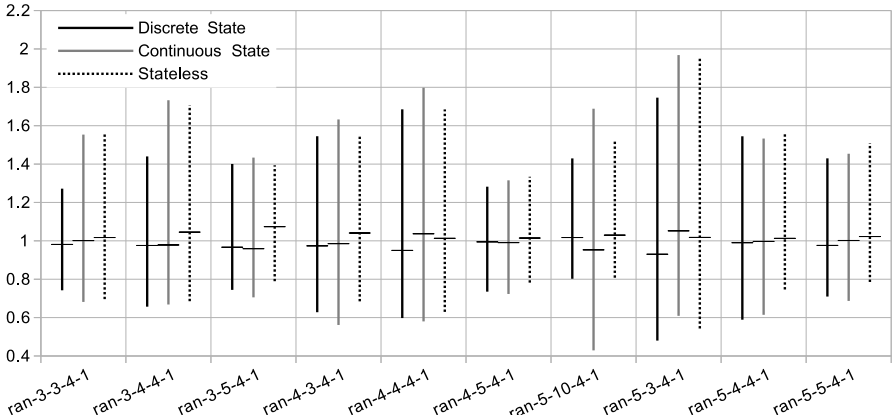

Fig. 18.c: Randomly generated DPAs where we learn on half of the runs.

Fig. 18: The time used by the algorithm for the different filtering methods.

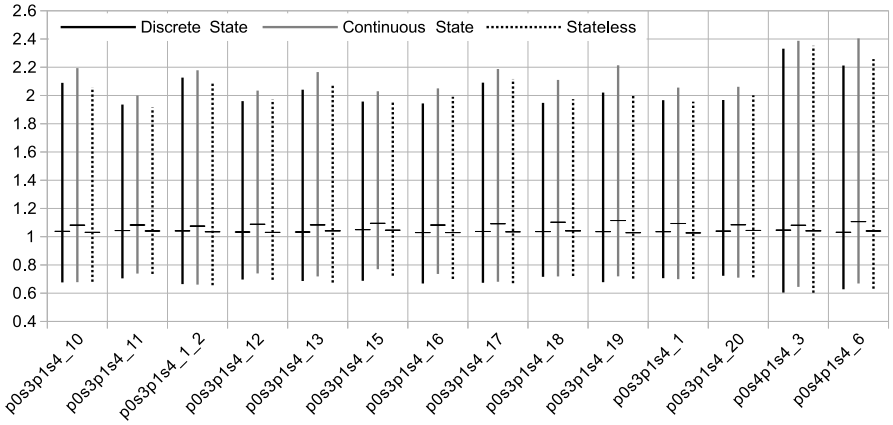

Fig. 19.a: The DPAs by Kempf el. al. [16] where we learn on half of the runs.

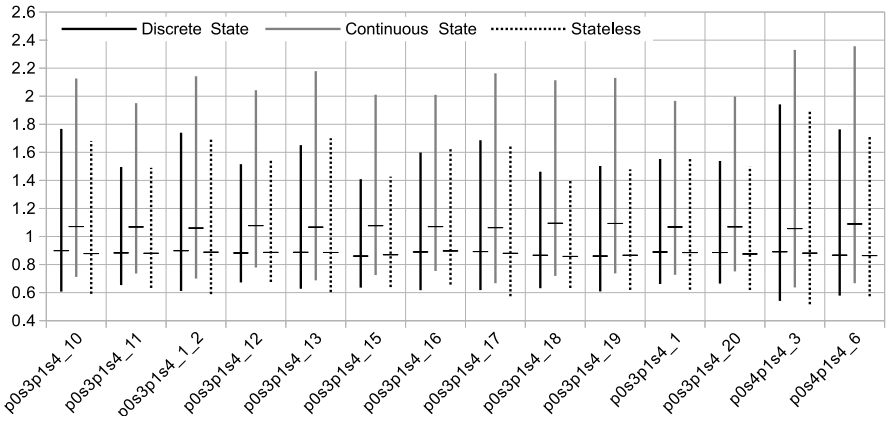

Fig. 19.b: The DPAs by Kempf el. al. [16] where we learn on 10% of the runs.

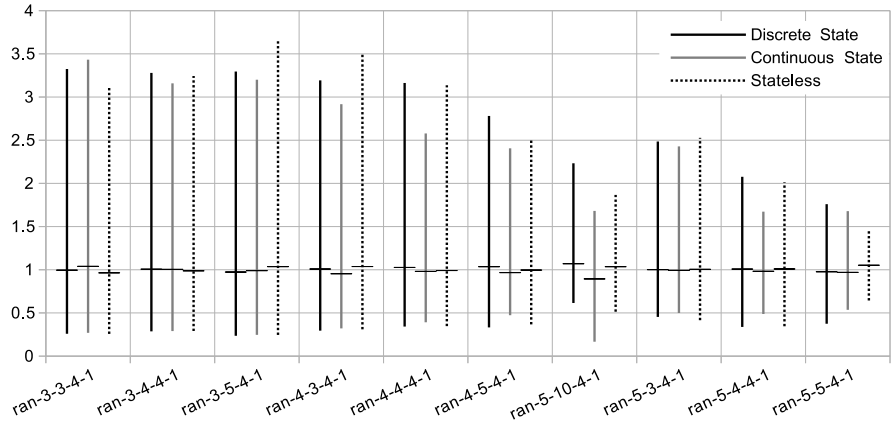

Fig. 19.c: Randomly generated DPAs where we learn on half of the runs.

Fig. 19: The memory used by the algorithm for the different filtering methods.

References

1. Yasmina Abdeddaïm, Eugene Asarin, and Oded Maler. On optimal scheduling under uncertainty. In Hubert Garavel and John Hatcliff, editors, *TACAS*, volume 2619 of *Lecture Notes in Computer Science*, pages 240–253. Springer, 2003.
2. Yasmina Abdeddaïm, Abdelkarim Kerbaa, and Oded Maler. Task graph scheduling using timed automata. In *Proc. FMPPTA'03*, 2003.
3. Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.
4. Gerd Behrmann, Patricia Bouyer, Emmanuel Fleury, and KimG. Larsen. Static guard analysis in timed automata verification. In Hubert Garavel and John Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619 of *Lecture Notes in Computer Science*, pages 254–270. Springer Berlin Heidelberg, 2003.
5. Gerd Behrmann, Agnès Cougnard, Alexandre David, Emmanuel Fleury, Kim Guldstrand Larsen, and Didier Lime. Uppaal-tiga: Time for playing games! In Werner Damm and Holger Hermanns, editors, *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 121–125. Springer, 2007.
6. Gerd Behrmann, Ansgar Fehnker, Thomas Hune, Kim Guldstrand Larsen, Paul Pettersson, Judi Romijn, and Frits W. Vaandrager. Minimum-cost reachability for priced timed automata. In Maria Domenica Di Benedetto and Alberto L. Sangiovanni-Vincentelli, editors, *HSCC*, volume 2034 of *Lecture Notes in Computer Science*, pages 147–161. Springer, 2001.
7. Gerd Behrmann, Kim G. Larsen, and Jacob I. Rasmussen. Resource-optimal scheduling using priced timed automata. In *SIGMETRICS Perform. Eval. Rev*, pages 220–235. Springer, 2004.
8. Jonathan Bogdoll, Arnd Hartmanns, and Holger Hermanns. Simulation and statistical model checking for modestly nondeterministic models. In Jens B. Schmitt, editor, *MMB/DFT*, volume 7201 of *Lecture Notes in Computer Science*, pages 249–252. Springer, 2012.
9. Thomas Brihaye, Véronique Bruyère, and Jean-François Raskin. On optimal timed strategies. In Paul Pettersson and Wang Yi, editors, *FORMATS*, volume 3829 of *Lecture Notes in Computer Science*, pages 49–64. Springer, 2005.
10. Véronique Bruyère, Emmanuel Filiot, Mickael Randour, and Jean-François Raskin. Meet your expectations with guarantees: Beyond worst-case synthesis in quantitative games. In Ernst W. Mayr and Natacha Portier, editors, *STACS*, volume 25 of *LIPIcs*, pages 199–213. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2014.
11. Franck Cassez, Alexandre David, Emmanuel Fleury, Kim G. Larsen, and Didier Lime. Efficient on-the-fly algorithms for the analysis of timed games. In *IN CONCUR 05, LNCS 3653*, pages 66–80. Springer, 2005.
12. Alexandre David, Dehui Du, Kim G. Larsen, Axel Legay, Marius Mikucionis, Danny Bøgsted Poulsen, and Sean Sedwards. Statistical model checking for stochastic hybrid systems. In Ezio Bartocci and Luca Bortolussi, editors, *HSB*, volume 92 of *EPTCS*, pages 122–136, 2012.
13. Alexandre David, Peter Gjøl Jensen, Kim Guldstrand Larsen, Alex Leagy, Didier Lime, Mathias Grund Sørensen, and Jakob Haahr Taankvist. On time with minimal expected cost!, 2014. Manuscript.
14. Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research*, 9:1871–1874, 2008.

15. D. Henriques, J.G. Martins, P. Zuliani, A. Platzer, and E.M. Clarke. Statistical model checking for markov decision processes. In *Quantitative Evaluation of Systems (QEST), 2012 Ninth International Conference on*, pages 84–93, 2012.
16. Jean-Francois Kempf, Marius Bozga, and Oded Maler. As soon as probable: Optimal scheduling under stochastic uncertainty. In Nir Piterman and Scott A. Smolka, editors, *TACAS*, volume 7795 of *Lecture Notes in Computer Science*, pages 385–400. Springer, 2013.
17. Richard Lassaigne and Sylvain Peyronnet. Approximate planning and verification for large markov decision processes. In Sascha Ossowski and Paola Lecca, editors, *SAC*, pages 1314–1319. ACM, 2012.
18. Xinxin Liu and Scott A. Smolka. Simple linear-time algorithms for minimal fixed points (extended abstract), 1998.
19. Yasmina Abdedda M and Oded Maler. Job-shop scheduling using timed automata.
20. Oded Maler, Amir Pnueli, and Joseph Sifakis. On the synthesis of discrete controllers for timed systems (an extended abstract). In *STACS*, pages 229–242, 1995.
21. Guo-Xun Yuan, Chia-Hua Ho, and Chih-Jen Lin. An improved glmnet for l1-regularized logistic regression. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, pages 33–41, New York, NY, USA, 2011. ACM.

A DPA Encoding

We illustrate our encoding of DPAs on one concrete example, the one referred as p0s4p1s4_1 in [16]. The DPA input is as follows:

```
/* Resources */  res1:1;
/* Processes */  P1: [2,4].<res1:1>[2,6].[3,4];
                 P2: [2,6].<res1:1>[3,8].[1,5];
```

We have one unit of one resource `res1` available. Process `P1` runs between 2 and 4 time units with no resource, needs 1 unit of `res1` for between 2 and 6 time units, and finishes to run between 3 and 4 time units. Similarly, `P2` runs between 2 and 6 time units, needs `res1` for 3 to 8 time units, and runs between 1 and 5 time units. We translate automatically these DPAs to the PTMDP shown in Fig. 20, where the delays of the uncontrollable moves are resolved by uniform distributions with races between the two processes.

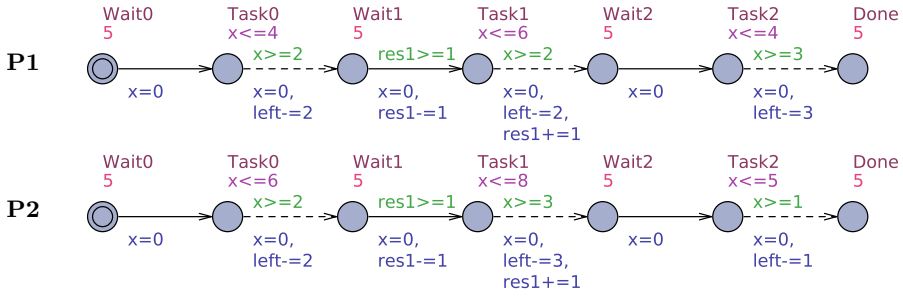

Fig. 20: TGs/PTMDPs corresponding to the processes P1 and P2.

For the purpose of SMC, we need to set exponential rates in locations with unbounded delays. These locations are waiting locations before starting a task (with or without resources). The pattern is to have a succession of wait-task locations. The task locations are entered if the constraints of the needed resources are met. These transitions also consume the resources. The lower bounds of the tasks are encoded into the guards of transitions exiting the tasks and the lower bounds into the invariants of these tasks. Finishing a task is not under the control of the controller player.

In addition, we also generate queries and decorate the model with a minimum time that is left to complete each job/process. This is encoded in the variable left. It is used by UPPAAL-TIGA to prune the search. The generated queries are:

```
// Time-optimal strategy, initial upper bound: 33,
// guaranteed lower bound on the time-to-completion:
// max(P2.left, P1.left).
control_t*(33, (P2.left >? P1.left)): A<> P1.Done && P2.Done
```

```
// Probability evaluation with tiga-strategy
Pr[<=1000](<> P1.Done && P2.Done)

// Learnt constrained strategy
control[<=1000]: A<> P1.Done && P2.Done

// Probability evaluation with constrained SMC strategy
Pr[<=1000](<> P1.Done && P2.Done)
```

The user can then generate an optimal UPPAAL-TIGA strategy with guaranteed upper bound. This should be done with the option `-w2` to make it most permissive. Then it is possible to apply learning on top of it to pick the ones with best expected time.