Efficiently Finding Ratio-Optimal Infinite Cycles in Doubly-Priced Timed Automata

by

Nicklas S. Johansen, Kristian Ø. Nielsen, and Rasmus G. Tollund
{nslorup, kristianodum, rasmusgtollund}@gmail.com

Master's Thesis in Computer Science

Department of Computer Science
Aalborg University, Denmark

June 2023

Abstract. In this report, we study symbolic approaches for efficiently finding ratio-optimal infinite cycles in doubly priced timed automata. As our main contribution, we present symbolic λ-deduction that reduces the problem to a single priced automaton and uses priced zones to incrementally improve the best found cycle. We prove that it converges on an optimal solution by proving its correctness and termination. Furthermore, we also present an algorithm that solves the problem through binary decision diagrams (BDDs) using the transition relation induced by the automaton. However, the BDD approach has wretched performance, but it does have theoretical value. In an experimental evaluation of practical models, we compare the various approaches to address their performance and show that symbolic λ-deduction outperforms the concrete alternative in some cases. We will also experimentally test the algorithm's symbolic strength of being completely unaffected by large clock space, where a concrete approach suffers an exponential slowdown.

Summary of Master's Thesis in Danish

I denne rapport undersøger vi symbolske teknikker til effektivt at finde forholdsoptimale uendelige cykler i dobbeltprismærkede tidsautomater. Dette problem finder relevans i den virkelige verden ved optimering af processer i realtidssystemer. Realtidssystemer har faste realtidsrestriktioner, der kræver, at bestemte handlinger skal ske indenfor specifikke tidsrammer. Det er dog ikke altid tilfredsstillende kun at garantere, at tidsrammerne bliver overholdt. Det er også vigtigt, at processen bliver udført effektivt. For cykliske processer, hvor der ikke er en sluttilstand, der skal opnås, kan effektiveten beskrives som et forhold mellem omkostninger og belønninger. Dette kan modelleres som en dobbeltprismærket tidsautomat. En optimal løsning er således en uendelig cyklus, der har lavest mulig omkostning per belønning. Problemet er bevist at være PSPACE-fuldstændigt.

Vores hovedbidrag er at have udviklet algoritmen symbolic λ-deduction, som udnytter prismærkede zoner til at repræsentere tidsrummet, og dermed leder efter cykler til trinvist at forbedre den hidtil bedste cyklus. Hovedprincippet i algoritmen er at omdanne den dobbelt prismærkede automat til en enkelt prismærkede automat, således at negative cykler i den enkelt prismærkede automat svarer til bedre (end den nuværende) cykler i den dobbelt prismærkede automat. Vi beviser dens korrekthed og endelighed, hvilket betyder, at denne trinvise forbedring konvergerer på en optimal løsning. Tilmed viser vi, hvordan den ændring en symbolsk cykel har på prisfunktionen for prismærkede zoner kan repræsenteres som en affin afbildning i form af en matrix, og hvordan dette kan bruges til at effektivt identificere negative symbolske cykler.

Vi udvikler også en anden symbolsk løsning, som udnytter binære beslutningsdiagrammer (BDD'er) til at konstruere den transitive lukning af overgangsrelation fra det diskrete tilstandsovergangssystem fra hjørnepunktsabstraktion (corner-point abstraction). Desværre viser denne algoritme sig at have elendig ydeevne.

Slutteligt laver vi en eksperimentel evaluering af vores to nyopfundne algoritmer, samt vores implementation af hjørnepunktsabstraktion fra et tidligere værk,

som vi kalder CP-MCR. Vi finder, at vores BDD metode slet ikke kan måle sig med de to andre. Tilmed ser vi, at CP-MCR er bedst på modeller, hvor der er et lille tidsrum, der skal ræsonneres over, hvorimod symbolic λ-deduction er upåvirket af størrelsen af konstanterne.

Acknowledgements

We would like to extend our sincere thanks to our supervisors, Kim Guldstrand Larsen and Álvaro Torralba, for excellent, engaged and tenacious supervision. It has really been a great pleasure working with you.

Pre-Specialisation

This report continues on our pre-specialisation project [1]. This report will be self-contained, hence, various sections throughout the report have been copied from the previous project. This includes the entirety of Sections 2 and 6 and also large portions of Sections 1, 3 and 5. We have allowed ourselves the liberty to make corrections to the copied sections.

1 Introduction

This introduction to our problem and model is copied from our pre-specialisation project since we continue with the same problem using the same model. Systems often have behaviour that requires deliberate timing of both the global time but also more complex timing of events intertwined between sub-processes. This class of systems is called *real-time systems*. They make decisions and handle events specifically by reacting to the timing of inputs or communications. These systems require both functional correctness in the sense that the logical ordering of events. But for a real-time system, functional correctness does not suffice, hard constraints on the timing between events must also be satisfied, e.g. an airbag in a vehicle must activate with very precise timing shortly after a collision has taken place.

Model-checking is a well-studied approach that has been extended to verify the correct timings of real-time systems. For instance, it can be used to verify whether a system can reach a state where communication has taken place correctly; or do synthesis of schedules optimising some objective.

One way to model real-time systems in the context of optimality is with *priced timed automata*, which was introduced in [2] and simultaneously (and independently) in [3], which has been used to model and verify various real-world problems [4, 5, 6, 7]. A further extension to double-priced timed automata was presented in [8]. This type of automaton facilitates a wide variety of practical real-time systems. For practical purposes, they can implement measures such as money, production, time, flow, energy, and consumption, and also support ratios between measures, e.g. energy/production.

Model-checking real-time systems is hard because of model-checking's inherent obstacle of state-space explosion, additionally, because continuous time makes the state-space infinite. There exists previous work combating these problems in the context of (double-)priced timed automata: discretisation of time values called corner-point abstraction [8], symbolic representation of states with so-called regions [9] or zones, which are used in the formal verification tools UPPAAL [10] and KRONOS [11], guided state space exploration (possibly symbolically) us-

ing on-the-fly branch-and-bound techniques to search only relevant parts of the state space [12], and finally, introducing maximal time values for clocks [2].

In this paper, we study the problem of finding ratio-optimal infinite cycles in cost-reward timed automata, which is proven to be PSPACE-complete by [8]. In this paper, we use the measure names cost and reward, which both can represent any of the aforementioned measures, and we focus on finding an infinite execution that optimises the limit ratio between the accumulated cost and reward.

Related work has previously studied ratio-optimal infinite cycles using various interesting approaches. In [8], they used corner-point abstraction to prove that ratio-optimal infinite runs are computable, with the complexity class of PSPACE-complete. Furthermore, discounting over time has been studied to model cost-optimal problems for infinite traces in [13]. Also, population-based methods have been used in [14] to approximate the optimal schedule. The problem has also been studied in doubly priced timed automata restricted to only 1 clock in [15]. They use strong time-abstracting bisimulation to construct 1-clock zone graphs and find the optimal cycle by using Lawler's algorithm[16] on the underlying doubly weighted graph. In [17] and [18], they conducted a complexity study on finding feasible cycles in the more general model called energy timed automata, which also supports negative rates and bounding constraints on the price. But with such extensions, the problem of finding a feasible cycle becomes undecidable already with 4 clocks [19].

1.1 Our Contributions

This report is an extension of our pre-specialisation paper [1], which had its own significant contributions: (i) it implemented a ground truth concrete algorithm that is based on the corner-point abstraction by [8], solving the induced minimal cycle ratio problem using Howard's algorithm; (ii) it studied symbolic representations and showed some significant theoretical barriers through counter-examples that arise in this setting; (iii) it proved that there is always a best concrete cycle (of a symbolic cycle) that only passes through the symbolic cycle once.

In this report, we contribute to the related work by presenting an efficient approach for finding ratio-optimal infinite executions in doubly-priced timed automata. As our main contribution, we present symbolic λ-deduction that uses priced zones to incrementally improve the best found cycle. We prove that it converges on an optimal solution by proving its correctness and termination. Additionally, we show how the change of the cost function of a symbolic state, from traversing a path (or cycle), can be represented as an affine transformation matrix, allowing us to efficiently determine whether a symbolic cycle is a better solution.

We also studied the application of binary decision diagrams to solve the problem. Here, we propose how to encode the discrete transition system of the automaton with its costs and rewards as a BDD. We find the optimal cycle by computing the closure of the transition relation.

Finally, we provide an experimental evaluation of our proposed algorithms: minimal-cost-ratio problem with Howard's and symbolic λ-deduction, omitting the BDD approach due to its wretched performance. We show that symbolic λ-deduction outperforms the MCR problem with Howard's on some problems. We will also test the symbolic strength of being completely unaffected by large clock values, and show that the concrete approach suffers an exponential slowdown here.

In the remainder of the paper, the structure is as follows. In Section 2, we present our model of cost-reward timed automata along with its semantics and define the ratio-optimal cycle problem. In Section 3.2, we show the ground truth algorithm from the pre-specialisation paper. In Section 4, we study BDDs to solve the problem. In section 5, we formally introduce the symbolic representations. In Section 6.1, we show the linear programming method for extracting the best concrete cycle from a symbolic cycle, which is from the pre-specialisation paper. In Section 7, we give our main contribution of symbolic λ-deduction through an algorithm, with a formal proof of correctness and ideas for optimisations. We provide an experimental evaluation of the ground truth algorithm and symbolic λ-deduction. We conclude the report in Section 9 and give some ideas for future work.

2 Cost-Reward Optimal Cycles

The vast majority of this section is copied from our pre-specialisation project for the purpose of self-containment of this report.

In this section, we formalise the underlying model, a timed automaton with cost and reward, and the semantics describing the transition system. Furthermore, we define the cost-reward ratio for an execution in the model, and we present the problem statement that the rest of the paper will concentrate on.

2.1 Cost-Reward Timed Automata

The definitions used in this section are based on the work of [20]. A clock *valuation* $u \in \mathbb{R}_{\geq 0}^{\mathbb{C}}$ over the set of clocks $\mathbb{C}$ is a function $u : \mathbb{C} \to \mathbb{R}_{\geq 0}$ assigning a value to each clock. For a delay $\delta \in \mathbb{R}_{\geq 0}$, we use the notation $u + \delta$ to denote the updated valuation, where $(u + \delta)(x) = u(x) + \delta$. Let $R \subseteq \mathbb{C}$ be a set of clocks to be reset then $u[R \mapsto 0]$ is the new valuation u' such that $u'(x) = 0$ if $x \in R$, otherwise, $u'(x) = u(x)$. Further, $\mathcal{B}(\mathbb{C})$ is the set of clock constraints over $\mathbb{C}$ obtained by conjunction over atomic constraints $x \bowtie n$ for $x \in \mathbb{C}, \bowtie \in \{\leq, =, \geq\}$, and $n \in \mathbb{N}$. Let $g \in \mathcal{B}(\mathbb{C})$ be such a constraint, then we write $u \models g$ when u satisfies the constraint g.

Definition 1. A Cost-Reward Timed Automaton (CRTA) over a set of clocks $\mathbb{C}$ is a tuple (L, ℓ_0, E, I, c, r), where L is the set of locations, ℓ_0 is the initial location, $E \subseteq L \times \mathcal{B}(\mathbb{C}) \times 2^{\mathbb{C}} \times L$ is the set of edges between locations, $I : L \to \mathcal{B}(\mathbb{C})$ assigns invariants to locations, and lastly $c : L \cup E \to \mathbb{Z}$ and $r : L \cup E \to \mathbb{N}_0$ are, respectively, cost and reward rates for locations and prices for edges. For edges, we write $\ell \xrightarrow{g,R} \ell'$ whenever $(\ell, g, R, \ell') \in E$, where g is called the guard of the edge and R is the set of clocks to be reset.

We point out an interesting subclass of CRTA where the reward corresponds to the elapsing of time. This can be done by setting the reward rate at each location to 1 and letting the transition rewards be 0.

***Example* 1** (Lawnmower) Consider the example CRTA shown in Figure 1. This will be our running example. This automaton models a lawnmower, which job

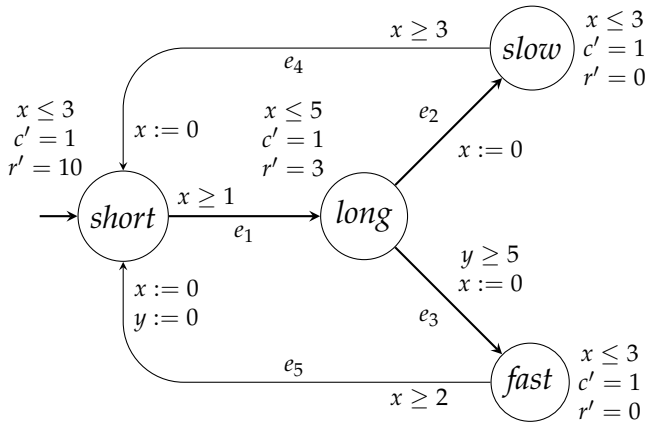


Fig. 1: CRTA of a lawnmower example. Resets are denoted by "$x := 0$", cost increments by "$c \mathrel{+}= n$" (or reward by "$r \mathrel{+}= n$"), the cost/reward rates of delaying in locations are denoted by "$c' = n$" (and "$r' = n$" for reward). The initial state is marked by a sourceless incoming edge.

is to tend a lawn by keeping it nicely short. It has to decide when the grass has become too long by transitioning to the *long* location. In the *long* location, it can wait an additional 2 time units, but receives less reward as the lawn is in a poorer state. Next, it has a choice of whether to mow fast or slow by going to their respective location. The slow approach takes 3 units of time, and the fast approach takes between 2 and 3 units of time. The reward models the quality of the lawn, and the cost models the time. When the grass is short, the quality of the lawn is high, and thus gives 10 reward each with each unit of time passing. The quality is worse when the grass is long, and therefore, the location gives a lesser reward of 3 per unit of time passing. With these measures, the lawn quality per time can be analysed. The robot must then perform the optimal cycle i.e. the one that yields the most lawn quality per cost, or reversely, least cost per lawn quality. One example of an infinite cycle is to delay 3 units of time in *short*, go to *long*, delay 2 in *long*, go to *fast*, delay 3 in *fast*, and complete the cycle by returning to *short*. This cycle accumulates 8 cost and 36 reward, and thus it has a ratio of $\frac{8}{36} \approx 0.22$. □

2.2 Semantics and Cost-Reward Optimal Cycles

The semantics of a CRTA is given by an underlying cost-reward weighted transition system. Intuitively, the cost and reward of both delay and edges in the automaton constitute the cost-reward weightings of the transitions in the transition system. When using the term *concrete* we refer to items in the underlying transition system.

Definition 2. A Cost-Reward Weighted Transition System (CR-WTS) is a tuple $\mathcal{T} = (S, s_0, T, \mathsf{cost}, \mathsf{reward})$, where S is the set of concrete states, s_0 is the initial state, $T \subseteq S \times S$ is the transition function, and $\mathsf{cost}, \mathsf{reward} : T \to \mathbb{R}$ assign cost and reward to transitions, respectively. We write $s \to s'$ whenever $(s, s') \in T$, and also $s \xrightarrow{c,r} s'$ whenever $(s, s') \in T$, $c = \mathsf{cost}(s, s')$ and $r = \mathsf{reward}(s, s')$.

The semantics of an automaton $\mathcal{A} = (L, \ell_0, E, I, c, r)$ over a set of clocks $\mathbb{C}$ is given by the underlying CR-WTS $\mathcal{T} = (L \times \mathbb{R}^{\mathbb{C}}_{\geq 0}, (\ell_0, \mathbf{0}), T, \mathsf{cost}, \mathsf{reward})$, where $\mathbf{0}$ assigns 0 to all clocks, with the delay and edge transitions

$$(\ell, u) \xrightarrow{\delta \cdot \mathrm{c}(\ell),\ \delta \cdot \mathrm{r}(\ell)} (\ell, u + \delta) \text{ if } \forall 0 \leq \delta' \leq \delta.\ u + \delta' \models I(\ell)$$

and

$$(\ell, u) \xrightarrow{c(e),\ r(e)} (\ell', u[R \mapsto 0]) \text{ if } e = \ell \xrightarrow{g,R} \ell', e \in E, u \models g \text{ and } u[R \mapsto 0] \models I(\ell').$$

Let $\pi = s_0 \to s_1 \to \cdots \to s_n$ be a *finite execution* in a CR-WTS $(S, s_0, T, \mathsf{cost}, \mathsf{reward})$ consisting of concrete states. The cost and reward functions extend to finite executions straightforwardly

$$\mathsf{Cost}(\pi) = \sum_{i=1}^{n} \mathsf{cost}(s_{i-1}, s_i) \quad \text{and} \quad \mathsf{Reward}(\pi) = \sum_{i=1}^{n} \mathsf{reward}(s_{i-1}, s_i).$$

We define the cost-reward *ratio* for a finite execution, where $\mathsf{Reward}(\pi) \neq 0$, as

$$\mathsf{Ratio}(\pi) = \frac{\mathsf{Cost}(\pi)}{\mathsf{Reward}(\pi)}.$$

We now consider the case of an *infinite execution*, Π. Let Π_n denote the finite prefix execution of Π with length n. The ratio of the infinite execution Π is then defined by

$$\mathsf{Ratio}(\Pi) = \lim_{n \to +\infty} \mathsf{Ratio}(\Pi_n),$$

provided this limit exists. The *optimal* ratio for a CR-WTS $\mathcal{A}$ is denoted by θ^*

$$\theta^* = \inf \{ \, \mathsf{Ratio}(\Pi) \mid \Pi \text{ is an infinite execution in } \mathcal{A} \, \} \, .$$

An infinite execution Π is said to be *ratio-optimal* if $\mathsf{Ratio}(\Pi) = \theta^*$.

***Example* 2** (Lawnmower - Semantics) We now continue with the example lawnmower automaton $\mathcal{A} = (L, \ell_0, E, I, c, r)$ over the clocks $\mathbb{C}$ from Figure 1. For concrete states in the underlying transition system, we apply the notational sugar : $(\ell, n_1, n_2, \ldots, n_{|\mathbb{C}|}) = (\ell, u)$ if $n_i = u(x_i)$ for all $1 \leq i \leq |\mathbb{C}|$. For example, the initial state is written as $(\mathit{short}, 0, 0)$. The infinite cycle from before is formulated in the semantics as the infinite occurrence of

$$C = (\mathit{short}, 0, 0) \xrightarrow{3,30} (\mathit{short}, 3, 3) \xrightarrow{0,0} (\mathit{long}, 3, 3) \xrightarrow{2,6} (\mathit{long}, 5, 5) \xrightarrow{0,0} (\mathit{fast}, 0, 5) \xrightarrow{3,0} (\mathit{fast}, 3, 8).$$

(with a return edge labelled $0, 0$ from $(\mathit{fast}, 3, 8)$ back to $(\mathit{short}, 0, 0)$)

The ratio of C is given by $\mathsf{Ratio}(C) = \frac{8}{36} \approx 0.22$. The ratio-optimal infinite cycle of $\mathcal{A}$ is to alternate between *fast* and *slow*. This makes sense as the robot can avoid stalling in the less prosperous *long* location by immediately transitioning to *slow*, and thus quickly returning to *short*, where the reward is highest, and the next round it can transition to *fast* without having to stall in *long*. Formally, the ratio-optimal cycle C^* is

$$(\mathit{short}, 0, 0) \xrightarrow{3,30} (\mathit{short}, 3, 3) \xrightarrow{0,0} (\mathit{long}, 3, 3) \xrightarrow{0,0} (\mathit{slow}, 0, 3) \xrightarrow{3,0} (\mathit{slow}, 3, 6)$$

$$\xrightarrow{0,0} (\mathit{short}, 0, 6) \xrightarrow{3,30} (\mathit{short}, 3, 9) \xrightarrow{0,0} (\mathit{long}, 3, 9) \xrightarrow{0,0} (\mathit{fast}, 0, 9) \xrightarrow{2,0} (\mathit{fast}, 2, 11)$$

(with a return edge labelled $0, 0$ from $(\mathit{fast}, 2, 11)$ back to $(\mathit{short}, 0, 0)$)

with $\mathsf{Ratio}(C^*) = \frac{11}{60} = \theta^* \approx 0.18$, which is better than the ratio of 0.19 from the cycle utilising only the fast option, or, for the sake of completeness, the cycle choosing only the slow option, which has a ratio of $\frac{6}{30} = 0.2$. □

2.3 Restrictions

In this paper, we make the restriction that automata must not contain any so-called Zeno-cycles, i.e. any infinite execution with an infinite number of discrete edges must also use an infinite amount of time. Additionally, we also only consider automata that are *reward-diverging* (detailed in [8]). Formally, let $\mathcal{A}$ be an automaton, then it is reward-diverging if all infinite executions that are time-divergent also result in an infinite accumulated reward. As a result of these restrictions, we also disconsider infinite executions that are either non-reward diverging or a Zeno-execution in the CR-WTS.

2.4 Boundedness of Automata

In this work, we apply the notion of bounded time on automata by [2]. A timed automaton $\mathcal{A}$ is bounded by M if all of the constraints in the guards and invariants of $\mathcal{A}$ contain no constants greater than M. We can then transform an automaton bounded by M into a timed-bisimilar automaton, where no clock's value exceeds the maximal bound $M+2$, and instead setting clock valuations to $M+1$ when they hit $M+2$. This means that when the value of a clock would normally have increased above $M+2$, it is instead kept between $M+1$ and $M+2$, which is equivalent since all of the constraints cannot differentiate between these values strictly greater than M.

***Example* 3** (Boundedness) Consider again the automaton in Figure 1. The maximum constant any clock is compared to in this example is 5. When cycling along the locations *short*, *long* and *slow*, the y-clock is not reset, and would therefore grow ad infinitum, however, by bounding the automaton with M, we get a cycle like

$$(short, 0, 6) \to (short, 1, 7) \xrightarrow[y:=6]{} (short, 1, 6) \to (short, 2, 7) \xrightarrow[y:=6]{} (short, 2, 6) \to \cdots$$

□

2.5 Problem Statement

Given reward-diverging CRTA with no Zeno-cycles, compute in the CR-WTS a ratio-optimal infinite execution π, if it exists.

3 Concrete Minimum Cycle Ratio

In this section, we describe a method for solving the cost-reward optimal cycle problem by analysing the concrete states of cost-reward timed automata. We apply the corner-point abstraction technique to reduce the number of concrete states to a finite amount. Sections 3.1 and 3.2 are copied from our pre-specialisation project for self-containment.

3.1 Corner-Point abstraction

Corner-point abstraction was introduced by Bouyer, Brinksma, and Larsen in [8]; this section is based on their work. The idea of corner-point abstraction is to realise that the clock valuation space can be discretised by only considering points that are exactly integer-valued. This can be done while still preserving optimality, i.e. at least one ratio-optimal infinite execution still exists. With the additional boundedness assumption of timed automata, this means that the clock valuation space is finite.

Corner-point abstraction is based on regions [9]. Regions are a way to partition the clock space into sets of valuations that are untimed bisimilar. Regions partition the clock space to reason about fractional values for clocks, however, since we only consider non-strict guards, we need not concern ourselves with the fractional values of the clocks. Therefore, we will simplify the corner-point abstraction here, to only care about integer-valued valuations, and therefore discard the region information.

In the corner-point abstraction, we restrict the clock-space to only integer-points. Fortunately, any execution in the corner-point abstraction $\mathcal{A}_{cp}$ can be reconstructed in the CRTA $\mathcal{A}$ with the same cost-reward ratio [8]. Wielding the

corner-point abstraction, we will constrain the solution space to only integer-valued cycles.

Definition 3. A *discrete cycle* is a concrete cycle which uses only integer valued delays.

A discrete cycle only uses integer valuations because all clock resets set the clock value to an integer, and thus afterwards integer delays ensure integer valuations.

We say a cycle is simple if no state is visited twice; otherwise, it is complex.

Theorem 1. *A complex concrete cycle in a CRTA $\mathcal{A}$ cannot have a better ratio than all of the simple cycles it is composed of.*

Proof. Let Γ be a complex cycle. We partition Γ into a set of simple cycles $\sigma_1, \sigma_2, \ldots, \sigma_n$, and let $\frac{c_1}{r_1}, \frac{c_2}{r_2}, \ldots, \frac{c_n}{r_n}$ be their ratios. Assume w.l.o.g. that $\frac{c_1}{r_1} \leq \frac{c_2}{r_2} \leq \cdots \leq \frac{c_n}{r_n}$, we then show by induction on n, that this implies $\frac{c_1}{r_1} \leq \frac{c_1+c_2+\cdots+c_n}{r_1+r_2+\cdots+r_n} \leq \frac{c_n}{r_n}$.
Basecase (n=1): Trivially, $\frac{c_1}{r_1} \leq \frac{c_1}{r_1}$.
Induction step: Assume that $\frac{c_1}{r_1} \leq \frac{c_1+c_2+\cdots+c_n}{r_1+r_2+\cdots+r_n}$. We then have that $\frac{c_1+c_2+\cdots+c_n}{r_1+r_2+\cdots+r_n} \leq \frac{c_n}{r_n} \leq \frac{c_{n+1}}{r_{n+1}}$. Recall the property that $\frac{a}{b} \leq \frac{c}{d} \implies \frac{a}{b} \leq \frac{a+c}{b+d} \leq \frac{c}{d}$. By applying this, we get that $\frac{c_1}{r_1} \leq \frac{c_1+c_2+\cdots+c_n+c_{n+1}}{r_1+r_2+\cdots+r_n+r_{n+1}} \leq \frac{c_{n+1}}{r_{n+1}}$.

Since $\mathsf{Ratio}(\Gamma) = \frac{c_1+c_2+\cdots+c_n}{r_1+r_2+\cdots+r_n}$, this proves that the lowest ratio simple cycle is always at least as good. □

From Theorem 1, it is evident that if there exists a cycle then there is a simple cycle that is ratio-optimal. Therefore, we will constrain the solution space to only simple discrete cycles.

3.2 Reduction to Minimum Cycle Ratio

In this section, we will show how to use corner-point abstraction to reduce the problem of finding a cost-reward ratio-optimal infinite cycle in a CRTA to the Minimum Cycle Ratio (MCR) problem.

The Minimum Cycle Ratio problem states that given a doubly weighted graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{C}, \mathcal{R})$, where $\mathcal{V}$ is a finite set of vertices, $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ a set of edges, $\mathcal{C} : \mathcal{E} \to \mathbb{R}$ a cost function, and $\mathcal{R} : \mathcal{E} \to \mathbb{R}_{\geq 0}$ a reward function, find a cycle in $\mathcal{G}$ where the ratio of accumulated cost to accumulated reward is minimal.

The reduction to MCR is achieved by using the semantics of corner-point abstraction and adding cost and reward weights. For a CRTA $\mathcal{A} = (L, \ell_0, E, I, \mathrm{c}, \mathrm{r})$, the doubly-weighted graph $\mathcal{G}_{\mathcal{A}_{cp}}$ is constructed by expanding the states according to the semantics. In this way, only the reachable part of the state space is constructed. The vertices of the graph are tuples of a location and a corner point, i.e. $\mathcal{V} \subseteq L \times \mathbb{N}^{\mathbb{C}}$. To construct the graph, we apply the following procedure until a fixed point is reached: for all $(\ell, u) \in \mathcal{V}$ add edges

$$(\ell, u) \xrightarrow{\mathrm{c}(e), \mathrm{r}(e)} (\ell', u[R \mapsto 0]) \quad \text{if } e = (\ell, g, R, \ell') \in E,$$

representing a discrete edge in the automaton; and

$$(\ell, r, \alpha) \xrightarrow{\mathrm{c}(\ell), \mathrm{r}(\ell)} (\ell, r, \alpha \oplus_M 1)$$

representing a unit delay, moving from one corner point to its unit time successor. For an automaton bounded by M, we define

$$(u \oplus_M 1)(x) = \begin{cases} u(x) + 1 & \text{if } u(x) \leq M \\ u(x) & \text{otherwise} \end{cases}$$

as a bounded delay, limiting the value of a clock to at most $M + 1$. This compounds the delay and boundedness reset explained in Section 2.4 into a single edge.

Naturally, a solution to the MCR problem of $\mathcal{G}_{\mathcal{A}_{cp}}$ always also constitutes a solution to the infinite execution in the corner-point abstraction of $\mathcal{A}_{cp}$, because the graph directly models its semantics.

In [21], they surveyed the numerous existing algorithms that solve the MCR problem. In this paper, we will use the extended version of Howard's algorithm[22, 23] mentioned in [21].

4 Binary Decision Diagrams

Binary decision diagrams (BDDs) has been shown to be an efficient and compact data structure with many practical applications. In this section, we describe how the discretised cost-reward weight transition system can be encoded as a BDD and used to solve the cost-reward optimal cycle problem. We achieve this by constructing a BDD representing the transition relation s.t. a truth valuation constitutes a transition from one state to another with a given cost and reward. Then, we compute the transitive closure of this relation and then analyse it for reflexive transitions, i.e. cycles.

In this section, we will use $\stackrel{\text{def}}{=}$ to define operations in order to avoid confusion with $=$ used as the equality relation between BDDs.

4.1 Encoding States as a BDD

For a CRTA (L, ℓ_0, E, I, c, r) with concrete states $(\ell, u) \in L \times \mathbb{N}_0^{\mathbb{C}}$, we encode discrete states into a BDD by determining some binary encoding for each location and clock valuation. We create a vector of BDD variables to encode locations $\overline{\ell} = (\overline{\ell}_0, \overline{\ell}_1, \ldots, \overline{\ell}_{k-1})$, where $k = \lceil \log_2 |L| \rceil$, and clock values $\overline{x} = (\overline{x}_0, \overline{x}_1, \ldots, \overline{x}_{n-1})$ for each clock $x \in \mathbb{C}$, where $n = \lceil \log_2 C_{\max} \rceil$ and $C_{\max}$ is the maximum constant that any clock is compared with (see Section 2.4 on boundedness). To also reason about the cost and reward, we encode these as $\overline{c}$ and $\overline{r}$, where the number of bits needed is found by the upper bound $\lceil \log_2 \left(((C_{\max})^{|\mathbb{C}|} \cdot |L|) \cdot \max_{a \in E \cup L} c(a) \right) \rceil$ for the cost and similarly for the reward. Here $(C_{\max})^{|\mathbb{C}|} \cdot |L|$ is an upper bound for the number of concrete states, and thus the length of the longest simple cycle, and $\max_{a \in E \cup L} c(a)$ is the cost of the most expensive transition. To describe the transitions, we will use non-primed variables for the *from* state and primed variables for the *to* state, i.e. a transition between two states can be represented using the

variables $\bar{l}, \bar{x}, \bar{l}', \bar{x}', \bar{c}, \bar{r}$ (for one clock x). We use $\bar{s}$ as shorthand for the variables for location and clocks.

4.2 BDD operations

There are a couple of BDD operations we will need to reason about clock values and costs/rewards. First, we need to model the constraints of the form $x \leq n$ and $x \geq m$, i.e. compare a clock with some constant. Secondly, we define the bounded increment operation to model unit delays in a bounded automaton and finally the increase in cost and reward $c + k$.

For the $x \leq n$ constraint, we create a BDD recursively by

$$LEQ(\bar{x}, n, i) \stackrel{\text{def}}{=} \begin{cases} \text{true} & \text{if } i < 0 \\ \neg \bar{x}_i \vee LEQ(\bar{x}, n, i-1) & \text{if } n_i = 1 \\ \neg \bar{x}_i \wedge LEQ(\bar{x}, n, i-1) & \text{if } n_i = 0 \end{cases},$$

where n_i is the i'th bit in the binary representation of n. It is then constructed by $LEQ(\bar{x}, n, |\bar{x}|)$. Other comparison relations can be created similarly.

We also require operations for addition and multiplication of BDD variables for defining bounded increment and computing the cost-reward ratio. This is achieved by operations similar to the adding and array multiplication operations from [24], which we will not reiterate here. The bounded delay—i.e. incrementing a clock value if it is less than the constant M, otherwise doing nothing (see Section 2.4)—

$$DELAY(\bar{x}, \bar{x}', M) \stackrel{\text{def}}{=} (GEQ(\bar{x}, M+1) \wedge \bar{x} = \bar{x}') \vee (\neg GEQ(\bar{x}, M+1) \wedge \bar{x}' = \bar{x} + 1).$$

4.3 Computing the Transition Relation Closure

We now show how to define the transition relation closure as a BDD, that tells which states can be reached in one or more steps and at what cost and reward. Importantly, we want to preclude 0-step transitions, as these do not constitute a valid cycle.

Although we will want to exclude 0-step transitions in the closure, we will need 0-step transitions later when computing the closure. We define it as

$$TR^0(\overline{s}, \overline{s}', \overline{c}, \overline{r}) \stackrel{\text{def}}{=} \overline{s} = \overline{s}' \wedge \overline{c} = 0 \wedge \overline{r} = 0,$$

as the identity relation, that stays in the same state and costs nothing and yields no reward. For the 1-step transition relation, we encode the possible transitions in the discrete semantics, namely unit delay

$$TR^1_{delay}(\overline{s}, \overline{s}', \overline{c}, \overline{r}) \stackrel{\text{def}}{=} \bigvee_{(\ell, g, R, \ell') \in E} \overline{\ell} = \ell \wedge \overline{\ell}' = \ell' \wedge (\bigwedge_{x \in \mathbb{C}} \overline{x} \models g \wedge ((x \in R \wedge \overline{x}' = 0) \vee (x \notin R \wedge \overline{x}' = \overline{x}))) \wedge \overline{c} = \mathrm{c}(e) \wedge \overline{r} = \mathrm{r}(e),$$

and discrete edge

$$TR^1_{edge}(\overline{s}, \overline{s}', \overline{c}, \overline{r}) \stackrel{\text{def}}{=} \bigvee_{\ell \in L} \overline{\ell} = \ell \wedge \overline{\ell}' = \ell \wedge (\bigwedge_{x \in \mathbb{C}} DELAY(\overline{x}, \overline{x}', c_{max}(x)) \wedge \overline{x}' \models I(\ell)) \wedge \overline{c} = \mathrm{c}(\ell) \wedge \overline{r} = \mathrm{r}(\ell).$$

Finally, the combination of these defines the 1-step transition relation

$$TR^1(\overline{s}, \overline{s}', \overline{c}, \overline{r}) \stackrel{\text{def}}{=} TR^1_{delay}(\overline{s}, \overline{s}', \overline{c}, \overline{r}) \vee TR^1_{edge}(\overline{s}, \overline{s}', \overline{c}, \overline{r}).$$

In order to compute the closure of the transition relation, we will need to concatenate two transition relations describing a double step, i.e. doing a step in the first and then a step in the second. We define the concatenation of two transition relations as

$$(TR \circ TR')(\overline{s}, \overline{s}', \overline{c}, \overline{r}) \stackrel{\text{def}}{=} \exists \overline{s}'', \overline{c}', \overline{c}'', \overline{r}', \overline{r}''.\ TR(\overline{s}, \overline{s}'', \overline{c}', \overline{r}') \wedge TR'(\overline{s}'', \overline{s}', \overline{c}'', \overline{r}'') \wedge \overline{c} = \overline{c}' + \overline{c}'' \wedge \overline{r} = \overline{r}' + \overline{r}''.$$

Notice that the existential quantifier removes the variables $\overline{s}'', \overline{c}', \overline{c}'', \overline{r}'$ and $\overline{r}''$, thus these are only present in the intermediate computation. We can then define the

compounding operation, λ, which computes the transition relation for 1 to $2n$ given the transition relation for 1 to n steps. This is given by

$$TR^{1-2n} \stackrel{\text{def}}{=} \lambda(TR^{1-n}) = TR^{1-n} \circ (TR^{1-n} \vee TR^0)$$

We can then compute the fixed point of λ from the initial TR^1 relation. We denote by $\lambda^*(TR)$ the fixed point of λ when applied to some transition relation TR, and additionally $TR^* = \lambda^*(TR^1)$. This will happen when the cost and reward maximum bounds are reached. Observe that the number of steps considered is doubled at each application. However, we may instead use the upper bound on the maximum length of a cycle, i.e. $k = (C_{\max})^{|\mathbb{C}|} \cdot |L|$, and only calculate TR^{1-k}, where $TR^* \supseteq TR^{1-k} \subseteq \lambda^{\log_2 k}(TR^1)$, which will include all necessary cycles.

4.4 Finding Optimal Cycles

In this section, we show how to construct a BDD that contains all reachable optimal simple cycles. The first step is to determine all of the concrete states that are reachable from the initial state. This step is straightforward using the transition relation closure

$$\mathit{Reach}(\bar{s}) \stackrel{\text{def}}{=} \exists \bar{c}, \bar{r}.\, TR^*(\bar{0}, \bar{s}, \bar{c}, \bar{r}),$$

where $\bar{0}$ is the encoding of the state with the initial location of the automaton and all clocks having the valuation 0.

We can now define the DBB representing the reachable cycles and the cost and reward of traversing the cycle as

$$\mathit{RCycle}(\bar{s}, \bar{c}, \bar{r}) \stackrel{\text{def}}{=} TR^*(\bar{s}, \bar{s}', \bar{c}, \bar{r}) \wedge \mathit{Reach}(\bar{s}) \wedge \bar{s} = \bar{s}'.$$

Informally, this finds the paths in the transition relation that begin and end in the same discrete state and are reachable from the initial state. Recall that TR^* contains no 0-step transitions, thus these are all actual cycles.

The final step is to find the optimal cycles among all of the reachable cycles. We need to pick the cycles which have the minimum cost-reward ratios. To get

around the fractional numbers of the ratios when comparing two cycles, we observe that $\frac{\overline{c}}{\overline{r}} \leq \frac{\overline{c}'}{\overline{r}'} \iff \overline{c} \cdot \overline{r}' \leq \overline{c}' \cdot \overline{r}$. The BDD containing the optimal cycles is then

$$BCycle(\overline{s}, \overline{c}, \overline{r}) \stackrel{\text{def}}{=} RCycle(\overline{s}, \overline{c}, \overline{r}) \wedge (\forall \overline{s}', \overline{c}', \overline{r}'.\ RCycle(\overline{s}', \overline{c}', \overline{r}') \implies \overline{c} \cdot \overline{r}' \leq \overline{c}' \cdot \overline{r}).$$

Informally, it contains all reachable cycles where the cost-reward ratios of all other reachable cycles are no better. Note, that this BDD contains all optimal simple cycles, but also some optimal complex cycles. The next section describes how we extract the optimal simple cycles from the BDD.

4.5 Extracting Optimal Simple Cycles

In the previous section, we defined the BDD $BCycle(\overline{s}, \overline{c}, \overline{r})$ that describes cycles from $\overline{s}$ to itself with optimal cost-reward ratios. In this section, we look into how to extract an optimal cycle from this BDD. To this end, we create a new 1-step transition relation TR^{opt}, which only contains transitions that are used in any optimal cycle

$$TR^{\text{opt}}(\overline{s}, \overline{s}', \overline{c}, \overline{r}) = \exists \overline{c}', \overline{c}'', \overline{r}', \overline{r}''.\ TR^1(\overline{s}, \overline{s}', \overline{c}, \overline{r}) \wedge TR^*(\overline{s}', \overline{s}, \overline{c}', \overline{r}') \wedge BCycle(\overline{s}, \overline{c}'', \overline{r}'') \wedge \overline{c} + \overline{c}' = \overline{c}'' \wedge \overline{r} + \overline{r}' = \overline{r}''.$$

It contains the 1-step transitions from $\overline{s}$ to $\overline{s}'$ where it is possible to go back to $\overline{s}$ again with exactly the same cost and reward as an optimal cycle in $\overline{s}$.

Theorem 2. *Let $\mathcal{T} = (S, s_0, T, \mathsf{cost}, \mathsf{reward})$ be a Cost-Reward Weighted Transition System, with an optimal cost-reward ratio of θ, and let $T^\theta \subseteq T$ be the transitions which are part of an optimal cycle. All cycles that use only transitions of T^θ are optimal.*

Proof. We do a proof by contradiction. Assume that there exists a cycle $\pi = s_0 \xrightarrow{c_0, r_0} \cdots \xrightarrow{c_n, r_n} s_0$ s.t. $\mathsf{Ratio}(\pi) = \frac{\mathsf{Cost}(\pi)}{\mathsf{Reward}(\pi)} \neq \theta$ and all transitions $\xrightarrow{c_i, r_i} \in T^\theta$ for all $0 \leq i \leq n$. Clearly, if $\mathsf{Ratio}(\pi) < \theta$, it is a contradiction, as then θ is not the optimal cycle ratio. Thus, we study if $\mathsf{Ratio}(\pi) > \theta$. By the assumption that all

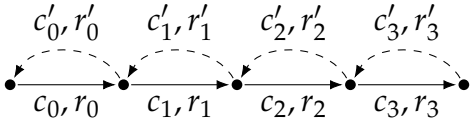
transitions in π are part of some optimal cycle, we observe that

$$\frac{c_i + c_i'}{r_i + r_i'} = \theta \quad \text{for all } 0 \leq i \leq n,$$

where c_i' and r_i' are the cost and reward of the rest of an optimal cycle that the transition $\xrightarrow{c_i, r_i}$ is part of. We can then rewrite this expression to obtain

$$c_i + c_i' = \theta r_i + \theta r_i' \iff c_i - \theta r_i = -(c_i' - \theta r_i').$$

We now construct a new cycle π' by replacing each transition $s_i \xrightarrow{c_i, r_i} s_{i+1}$ with the sequence of transitions that constitute the rest of an optimal cycle that the transition appears in, i.e. $s_{i+1} \overset{c_i', r_i'}{\rightsquigarrow} s_i$.



We then show that if $\mathsf{Ratio}(\pi) > \theta$ then $\mathsf{Ratio}(\pi') < \theta$. First, we have that

$$\frac{c_0 + \cdots + c_n}{r_0 + \cdots + r_n} > \theta,$$

which we can be rewritten, similarly as before, to

$$(c_0 - \theta r_0) + \cdots + (c_n - \theta r_n) > \theta.$$

We can then substitute $c_i - \theta r_i$ with $-(c_i' - \theta r_i')$

$$-(c_0' - \theta r_0') - \cdots - (c_n' - \theta r_n') > \theta.$$

Then we multiply both sides by -1, flipping the inequality, and rewrite to

$$\frac{c_0' + \cdots + c_n'}{r_0' + \cdots r_n'} < \theta.$$

Which is of course the ratio of π', thus again creating a contradiction, since π' would have a ratio lower than the optimal. □

Finding an optimal cycle can now be done by starting with an arbitrary transition in $TR^{\text{opt}}(\bar{s}, \bar{s}', \bar{c}, \bar{r})$ (i.e. a minterm) and continue following connected transitions until some state is visited again, at which point a cycle is found. These transitions between states are then decoded which results in an optimal cost-reward concrete cycle.

5 Symbolic Representation

Sections 5.1, 5.3 and the counter-examples in 5.2 are copied from our pre-specialisation project for self-containment of this report.

In this section, we will describe the symbolic structures, zones and priced zones, which have proved to be efficient in symbolic reachability analysis for timed automata [25]. In Section 5.2, we summarise complications regarding the discovery of cost-reward optimal cycles in the symbolic state space. The problem of extracting the best concrete cycle from a symbolic cycle will be discussed in Section 6.

5.1 Zones and Symbolic States

A *zone* is a symbolic structure abstracting over a subset of the clock space [10, 11]. Zones is an extension of the idea of regions by considering convex unions of clock regions. A zone is defined by a conjunction of clock constraints, which can either be upper or lower bounds on clocks or bounds on differences between two clocks. For a Cost-Reward Timed Automata (CRTA) over a set of clocks $\mathbb{C}$, a zone is a convex set in the $|\mathbb{C}|$-dimensional Euclidean space. In Figure 2 a diagram of a zone is shown.

We require four operations on zones to model the semantics of a CRTA. The first operation is the *up* operation, $Z^{\uparrow} = \{ u + d \mid u \in Z, d \in \mathbb{R}_{\geq 0} \}$, which removes the upper bounds of the zone Z. Similarly, we have the *down* operation, $Z^{\downarrow} = \{ u - d \mid u \in Z, d \in \mathbb{R}_{\geq 0}, \forall x \in \mathbb{C}.\, u(x) \geq d \}$, which sets the lower

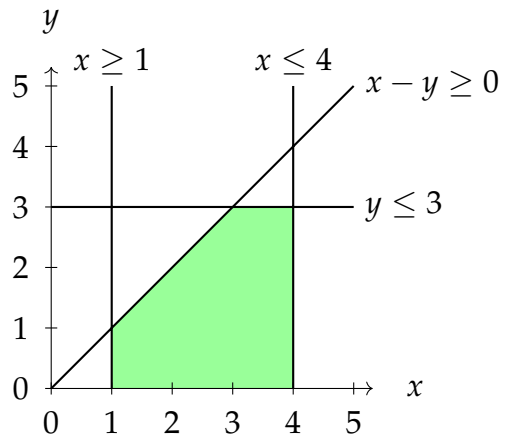


Fig. 2: Diagram of a zone induced by 4 clock constraints. The zone is the green area, which is the clock valuation that satisfy the constraints.

bounds of a zone to 0. The third operation is the project operation, which we denote by $\{R\}Z$, where R is a set of clocks to project the zone onto, $\{R\}Z = \{\, u[R \mapsto 0] \mid u \in Z \,\}$, where every clock in R set to 0. Finally, the reverse projection, $\{R\}^{-1}Z = \{\, u' \in \mathbb{R}^{\mathbb{C}}_{\geq 0} \mid \exists u \in Z \; \forall x \notin R.\; u(x) = u'(x) \,\}$, frees the clocks in R to have any value. Note that zones are closed under these operations.

For a cost-reward timed automaton $\mathcal{A} = (L, \ell_0, E, I, \mathrm{c}, \mathrm{r})$, the symbolic state-space is $S = L \times \mathfrak{Z}$, where $\mathfrak{Z}$ is the set of all zones over the clocks.

In the semantics of a cost-reward timed automaton, we use two types of transitions for discrete edges and delays. We use dual operations on symbolic states called Post_e and Post_ϵ, which together give the symbolic transitions, given by

$$\mathsf{Post}_e((\ell, Z)) = (\ell', (\{R\}(Z \wedge g)) \wedge I(\ell')), \text{ where } e = (\ell, g, r, \ell')$$

and

$$\mathsf{Post}_\epsilon((\ell, Z)) = (\ell, Z^{\uparrow} \wedge I(\ell)).$$

We say that a concrete state $s = (\ell, u)$ is in a symbolic state $S = (\ell, Z)$, denoted by $s \in S$, iff $u \in Z$. We write $S \xrightarrow{\epsilon} S'$ whenever $S' \in \mathsf{Post}_\epsilon(S)$ and $S \xrightarrow{e} S'$ whenever $S' \in \mathsf{Post}_e(S)$. A non-annotated transition, $S \to S'$, is defined if either Post_e or

Post_ϵ leads to S'. A symbolic path $\Pi = S_0 \xrightarrow{\epsilon} S_0' \xrightarrow{e_0} S_1 \xrightarrow{\epsilon} \cdots \xrightarrow{e_{i-1}} S_i \xrightarrow{\epsilon} S_i' \xrightarrow{e_i} \cdots$ is a possibly infinite sequence of symbolic transitions, alternating between Post_ϵ and Post_e transitions. When the states of a symbolic path are distinct, we call it *simple*. We use the notation $S_0 \rightsquigarrow S_n$ whenever there exists a symbolic path from S_0 to S_n.

Definition 4 (Symbolic Cycle). A symbolic path Π is a *symbolic cycle* if there exists a concrete cycle $\pi \in \Pi$.

Similarly to a simple path, a simple cycle is a cycle wherein each state is visited at most once per revolution, except the initial state, which is visited exactly twice.

We say that a symbolic transition $S \to S'$ is forward stable if for all concrete states $s' \in S'$ there exists $s \in S$ s.t. $s \to s'$, and it is backwards stable if for all $s \in S$ there exists $s' \in S'$ s.t. $s \to s'$. We say that a symbolic path is *stable* if all of its transitions are both forward and backwards stable. Symbolic transitions are created by the Post-operations and are therefore already forward stable. To backwards stabilise a symbolic transition, we introduce the Pre-operations

$$\mathsf{Pre}_\epsilon((\ell, Z)) = (\ell, Z^{\downarrow} \wedge I(\ell)),$$

and

$$\mathsf{Pre}_e((\ell, Z)) = (\ell, (\{R\}^{-1}Z) \wedge g \wedge I(\ell)), \text{ where } e = (\ell, g, R, \ell')$$

These operations compute the pre-states of delay and discrete edge transitions. A forward stable transition $S \xrightarrow{\alpha} S'$, is stabilised by $(S \cap \mathsf{Pre}_\alpha(S')) \xrightarrow{\alpha} S'$. A forward stable symbolic path can be stabilised by stabilising each transition until a fixed point is reached. Each stabilisation step on a transition always yields a new state that is a subset of the previous, therefore, at some point, the stabilisation will come to a fixed point.

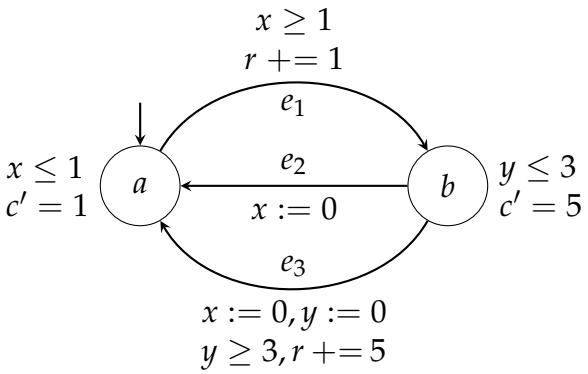
5.2 Pitfalls of Symbolic Representation

In the pre-specialisation project, we proposed a simple symbolic on-the-fly algorithm to find symbolic cycles in a Cost-Reward Timed Automaton. Starting in the

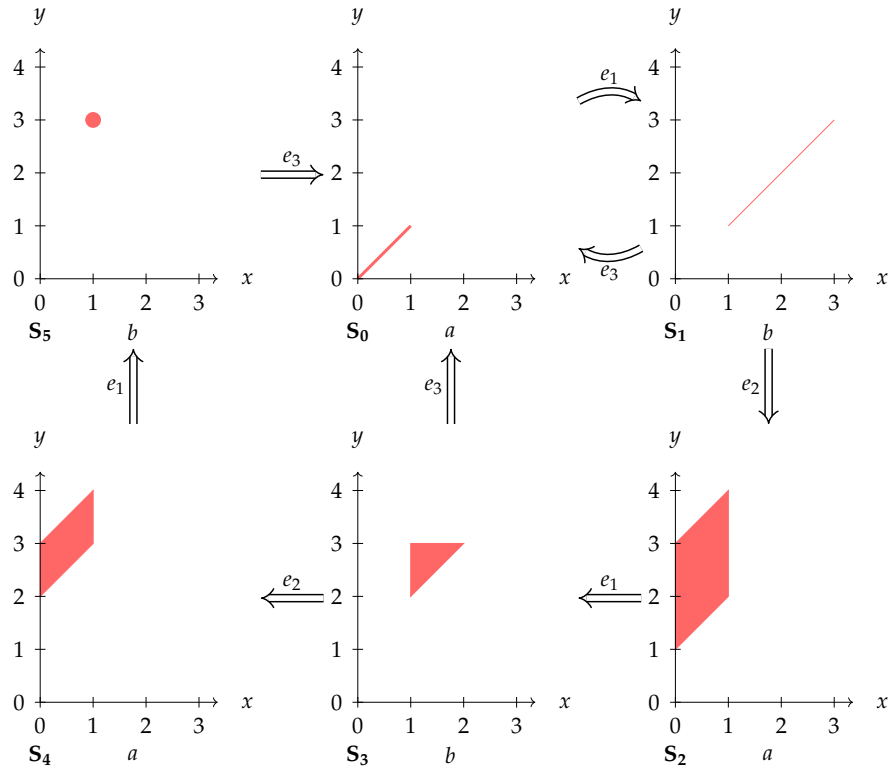
initial state, it explores all symbolic paths in a depth-first manner, as per the post operations of zones, until a non-specified stopping criterion is reached.

One stopping criterion, we looked into, was to stop the search of the branch when we reached a symbolic state that was a zone-wise subset of an already visited symbolic state, in the current trace. We say that a symbolic state $S = (\ell, Z)$ is a subset of another symbolic state $S' = (\ell', Z')$ if $\ell = \ell'$ and $Z \subseteq Z'$. Unfortunately, this approach is not complete as it is possible to miss optimal concrete cycles as shown in the following example.

***Counter-Example* 1** (Early Stopping) In this example, we will show an automaton where the optimal cycle is missed when a superset state has already been seen. Consider the automaton:



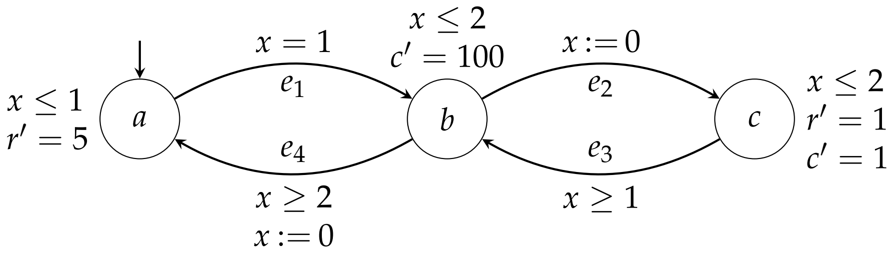
Below is the graph of the symbolic states where the $\Rightarrow$ edges denote the successor state after applying first a discrete edge transition by Post_e and then the delay by Post_ϵ. The e_1 successor for S_5 has been omitted for brevity.



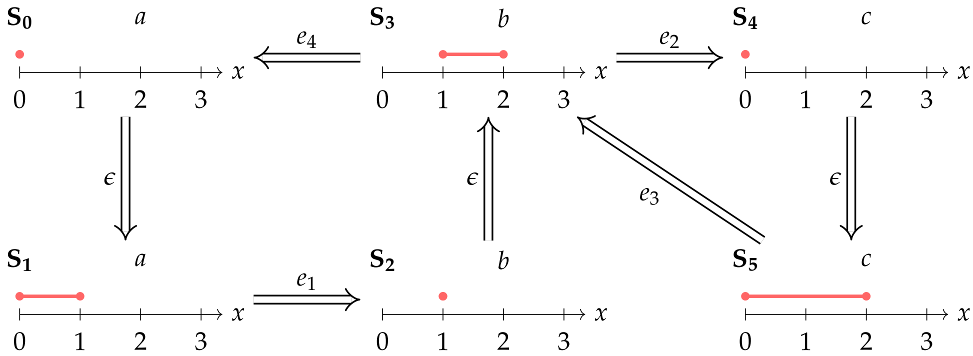
From this, we can see that S_4 is a subset of S_2. However, if the search for cycles is stopped here already, we will only get the cycles $S_0 \to S_1 \to S_0$, $S_0 \to S_1 \to S_2 \to S_3 \to S_0$, and $S_2 \to S_3 \to S_4$; each having optimal ratios of $\frac{11}{6}$, $\frac{7}{6}$, and $\frac{1}{1}$. The latter cycle is where the problem occurs: due to the stopping criterion, the optimal cycle $S_0 \to S_1 \to S_2 \to S_3 \to S_4 \to S_5 \to S_0$ is not found, which has an optimal ratio of $\frac{3}{8}$. The others are much worse than the optimal because they require delaying in the expensive location b. □

This problem can be solved by changing the stopping criterion to only stopping when the exact same symbolic state has been visited again, i.e. zone-wise subset is replaced with equals. By doing this, the approach will find all simple symbolic cycles. We know that simple concrete cycles are sufficient, but that is unfortunately not the case for simple symbolic cycles. The following counterexample shows this.

***Counter-Example* 2** (No Optimal Simple Symbolic Cycle) In this example, we show a cost-reward timed automaton, where the optimal concrete cycle is not in any simple symbolic cycle. Consider the automaton:



The symbolic state space is illustrated here:



Each of the 6 smaller diagrams represents a zone, where the red line represents the range of values the x-clock can have. The $\Rightarrow$ edges point to the symbolic state that is achieved by applying either Post_e or Post_ϵ.

The optimal concrete cycle is

$$(a,0) \xrightarrow{1} (a,1) \xrightarrow{e_1} (b,1) \xrightarrow{e_2} (c,0) \xrightarrow{2} (c,2) \xrightarrow{e_3} (b,2),$$

$$e_4$$

which has a cost-reward ratio of $\frac{2}{7}$. However, this concrete cycle is in the non-simple symbolic cycle $S_0 \to S_1 \to S_2 \to S_3 \to S_4 \to S_5 \to S_3 \to S_0$, with both S_0 and S_3 appearing twice. There are two simple symbolic cycles $S_0 \to S_1 \to S_2 \to S_3 \to S_0$ and $S_3 \to S_4 \to S_5 \to S_3$, however, the optimal concrete cycles in these only achieve a cost-reward ratio of $\frac{100}{5}$ and $\frac{1}{1}$, respectively. $\square$

5.3 Priced Zones

This section is copied from our pre-specialisation report. Priced zones were introduced by Larsen et al. [20], and the definitions of this section is based on their work. However, we generalise the zones to include rational-valued costs instead of just non-negative integers. Priced zones will be used later as the symbolic data structure in our main algorithm, symbolic λ-deduction in Section 7. This is also why we only consider single-priced zones, and not doubly-priced zones that accommodate both cost and reward.

Definition 5 (Priced Zone). A priced zone is a 3-tuple $\mathcal{Z} = (Z, c, r)$, where $Z \subseteq \mathbb{R}^{\mathbb{C}}_{\geq 0}$ is a zone and is a convex set of clock valuations with a unique minimal (by componentwise order) value $\Delta_Z \in Z$, c is the cost of Δ_Z, and $r : \mathbb{C} \to \mathbb{Q}$ assigns a cost rate to each clock. We write $u \in \mathcal{Z}$ whenever $u \in Z$. For any $u \in \mathcal{Z}$, the cost of u is $\mathsf{cost}(u, \mathcal{Z}) = c + \sum_{x \in \mathbb{C}} r(x) \cdot (u(x) - \Delta_Z(x))$. We denote by $\mathfrak{Z} = 2^{\mathbb{R}^{\mathbb{C}}_{\geq 0}} \times \mathbb{N} \times \mathbb{N}^{\mathbb{C}}$ the set of all priced zones.

Contrary to normal zones, priced zones are not closed under the Post_e and Post_ϵ operations; since the cost function is affine, it cannot capture the difference in cost rates that occur when a point in a zone is reached by using two delays with different cost rates.

To this end, we instead split the zones to represent this discrepancy. In order to determine the optimal splitting points, we use the notion of facets. Let $x \bowtie n$ (or $x - y \bowtie m$) be a constraint of the zone Z, then the "border" zone $Z \wedge (x = n)$ (or $Z \wedge (x - y = m)$ is a facet of Z. Facets derived from lower bound constraints (using $x \geq n$) are classified as lower facets, and we denote by $\mathsf{LF}(Z)$ all the lower facets. Likewise, upper bound constraints on the form $x \leq n$ are the upper facets and are denoted by $\mathsf{UF}(Z)$. We classify the facets relating to x, i.e. that are derived from the forms $x \geq n$ and $x - y \geq m$ as the lower relative facets w.r.t. x. The collection of lower relative facets of Z w.r.t. x is denoted by $\mathsf{LF}_x(Z)$. The collection of upper relative facets w.r.t. x, $\mathsf{UF}_x(Z)$ is expectedly derived using the forms $x \leq n$ and $x - y \leq m$.

Definition 6. Let $\mathcal{Z} = (F, c, r)$ be a priced zone, where F is a relative facet w.r.t. y s.t. $y - x = m$ is a constraint of F. Then $\mathcal{Z}_{\downarrow\{y\}} = (F', c', r')$, where $F' = F_{\downarrow\{y\}}, c' = c, r'(x) = r(x) + r(y), r'(z) = r(z)$ for all $z \neq x$. If $y = n$ is a constraint of F then $\mathcal{Z}_{\downarrow\{y\}} = (F', c, r)$ with $F' = F_{\downarrow\{y\}}$.

$$(Z, c, r)_{\downarrow R} = \begin{cases} (Z_{\downarrow R}, c, r[x \mapsto r(x) + r(y)]) & \text{if } y - x = m \text{ is a constraint of } Z, x \notin R \\ & \text{and } y \in R \\ (Z_{\downarrow R}, c, r) & \text{otherwise} \end{cases}$$

$$(Z, c, r)^{\uparrow p} = (Z^{\uparrow}, c, r[y \mapsto p - \sum_{x \neq y} r(x)]) \text{ where } y = n \text{ is a constraint of } Z$$

Conjunctions with constraints on priced zones can be achieved by simply addressing the change in the offset. Formally let $\mathcal{Z} = (Z, c, r)$ be a priced zone and let $g \in \mathcal{B}(\mathbb{C})$. Then $\mathcal{Z} \wedge g$ is the priced zone $\mathcal{Z}' = (Z', c', r)$, where $Z' = Z \wedge g$ and $c' = \mathsf{cost}(\Delta_{Z'}, \mathcal{Z})$. Also, for $\mathcal{Z} = (Z, c, r)$ we introduce the operation $\mathcal{Z} + n = (Z, c + n, r)$ for any number $n \in \mathbb{N}$.

Definition 7. Let $\mathcal{Z} = (F, c, r)$ be a priced zone, where F is a lower or upper facet in the sense that $y = n$ is a constraint of F. Let $p \in \mathbb{N}$ be a cost-rate. Then $\mathcal{Z}^{\uparrow p} = (F', c, r')$, where $F' = F^{\uparrow}, r'(y) = p - \sum_{z \neq y} r(z)$ and $r'(z) = r(z)$ for all $z \neq y$.

We are now ready to define the Post_e and Post_c operations for priced zones using their facets.

Definition 8. Let $\mathcal{A} = (L, \ell_0, E, I, P)$ be a PTA, and $e = (\ell, g, \{y\}, \ell') \in E$ and $\mathcal{Z} = (Z, c, r)$ be priced zone. Then:

$$\mathsf{Post}_e((\ell, \mathcal{Z})) = \begin{cases} \{\, (\ell', Q_{\downarrow\{y\}} + P(e)) \mid Q \in \mathsf{LF}_y(\mathcal{Z} \wedge g) \,\} & \text{if } r(y) \geq 0 \\ \{\, (\ell', Q_{\downarrow\{y\}} + P(e)) \mid Q \in \mathsf{UF}_y(\mathcal{Z} \wedge g) \,\} & \text{if } r(y) \leq 0 \end{cases}$$

$$\mathsf{Post}_\epsilon((\ell, \mathcal{Z})) = \begin{cases} \{\, (\ell, \mathcal{Z}) \,\} \cup \{\, (\ell, Q^{\uparrow P(\ell)} \wedge I(\ell) \mid Q \in \mathsf{UF}(\mathcal{Z} \wedge I(\ell)) \,\} & \text{if } p \geq \sum_{x \in \mathbb{C}} r(x) \\ \{\, (\ell, Q^{\uparrow P(\ell)} \wedge I(\ell) \mid Q \in \mathsf{LF}(\mathcal{Z} \wedge I(\ell) \,\} & \text{if } p \leq \sum_{x \in \mathbb{C}} r(x) \end{cases}$$

Any priced zone $\mathcal{Z} = (Z, c, r)$ can also be more intuitively rewritten as the pair Z and c, where Z is its unpriced zone and c is the induced *cost plane* s.t. $c(u) = c + \sum_{x \in \mathbb{C}} r(x) \cdot (u(x) - \Delta_Z(x))$. A priced symbolic state is then a location, zone and a cost plane. Similarly to (unpriced) symbolic states, we define a priced symbolic path $\Pi = S_0 \xrightarrow{\epsilon} S_0' \xrightarrow{e_0} S_1 \xrightarrow{\epsilon} \cdots \xrightarrow{e_{i-1}} S_i \xrightarrow{\epsilon} S_i' \xrightarrow{e_i} \cdots$ as a possibly infinite sequence of priced symbolic states, alternating between Post_ϵ and Post_e transitions. As with unpriced symbolic paths, we apply the following equivalent notations for priced symbolic paths. When the states of a priced symbolic path are distinct, we call it *simple*. We use the notation $S_0 \rightsquigarrow S_n$ whenever there exists a priced symbolic path from S_0 to S_n.

Definition 9 (Priced Symbolic Cycle). A priced symbolic path $\Pi = (\ell, Z, c) \rightsquigarrow (\ell', Z', c')$ is a *priced symbolic cycle* if the unpriced path $(\ell, Z) \rightsquigarrow (\ell', Z')$ is a symbolic cycle.

6 Extracting the Optimal Concrete Cycle from a Symbolic Cycle

The entirety of this section is copied from our pre-specialisation project due to the purpose of self-containment of this report.

In this section, we will present a method for finding the optimal concrete cycle from a symbolic cycle using linear-fractional programming. Essentially, this consists of finding optimal concrete delays for the symbolic delays. However, it is not obvious that there is an optimal concrete simple cycle that only goes through

the symbolic cycle once. Given a symbolic cycle Π, we want to find the optimal concrete cycle π s.t. there exists a k for which $\pi \in \Pi^k$, where Π^k is Π concatenated onto itself k times.

Theorem 3. *Let $\Pi = (\ell, Z_1) \rightsquigarrow (\ell, Z_n)$ be a symbolic cycle. Then for all $k > 1$, the optimal concrete cycle in Π^1 is exactly as good as the optimal concrete cycle in Π^k.*

Proof. Let π^k be a cycle in Π^k. We will then show how to construct a cycle $\pi^1 \in \Pi^1$ that has the same ratio. Let $\pi_1 \pi_2 \cdots \pi_k = \pi^k$ be the partitioning of π^k s.t. $\pi_i \in \Pi$ for $0 < i \leq k$. We then construct π^1 as a convex combination of all π_i. For a scalar $\alpha \in \mathbb{R}_{\geq 0}$ and a valuation $u \in \mathbb{R}^{\mathbb{C}}_{\geq 0}$, we define $(\alpha \cdot u)(x) = \alpha \cdot u(x)$ for $x \in \mathbb{C}$. Also, we define $(u + u')(x) = u(x) + u'(x)$ for $x \in \mathbb{C}$.

Let $(\alpha_i)_{i=1..k}$ be non-negative real scalars s.t. $\sum_{i=1}^{k} \alpha_i = 1$ and $\pi_i = (\ell_0, u_0^i) \to (\ell_1, u_1^i) \to \cdots \to (\ell_n, u_n^i)$. We then construct $\pi^1 = (\ell_0, v_0) \to (\ell_1, v_1) \to \cdots \to (\ell_n, v_n)$, where $v_j = \sum_{i=0}^{k} \alpha_i u_j^i$, i.e. the valuations of π^i is the convex combination of the valuations of all π_i. Also, whenever $\to$ is a delay transition, $(\ell_j, v_j) \xrightarrow{d_j} (\ell_{j+1}, v_{j+1})$, we let $d_j = \sum_{i=1}^{k} \alpha_i \delta_j^i$, i.e. the convex combination of the delays of π_i.

We now show that there exists $(\alpha_i)_{i=1..k}$ s.t. π^1 is a valid cycle. Specifically, we want to show that (i) each valuation satisfies the invariant of the location, (ii) the start and end valuations are the same (it is a cycle), and (iii) the delay and (iv) edge transitions are correct.

Firstly, (i) holds because the zones are convex, therefore any convex combination of valuations in a zone yields a valuation in the zone. For (ii) we know that $u_n^i = u_0^{(i+1 \bmod k)}$, since $\pi^{(i+1 \bmod k)}$ starts directly where π^i ends, and therefore, if we choose all α_i to be equal, i.e. $\alpha_i = \frac{1}{k}$, we have that

$$v_0 = \frac{1}{k}u_0^1 + \frac{1}{k}u_0^2 + \cdots + \frac{1}{k}u_0^k = \frac{1}{k}u_n^k + \frac{1}{k}u_n^1 + \cdots + \frac{1}{k}u_n^{k-1} = v_n.$$

For (iii) a delay transition $(\ell_j, v_j) \xrightarrow{d_j} (\ell_{j+1}, u_{j+1})$ we have that

$$v_j + d_j = \sum_{i=1}^{k} \alpha_i u_j^i + \sum_{i=1}^{k} \delta_j^i = \sum_{i=1}^{k} \alpha_i u_j^i + \delta_j^i = \sum_{i=1}^{k} \alpha_i u_{j+1}^i = v_{j+1}.$$

And finally, for (iv) a discrete edge transition $(\ell_j, v_j) \xrightarrow{e_j} (\ell_{j+1}, u_{j+1})$, where $e_j = (\ell_j, R, g, \ell_{j+1})$, the guard is satisfied for the same reason (i), and

$$v_j[R \mapsto 0] = \left(\sum_{i=1}^{k} \alpha_i u_j^i\right)[R \mapsto 0] = \sum_{i=1}^{k} \alpha_i u_j^i[R \mapsto 0] = \sum_{i=1}^{k} \alpha_i u_{j+1}^i = v_{j+1}.$$

To see that the reset operation distributes over a sum of valuations, we observe that

$$\left(\sum_{i=1}^{k} \nu_i\right)[R \mapsto 0](x) = \begin{cases} 0 & \text{if } x \in R \\ \sum_{i=1}^{k} \nu_i(x) & \text{otherwise} \end{cases} = \left(\sum_{i=1}^{k} \nu_i[R \mapsto 0]\right)(x).$$

Finally, we will show that π^1 is indeed at least as good as π^k. The ratio of π^k is

$$\mathsf{Ratio}(\pi^k) = \frac{\sum_{i=1}^{k} \left(\sum_{j=0}^{n-1} \delta_j^i \cdot \mathrm{c}(\ell_j) + c_e\right)}{\sum_{i=1}^{k} \left(\sum_{j=0}^{n-1} \delta_j^i \cdot \mathrm{r}(\ell_j) + r_e\right)} = \frac{\sum_{i=1}^{k} \sum_{j=0}^{n-1} \delta_j^i \cdot \mathrm{c}(\ell_j) + k \cdot c_e}{\sum_{i=1}^{k} \sum_{j=0}^{n-1} \delta_j^i \cdot \mathrm{r}(\ell_j) + k \cdot r_e}$$

where $c_e = \sum_{j=0}^{n-1} \mathrm{c}(e_j)$ and $r_e = \sum_{j=0}^{n-1} \mathrm{r}(e_j)$. We then rewrite this by swapping the order of the summation

$$= \frac{\sum_{j=0}^{n-1} \sum_{i=1}^{k} \delta_j^i \cdot \mathrm{c}(\ell_j) + k \cdot c_e}{\sum_{j=0}^{n-1} \sum_{i=1}^{k} \delta_j^i \cdot \mathrm{r}(\ell_j) + k \cdot r_e},$$

and applying the definition of $d_j = \sum_{i=1}^{k} \alpha_i \delta_j^i$ for $\alpha_i = \frac{1}{k}$

$$= \frac{\sum_{j=0}^{n-1} k \cdot \frac{1}{k} \sum_{i=1}^{k} \delta_j^i \cdot \mathrm{c}(\ell_j) + k \cdot c_e}{\sum_{j=0}^{n-1} k \cdot \frac{1}{k} \sum_{i=1}^{k} \delta_j^i \cdot \mathrm{r}(\ell_j) + k \cdot r_e} = \frac{k \cdot \sum_{j=0}^{n-1} d_j \cdot \mathrm{c}(\ell_j) + k \cdot c_e}{k \cdot \sum_{j=0}^{n-1} d_j \cdot \mathrm{r}(\ell_j) + k \cdot r_e},$$

and by removing the common factor k we get

$$= \frac{\sum_{j=0}^{n-1} d_j \cdot \mathrm{c}(\ell_j) + c_e}{\sum_{j=0}^{n-1} d_j \cdot \mathrm{r}(\ell_j) + r_e)} = \mathsf{Ratio}(\pi^1).$$

□

Corollary 4. *Let $\Pi = (\ell, Z_1) \rightsquigarrow (\ell, Z_n)$ be a symbolic cycle in a single weight timed automata. Then for all $k > 1$, there exists a concrete cycle with weight w in Π^1 if and only if there exists a concrete cycle with weight $w \cdot k$ in Π^k.*

Proof. This follows easily from Theorem 3. We can simply pretend that $\mathsf{Reward}(\Pi) = 1$, and then let $\pi \in \Pi$ be a minimum ratio cycle, and $\mathsf{Ratio}(\pi) = \frac{w}{1}$, then π is also the minimum weight cycle in Π and has weight w. This is because when the reward is fixed, the only objective is to minimise cost. Now, let $\pi^k \in \Pi^k$ be a minimum ratio cycle of Π^k, we then know from Theorem 3 that $\mathsf{Ratio}(\pi^k) = \frac{k \cdot w}{k}$, and thus has weight $k \cdot w$. □

6.1 Linear-Fractional Programming

In this section, we will describe a method for finding the optimal concrete cycle in a symbolic cycle by utilising linear-fractional programming. This technique was introduced by Tolonen et al. [14].

Let $\Pi = (\ell_0, Z_0) \xrightarrow{\epsilon_0} (\ell_0, Z_0') \xrightarrow{e_0} (\ell_1, Z_1) \xrightarrow{\epsilon_1} \dots \xrightarrow{e_n} (\ell_n, Z_n)$ be a symbolic cycle. To formulate the linear-fractional program we will use timestamps. A timestamp is the total time that has occurred since the start of the cycle to the point of an event. We will create n time stamps $\mathbf{t} = (t_1, \dots, t_n)$ specifying the time at which the state (ℓ_i, Z_i) was reached. Additionally, we have the zero timestamp $t_0 = 0$. We can extract the value of a clock $x \in \mathbb{C}$ in a timestamp t_i by the difference between t_i and the latest timestamp t_j at which x was reset. Here we define

$$\textsc{lastReset}(i, x) = \begin{cases} \max\{\, j \leq i \mid x \in R_j \,\} & \text{if such } j \text{ exists} \\ \max\{\, j > i \mid x \in R_j \,\} & \text{otherwise} \end{cases},$$

where $e_j = (\ell_{j-1}, g, R_j, \ell_j)$, as the index of the latest edge where x was reset before i. By using the boundedness technique we guarantee that all clocks are reset in the cycle, however, for this section we will assume that all resets are to 0 to simplify matters. Notice that because we are working with cycles, the second case is needed because the index of the last reset of the clock might be larger than i. We can now give the value of a clock at each index before delay

$$v_b(i,x) = \begin{cases} t_i - t_{\text{LASTRESET}(i,x)} & \text{if LASTRESET}(i,x) \leq i \\ (t_i + t_n) - t_{\text{LASTRESET}(i,x)} & \text{otherwise} \end{cases}$$

and after delay

$$v_a(i,x) = \begin{cases} t_{i+1} - t_{\text{LASTRESET}(i,x)} & \text{if LASTRESET}(i,x) \leq i \\ (t_{i+1} + t_n) - t_{\text{LASTRESET}(i,x)} & \text{otherwise} \end{cases}.$$

With this notation in order, we can now state the objective function in terms of $\mathbf{t}$ and the constraints. Firstly, the objective function is the cost-reward ratio but rewritten as timestamps. The definition of Ratio from Section 2.2, can be rewritten by defining $\mathsf{cost}(s_i, s_{i+1}) = (t_{i+1} - t_i) \cdot \mathrm{c}(\ell_i)$ and $\mathsf{reward}(s_i, s_{i+1}) = (t_{i+1} - t_i) \cdot \mathrm{r}(\ell_i)$. This way we get the ratio as

$$\mathsf{Ratio}(\mathbf{t}) = \frac{\sum_{i=0}^{n-1} (t_{i+1} - t_i) \cdot \mathrm{c}(\ell_n) + \sum_{i=0}^{n} \mathrm{c}(e_n)}{\sum_{i=0}^{n-1} (t_{i+1} - t_i) \cdot \mathrm{r}(\ell_n) + \sum_{i=0}^{n} \mathrm{r}(e_n)}.$$

Firstly, we add constraints on the timestamps to ensure that there are no negative delays, i.e.

$$t_i \leq t_{i+1} \quad \textit{and} \quad 0 \leq t_i.$$

The constraints can then be extracted from the zones. For each before-delay zone Z_i and each constraint $x - y \leq m$ of Z_i in the DBM representation, we add the constraint

$$v_b(i,x) - v_b(i,y) \leq m,$$

where we define $v_b(i, \underline{\mathbf{0}}) = 0$, with $\underline{\mathbf{0}}$ being the 0-clock. Similarly, for each after-delay zone Z_i' and each constraint $x - y \leq m$ we add the constraint

$$v_a(i, x) - v_a(i, y) \leq m.$$

Since the symbolic cycle is stable, the zones represent exactly the set of allowed valuations in each state, therefore any assignment of $\mathbf{t}$ that satisfies these constraints corresponds exactly to the set of concrete cycles in the symbolic cycle.

The linear-fractional programming formulation is usually given as

$$\begin{aligned} \text{minimize} \quad & \frac{\mathbf{c}^T\mathbf{t} + c}{\mathbf{r}^T\mathbf{t} + r} \\ \text{subject to} \quad & A\mathbf{t} \leq \mathbf{b}. \end{aligned}$$

The cost vector $\mathbf{c}$ needs to represent the difference in cost between the timestamps. This can be seen by observing that the cost should be calculated as

$$\begin{aligned} c_0(t_1 - t_0) + c_1(t_2 - t_1) + \cdots + c_{n-1}(t_n - t_{n-1}) = \\ t_0(0 - c_0) + t_1(c_0 - c_1) + t_2(c_1 - c_2) + \cdots t_{n-1}(c_{n-2} - c_{n-1}) + t_n(c_{n-1} - 0). \end{aligned}$$

Therefore,

$$\mathbf{c}_i = \mathrm{c}(\ell_{i-1}) - \mathrm{c}(\ell_i)$$

for $1 \leq i < n$ and $\mathbf{c}_n = \mathrm{c}(\ell_{n-1})$. And dually for $\mathbf{r}$. Also, c and r are simply the sums of the cost and reward of discrete edges, respectively. The matrix A is of the form $J \times n$, where J is the number of constraints, and the vector b is of length n. The $j'th$ row in A and b corresponds to the j'th constraint. Let the j'th constraint be $v_b(i, x) - v_b(i, y) \leq m$, then substitute each t_i with $\mathbf{1}^i$, where $\mathbf{1}^i$ is the one-hot vector $\mathbf{1}^i_h = 0$ for all $h \neq i$ and $\mathbf{1}^i_i = 1$, and compute $v_b(i, x) - v_b(i, y)$ as a the j'th row of A and let $\mathbf{b}_j = m$.

The linear-fractional program can be transformed into a linear program by the Charnes-Cooper transformation[26]. This yields the linear program

$$\begin{aligned} \text{minimize} \quad & \mathbf{c}^T\mathbf{y} + c \cdot d \\ \text{subject to} \quad & A\mathbf{y} \leq \mathbf{b}d \\ & \mathbf{r}^T\mathbf{y} + r \cdot d = 1 \\ & d \geq 0. \end{aligned}$$

where

$$\mathbf{y} = \frac{1}{\mathbf{r}^T\mathbf{t} + r} \cdot \mathbf{t} \quad \text{and} \quad d = \frac{1}{\mathbf{r}^T\mathbf{t} + r}.$$

From the solution to the transformed linear program, we can extract the linear-fractional program solutions as $\mathbf{t} = \frac{1}{d}\mathbf{y}$.

7 Symbolic λ-deduction

In this section, we give our main contribution, symbolic λ-deduction—an algorithm for finding ratio-optimal cycles in CRTA that is based on an abstract algorithm presented in [27]. They show how ratio-style problems can be solved by incrementally finding better solutions until the optimal is found. This style of algorithm works by maintaining a λ value that is the ratio of the best solution found so far. It uses this λ value to reduce the problem of finding ratio-optimal solutions to finding negative weight solutions, by replacing cost and reward with a single λ-deducted weight. For a solution $x \in X$, the λ-deducted weight is $w_\lambda(x) = \mathsf{Cost}(x) - \lambda \cdot \mathsf{Reward}(x)$.

Proposition 5. *A solution $x \in X$ has negative λ-deducted weight if and only if $\frac{\mathsf{Cost}(x)}{\mathsf{Reward}(x)} < \lambda$.*

Proof. $\mathsf{Cost}(x) - \lambda \cdot \mathsf{Reward}(x) < 0 \iff \frac{\mathsf{Cost}(x)}{\mathsf{Reward}(x)} < \lambda.$ □

The above proposition suggests a method using the following principle: First, pick an element x from the solution space X, and let $\lambda := \frac{\mathsf{Cost}(x)}{\mathsf{Reward}(x)}$. For a solution $x' \in X$, we call $w_\lambda(x') = \mathsf{Cost}(x') - \lambda \cdot \mathsf{Reward}(x')$ the λ-deducted weight of

x'. Then, find a solution $x' \in X$ s.t. $\mathsf{Cost}(x') - \lambda \cdot \mathsf{Reward}(x') < 0$, and update $\lambda := \frac{\mathsf{Cost}(x')}{\mathsf{Reward}(x')}$. This is repeated until no such x' exists, at which point an optimal solution has been found.

In order to use this approach in our setting, we need to devise a method for finding cycles that have a negative λ-deducted weight. We therefore introduce a transformation of the original cost-reward automaton into a λ-deducted single priced timed automaton.

Definition 10. Let $\mathcal{A} = (L, \ell_0, E, I, \mathrm{c}, \mathrm{r})$ be a CRTA. The λ-deducted single priced timed automata of $\mathcal{A}$ is $\mathcal{A}_\lambda = (L, \ell_0, E, I, w_\lambda)$, where

$$w_\lambda(a) = \mathrm{c}(a) - \lambda \cdot \mathrm{r}(a) \qquad \text{for } a \in L \cup E.$$

Notice that the weight of any path in $\mathcal{A}_\lambda$ corresponds to the λ-deducted weight of the path in $\mathcal{A}$. This follows from the semantics of $\mathcal{A}$ and the additive nature of the λ-deducted weight. We say a cycle in $\mathcal{A}_\lambda$ is a *negative-weight concrete cycle* if the summed weight of the path is negative.

Algorithm 1 shows an abstract algorithm for finding ratio-optimal cycles using the λ-deducted automaton. The value of λ is the ratio of the so far best found cycle. The intuition is to initialise λ to ∞, which causes any cycle incurring some reward to be negative, and then iteratively improve it by finding negative cycles in $\mathcal{A}_\lambda$. Algorithm 1 terminates only when there no longer exists a negative cycle in $\mathcal{A}_\lambda$, which happens only when the value of λ is exactly the optimal ratio. For this algorithm, we need only one operation: find any negative-weight concrete cycle in $\mathcal{A}_\lambda$ if one exists.

Theorem 6. *Algorithm 1 terminates, and returns a ratio-optimal concrete cycle, if one exists, otherwise* NO CYCLE.

Proof. First, realise that, for any $\lambda \in \mathbb{Q}$ (rational numbers), $\mathcal{A}_\lambda$ has identical valuations and paths as $\mathcal{A}$ (only the pricing differs). We know from the corner-point abstraction [8], that we can discretise the state space into only considering integer-valued points with at least one ratio-optimal cycle being kept. Then, together with

Algorithm 1: Abstract algorithm for finding ratio optimal concrete cycle by improving λ.

input : A bounded and strongly reward-divergent CRTA $\mathcal{A} = (L, \ell_0, E, I, \mathrm{c}, \mathrm{r})$ over the clocks $\mathbb{C}$.
output: A ratio-optimal concrete cycle, if one exists, otherwise NO CYCLE.

$\lambda := \infty$
$C_\lambda :=$ NO CYCLE
while $\mathcal{A}_\lambda$ has negative-weight simple discrete cycle C **do**
    $\lambda := \mathsf{Ratio}(C)$
    $C_\lambda := C$
return C_λ

$\mathcal{A}_\lambda$ begin bounded, there is only a finite amount of simple concrete cycles in $\mathcal{A}_\lambda$, i.e. the considered solution space is finite.

With $\lambda = \infty$, any cycle C in $\mathcal{A}_\lambda$ will have negative weight. This is because the automaton is reward-divergent, and therefore $\mathsf{Reward}(C) > 0$, which gives $\mathsf{Cost}(C) - \infty \cdot \mathsf{Reward}(C) < 0$. Thus, Algorithm 1 only returns NO CYCLE if and only if $\mathcal{A}$ contains no cycle.

Assume $\mathcal{A}$ contains a simple discrete cycle C, and recall that $\mathsf{Cost}(C) - \lambda \cdot \mathsf{Reward}(C) < 0 \iff \frac{\mathsf{Cost}(C)}{\mathsf{Reward}(C)} < \lambda$. Thus, at each iteration of the while loop, λ is decreased. The value of λ can only be decreased a finite amount of times because the solution space is finite. When $\mathcal{A}_\lambda$ contains no negative-weight cycle, Algorithm 1 terminates, and C_λ is returned with the optimal ratio of λ. □

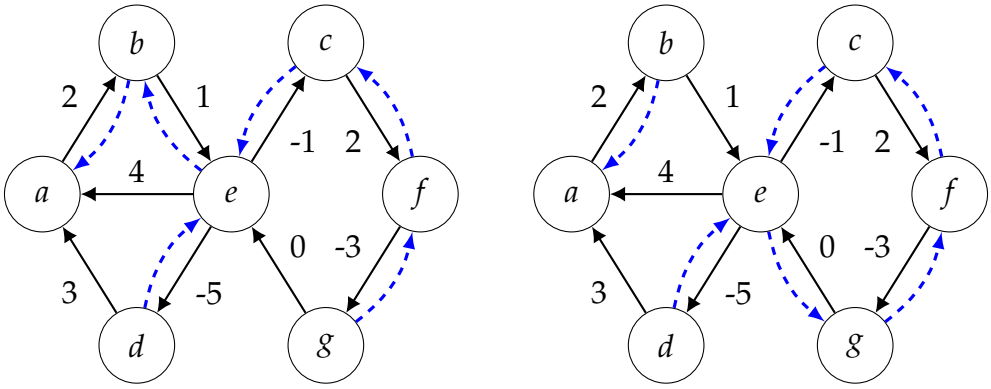
The λ-deducted automaton has only a single weight function (as opposed to the two of cost and reward), and we can therefore use techniques similar to those of cost-optimal reachability for priced timed automata [20], namely priced zones. The only difference being that we will extend the domain for prices from non-negative integers to rational numbers, because the weights of the λ-deducted automaton may be a rational number. Fortunately, this does not impact any of the formal definitions of the operations on priced zones.

7.1 Finding Negative-Weight Concrete Cycles

In this section, we elaborate on how to find a negative-weight concrete cycle in λ-deducted automata. Specifically, how to compute "$\mathcal{A}_\lambda$ has negative-weight concrete cycle C" on line 3 in Algorithm 1. This section will conclude with a complete non-abstract algorithm for symbolic λ-deduction.

In a finite weighted graph, the most prevalent approach for detecting negative cycles is to maintain some type of information about the shortest path between nodes. For example, the Bellman-Ford shortest path algorithm works by maintaining, for each vertex, a *parent* pointer s.t. for a source vertex s and a vertex v, the shortest path from s to v found has $\mathit{parent}(v)$ as its second-to-last vertex. When the main part of the Bellman-Ford algorithm is completed, the *parent* pointer, for any vertex, contains a shortest path predecessor. In order to detect a negative cycle, we can then follow the *parent* pointer backwards and if it contains a cycle this is a negative-weight cycle. A cycle in the *parent* pointer implies that the cheapest way to reach a vertex v (in the *parent* cycle) is to start in v itself, therefore the weight of the cycle must be non-positive. By also adding the constraint that the *parent* pointer is only updated whenever a strictly cheaper path has been found, the cycle in the parent pointer must be negative, because the second time v is found it is strictly cheaper.

***Example* 4** Below we show a simple weighted directed graph, where the solid black arrows show the edges of the graph and the dashed blue arrows show the parent pointer. On the left, we see the state of the parent pointers before checking edge $g \to e$. On the right, we see the state of the parent pointers after exploring the edge $g \to e$. Observe how the parent pointer now contains a cycle, as the shortest path to e is $a \to b \to e \to c \to f \to g \to e$.



□

We can apply the *parent* function to the single-priced λ-deducted automata setting. By using corner-point abstraction together with boundedness, this will directly reduce to the finite weighted graph problem. However, as a possibly more efficient approach, we propose to instead use symbolic priced zones to find negative concrete cycles. Searching symbolically will yield symbolic cycles, which abstract over many concrete cycles.

Definition 11 (Negative-Weight Symbolic Cycle). A priced symbolic cycle Π is a *negative -weight symbolic cycle* if there exists a negative-weight concrete cycle $\pi \in \Pi$.

We shall often refer to a negative-weight symbolic cycle as simply a *negative* cycle, when it is clear from the context. It is non-trivial to determine whether or not a priced symbolic cycle is negative. One method is to use linear programming, as described in Section 6, to compute the optimal concrete cycle contained in the symbolic cycle, and then check whether it has negative weight.

In this symbolic version, we also maintain a *Parent* pointer function as our method of detecting cycles. The *Parent* pointer maps each symbolic state S to either a pair of the predecessor state S' and an action $\alpha \in E \cup \{\epsilon\}$, s.t. $S' \xrightarrow{\alpha} S$, or the special symbol NIL. The NIL symbol is used as the parent of the initial state. Similarly to the finite weighted graph case, we are only interested in the cheapest way to reach a state. Therefore, we are not interested in exploring paths where we have already found a cheaper way of getting to the same state.

Definition 12 (Domination Between Priced Symbolic States). A priced symbolic state $S = (\ell, Z, w)$ *dominates* another state $S' = (\ell', Z', w')$, denoted by $S \sqsubseteq S'$, if and only if (i) $\ell = \ell'$, (ii) $Z \supseteq Z'$, and (iii) $w(u) \leq w'(u)$ for all $u \in Z'$.

In the finite weighted graph case, if the *parent* pointer function composes a concrete cycle, then it is guaranteed to be a cycle with a non-positive ratio. Contrarily, a symbolic cycle formed by the *Parent* pointer function has no such guarantees. Therefore, we must always check whether the cycle contains a negative-weight concrete cycle. We will discuss a more efficient method of doing this in Section 7.3. There is an important difference between cycles in the *Parent* pointer in the symbolic case and the *parent* pointer in the concrete case. Namely, in the concrete case the formed cycle will visit the exact same states, whereas, the symbolic cycles include cost-information of reaching the underlying state, wherefore, there can be a cycle without visiting the exact same state again. Importantly, there can be no negative-weight symbolic cycle where the exact same state is visited again, because this implies that in each revolution the price stays the same—whereas a negative cycle would cause it to become cheaper and cheaper.

Algorithm 2 shows symbolic λ-deduction: a best-first-search style algorithm that uses priced zones to find negative concrete cycles in a single-priced timed automaton. We use the function $\mathrm{dom}(f) = X$ to give the domain of a (partial) function $f : X \rightharpoonup Y$. On lines 1-3 the algorithm initialises the λ-value, the concrete cycle that produces the given λ-value, and a *Waiting* set with states that are yet to be explored, which is initialised to the initial symbolic state.

The main while loop on line 4 is equivalent to the abstract Algorithm 1. At each iteration, a new negative cycle is returned by FIND-NEGATIVE-CYCLE with a better ratio than C_λ. Then λ and C_λ are updated to reflect the new best cycle.

The subroutine FIND-NEGATIVE-CYCLE returns a negative-weight symbolic cycle in $\mathcal{A}_\lambda$, if one exists, otherwise NO CYCLE. It extracts and expands a minimum element from *Waiting* on line 12, according to some ordering. The order chosen has no consequence on termination and correctness of the algorithm, but it may affect the efficiency. We suggest the minimum to be the state containing minimum weight valuation—i.e. the valuation which is smaller than all other

valuations in all other states—from the intuition that cheap valuations are more likely to produce negative cycles. On line 14, the algorithm avoids expanding successors where a dominating variant has already been expanded—this is needed for termination so that we do not follow cycles that become more and more expensive each revolution in the cycle. Here we only do a fast comparison, i.e. we do not check if any combination of cost planes is cheaper—e.g. a state S may not be dominated by neither S' nor S'', but S could be partitioned s.t. one partition is dominated by S' and the other is dominated by S''. If the successor is not dominated, the parent of the successor is set, and it is added to the *Waiting* list. We finally check whether a negative-weight concrete cycle is present in the parent function on line 17 by calling the subroutine FIND-NEGATIVE-CYCLE. This recursive subroutine follows the *Parent* pointer backwards, reconstructing the path taken to the state until a negative-weight symbolic cycle is found (if one exists).

Algorithm 2: Symbolic λ-deduction.

input : A bounded and strongly reward-divergent CRTA $\mathcal{A} = (L, \ell_0, E, I, c, r)$ over the clocks $\mathbb{C}$.

output: A ratio-optimal concrete cycle, if one exists, otherwise NO CYCLE.

$\lambda := \infty$
$C_\lambda :=$ NO CYCLE
Waiting $:= \{ S_0 \}$ `// initial state`
while FIND-NEGATIVE-CYCLE() returns a negative concrete cycle C **do**
    $\lambda := \mathsf{Ratio}(C)$
    $C_\lambda := C$
return C_λ
Function FIND-NEGATIVE-CYCLE()
    Waiting $:= \{ S_0 \}$
    Parent $:= \emptyset[S_0 \mapsto \text{NIL}]$`// function mapping` S_0 `to NIL`
    while *Waiting* $\neq \emptyset$ **do**
        $S :=$ EXTRACT-MIN(*Waiting*)
        forall $S' \in \mathsf{Post}_\alpha(S)$ for all $\alpha \in E \cup \{\epsilon\}$ **do**
            if $\forall S'' \in \mathrm{dom}(\mathit{Parent}).\ S'' \not\sqsubseteq S'$ **then**
                $\mathit{Parent}(S') := (S, \alpha)$
                insert S' into *Waiting*
                `/* Recursively follow parent to check for cycle */`
                if PARENT-NEGATIVE-CYCLE($S \xrightarrow{\alpha} S'$) returns cycle C **then**
                    return C
    return NO CYCLE
Function PARENT-NEGATIVE-CYCLE($S_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_n} S_n$)
    if $S_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_n} S_n$ is a negative-weight symbolic cycle **then**
        return best concrete cycle in $S_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_n} S_n$
    if $S_1 \in \mathrm{dom}(\mathit{Parent})$ **then**
        $(S, \alpha) := \mathit{Parent}(S_1)$
        return PARENT-NEGATIVE-CYCLE($S \xrightarrow{\alpha} S_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_n} S_n$)
    else
        return NO CYCLE

7.2 Proof of Correctness

Theorem 7. *Algorithm 2 terminates, and it returns a ratio-optimal concrete cycle, if one exists, otherwise* NO CYCLE.

Proof. First, realise that the main while loop on line 4 follows the exact same structure as in Algorithm 1, which is correct and terminates (Theorem 6). Therefore, it suffices to show that FIND-NEGATIVE-CYCLE is correct and terminates. We divide this proof into several lemmas and prove them individually. We prove termination in Lemma 9. We then prove the correctness by soundness in Lemma 10 and completeness in Lemma 11. □

In order to prove that FIND-NEGATIVE-CYCLE terminates, we first prove that PARENT-NEGATIVE-CYCLE always terminates when called in FIND-NEGATIVE-CYCLE on line 21. This is done by proving that it always forms a tree structure because then recursively following the pointer backwards stops once the root is reached.

Lemma 8. *The while-loop on line 11 in* FIND-NEGATIVE-CYCLE *has the loop-invariant: the Parent pointer always forms an in-tree (all nodes lead to the root-node).*

Proof. A directed graph is an in-tree if it satisfies the property that for all nodes there exists a unique walk to the root. A *walk* is a sequence of not-necessarily distinct edges, which form a sequence of not-necessarily distinct nodes.

Trivially, $Parent := (\text{NIL}, S_0)$ initialises it as an in-tree with NIL as the root-node, thus the loop starts with the invariant satisfied. The statement $Parent(S') := (S, \alpha)$ on 15 is the only place that the *Parent* pointer is altered. We prove that this statement does not break the invariant. A state is added to *Waiting* only if it is defined in *Parent* (lines 15 and 16). On line 14, it is ensured that a successor state S' of S is only added to *Parent* if it is not already a state in *Parent*, because the equality relation is a subset of the domination relation. There is a unique walk from S to NIL and a unique walk from S' to S (single edge), and therefore also a unique walk from S' to NIL via S. Conclusively, it maintains that the walk from any node to the root is unique. □

Lemma 9. *The subroutine* FIND-NEGATIVE-CYCLE *in Algorithm 2 terminates.*

Proof. We first argue that there exists a constant w_{min} for $\mathcal{A}_\lambda$ s.t. any symbolic path that contains a concrete path with weight less than w_{min} must contain a negative-weight concrete cycle. Let $\mathcal{G}_{\mathcal{A}_{cp}} = (\mathcal{V}, \mathcal{E}, \mathcal{C}, \mathcal{R})$ be the finite doubly weighted graph induced by the corner point abstraction of the automaton $\mathcal{A}$, then

$$w_{min} = \sum \{ w_\lambda(e) \mid e \in \mathcal{E}, w_\lambda(e) < 0 \},$$

where $w_\lambda(e) = \mathcal{C}(e) - \lambda \cdot \mathcal{R}(e)$, i.e. the sum of all of the negative λ-deducted edges. Any concrete path with weight less than w_{min} must use at least one negative edge more than once, and therefore, it contains at least one concrete cycle. At least one of these cycles must be negative, otherwise the weight of the acyclic part of the path is less than w_{min}, which cannot happen because some negative edge must be used twice.

Now, let $\mathcal{S}$ be the set of all priced symbolic states and $\mathcal{S}_{\geq w_{min}} = \{ (\ell, Z, c) \in \mathcal{S} \mid \forall u \in Z.\, c(u) \geq w_{min} \}$. We now argue that $(\mathcal{S}_{\geq w_{min}}, \sqsubseteq)$ is a well-quasi order, i.e. there exists no infinite descending sequence where the states are not dominated by a previous state. For a bounded priced timed automaton, there is only a finite number of (unpriced) zones, thus a zone must eventually repeat in an infinite sequence. The minimum of a priced zone is attained in an integer point, therefore, we can limit ourselves to study the cost plane only in the finite number of integer points. Let $u \in Z$ be an integer point of zone Z and $\lambda = \frac{a}{b}$ for $a \in \mathbb{Z}$ and $b \in \mathbb{N}^+$, then the λ-deducted weight of reaching the integer point is

$$w_\lambda(u) = \mathsf{Cost}(u) - \frac{a}{b} \cdot \mathsf{Reward}(u) = \frac{\mathsf{Cost}(u) \cdot b - a \cdot \mathsf{Reward}(u)}{b}.$$

Since $\mathsf{Cost}(u)$ and $\mathsf{Reward}(u)$ are integer-valued, the λ-deducted weight of integer points can be written as a rational with b as a constant denominator. We observe that we can scale the cost-plane by b,

$$w_\lambda(u) \geq w_{min} \iff \mathsf{Cost}(u) \cdot b - a \cdot \mathsf{Reward}(u) \geq w_{min} \cdot b,$$

where $\mathsf{Cost}(u) \cdot b - a \cdot \mathsf{Reward}(u)$ is integer-valued. Thus, the cost-plane cannot keep improving ad infinitum, since the integer points can be considered to have integer-valued weights greater than $w_{min} \cdot b$, and there are only finitely many integer points in zone Z. This, in conjunction with the finiteness of zones, shows that $(\mathcal{S}_{\geq w_{min}}, \sqsubseteq)$ is a well-quasi order. The argument is similar to Dickson's Lemma[28], but instead of dealing with natural numbers, we have lower bounded integers.

We now argue that these properties guarantee that our algorithm terminates. Firstly, the subroutine FIND-NEGATIVE-CYCLE terminates either when $\mathit{Waiting} = \emptyset$ or when PARENT-NEGATIVE-CYCLE returns a cycle. The if-statement on line 14 ensures that we never expand a state that is dominated by a previously expanded state. If no state is ever expanded that contains a point cheaper than w_{min}, then there exists no infinite descending sequence for $\sqsubseteq$, and thus the algorithm will terminate. Otherwise, whenever any state containing a point cheaper than w_{min} is expanded, then there must be a negative cycle present in the parent pointer. We now argue that whenever the parent pointer contains a suffix that constitutes a negative symbolic cycle, PARENT-NEGATIVE-CYCLE returns a negative concrete cycle.

By Lemma 8, the *Parent* pointer always forms an in-tree, thus the recursive subroutine PARENT-NEGATIVE-CYCLE terminates because it keeps following the finite (as in $|\mathrm{dom}(\mathit{Parent})| \neq \infty$) parent function backwards, growing it until it either forms a negative-weight symbolic cycle or reaches the initial state, whence the recursion stops. Since PARENT-NEGATIVE-CYCLE checks all suffixes to see if they constitute a negative symbolic cycle, it will find a negative symbolic cycle if it exists. □

Lemma 10. *The subroutine* FIND-NEGATIVE-CYCLE *in Algorithm 2 is sound, i.e. if there does not exist a negative-weight concrete cycle in $\mathcal{A}_\lambda$ then it returns* NO CYCLE.

Proof. We prove the contrapositive, i.e. if it returns a cycle then there exists a negative-weight concrete cycle in $\mathcal{A}_\lambda$. We do this by showing that any cycle returned by FIND-NEGATIVE-CYCLE is a reachable negative-weight concrete cycle.

Whenever FIND-NEGATIVE-CYCLE is called, the *Waiting* list contains exactly the initial state. The successor states generated by the Post operation implies that there exists a concrete path to any point in the successor state from some point in the predecessor state. Since the initial state is the only state in *Waiting* in the beginning, all concrete states contained in a generated symbolic state are reachable. When assuming that the concrete cycle found in line 22 is a negative-weight concrete cycle, then when Algorithm 2 returns a cycle, it is a reachable negative-weight concrete cycle. □

Lemma 11. *The subroutine* FIND-NEGATIVE-CYCLE *of Algorithm 2 is complete, i.e. if it returns* NO CYCLE *then there does not exist a negative cycle in* $\mathcal{A}_\lambda$.

Proof. Again, we prove the contrapositive, i.e. if there exists a negative cycle in $\mathcal{A}_\lambda$ then FIND-NEGATIVE-CYCLE returns a cycle. Let π be a negative-weight simple concrete cycle in $\mathcal{A}_\lambda$. Construct from π the symbolic simple cycle Π by replacing all concrete delays with symbolic delays. Now, let Π' be the smallest, in length, subcycle s.t. $\Pi = (\Pi')^k$, for some $k > 0$. Since $\pi \in \Pi$ and π is a negative-weight cycle, by Corollary 4 there also exists a negative-weight cycle $\pi' \in \Pi'$. We will now show that FIND-NEGATIVE-CYCLE, if it does not find some other negative cycle, will find a negative cycle in Π'.

The core principle of the algorithm is that the *Parent* function maintains a dominating (w.r.t. $\sqsubseteq$) predecessor for all concrete states that have been explored through a symbolic state. In other words, a cheapest path from the initial state to any expanded state is reconstructible from the *Parent* function by following the states backwards. Observe that since Π' is a negative-weight symbolic cycle, if $S \overset{\Pi'}{\rightsquigarrow} S'$ then there is at least one concrete state that is cheaper to reach in S' than in S, therefore $S \not\sqsubseteq S'$. Thus, when S' is generated, it will pass the if-statement on line 14, and PARENT-NEGATIVE-CYCLE will look for a cycle ending in S', and return a negative-weight concrete cycle. □

7.3 Efficiently Determining Whether a Symbolic Cycle is Negative

We will now discuss an efficient approach for determining whether or not a priced symbolic cycle is a negative weight symbolic cycle. Specifically in Algorithm 2, it is done in the function PARENT-NEGATIVE-CYCLE. Whenever the *Parent* pointer contains a priced symbolic cycle, it is checked whether that cycle is negative. Checking for such negativity is non-trivial; we need an efficient method of doing so since this check may be executed many times in Algorithm 2. We already know we can check this using linear programming (described in Section 6), however, this is undesirable since constructing and solving the linear program is rather slow. In this section, we show a less computationally expensive method to determine if a symbolic cycle has negative weight.

The core idea is motivated by the fact that a negative weight symbolic cycle must have some point in the starting/ending zone that becomes cheaper after each application. Let that point be u, then the cost of u is $\mathsf{cost}(u) = o + \sum_{x \in \mathbb{C}} r(x) \cdot (u(x) - \Delta Z)$. However, simply checking that some point becomes cheaper is a weak necessary condition, as this is very likely despite there not being a negative cycle. What we want to know is that no matter the initial cost plane values, applying the cycle will cause some point to be cheaper. Observe that this must be the case for negative cycles, because irrespective of the cost to reach that point, taking the negative concrete weight cycle will decrease the cost.

The cost plane can be defined by a pair (o, r) being the cost of the offset and the rates of the clocks, respectively. Recall the 4 operations needed on the cost-plane from Section 5.3:

$$
\begin{aligned}
(o, r) &\xrightarrow{\uparrow w, x} (o, r[x \mapsto w - \sum_{y \in \mathbf{C} \setminus \{x\}} r(y)]) && \text{(Delay)} \\
(o, r) &\xrightarrow{\Delta k, x} (o + r(x) \cdot k, r) && \text{(Continous Offset)} \\
(o, r) &\xrightarrow{\Delta w} (o + w, r) && \text{(Discrete Offset)} \\
(o, r) &\xrightarrow{\downarrow x, y} (o, r[y \mapsto r(y) + r(x), x \mapsto 0]) && \text{(Reset Relative).}
\end{aligned}
$$

These represent (in order), a delay on a facet of $x \in \mathbb{C}$ with a delay rate w; an application of a guard $x \geq n$ s.t. k is the amount that the lower constraint on x increases; add a discrete cost increment w to the offset; resetting a facet of $x \in \mathbb{C}$ that is relative to $y \in \mathbb{C}$.

We observe that the operations on the cost plane are affine, therefore, we can represent them as an affine transformation matrix. In this interpretation, the cost plane is a vector $\vec{c} = (r(x_0), r(x_1), \cdots, r(x_n), o, 1)$, where $x_i \in \mathbb{C}$ for all $0 \leq i \leq n$, o is the cost of the offset point, and 1 is the constant used for affine transformation. The affine transformation matrix for a cost plane operation is then a $(|\mathbb{C}| + 2) \times (|\mathbb{C}| + 2)$ matrix. The cost plane operations can then all be translated into affine transformation matrices. The transformation matrix for two clocks x and y is of the form

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r(x) \\ r(y) \\ o \\ 1 \end{bmatrix} = \begin{bmatrix} a \cdot r(x) + b \cdot r(y) + c \cdot o + d \\ e \cdot r(x) + f \cdot r(y) + g \cdot o + h \\ i \cdot r(x) + j \cdot r(y) + k \cdot o + l \\ 1 \end{bmatrix}.$$

The matrix representation of the operations, for a cost plane with 2 clocks, are

$$\xrightarrow{\uparrow w,x} \begin{bmatrix} 0 & -1 & 0 & w \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \xrightarrow{\Delta k,x} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ k & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \xrightarrow{\Delta w} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & w \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \xrightarrow{\downarrow x,y} \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

From a given symbolic cycle Π, we can extract the sequence of cost plane operations used. We can then compute the product of all of the matrices to find the matrix representing the application of the entire cycle. Let there be n cost-plane operations of Π and $M_1, M_2, \ldots, M_n$ be their respective matrix representations, in the order they appear, then

$$M_\Pi = \prod_{i=0}^{n-1} M_{n-i}$$

is the compound matrix representing the cost-plane transformation after applying the entire cycle.

Theorem 12. *Let Π be a symbolic cycle starting in $S = (\ell, Z, w)$, and let $S' = (\ell', Z', w')$ and $S'' = (\ell'', Z'', w'')$ be priced symbolic states s.t. $S \overset{\Pi^i}{\rightsquigarrow} S'$, $S \overset{\Pi^{i+1}}{\rightsquigarrow} S''$, and $Z = Z' = Z''$. Then, Π is a negative weight symbolic cycle only if $S' \not\sqsubseteq S''$, and if $i \geq N$, then Π is a negative weight symbolic cycle if $S' \not\sqsubseteq S''$, where N is the number of integer points in Z.*

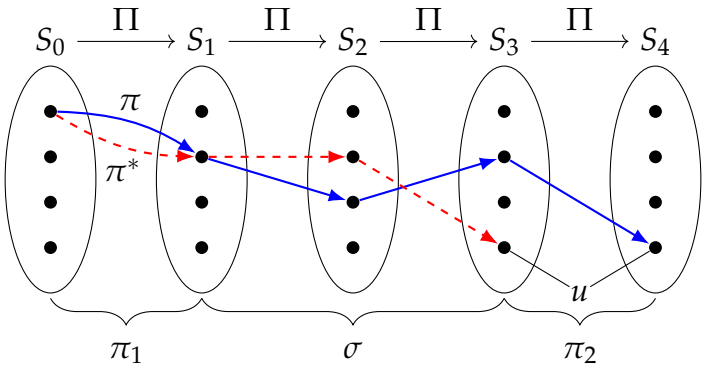
Proof. We first show that Π is a negative weight symbolic cycle only if $S' \not\sqsubseteq S''$ for all $i \in \mathbb{N}$. By definition of a negative weight symbolic cycle, there exists a negative weight concrete cycle $\pi \in \Pi$ going through a valuation $u \in Z \cap Z'$, where $w''(u) < w'(u)$. Thus, $S' \not\sqsubseteq S''$.

We now show that if $i \geq N$, then Π is a negative weight symbolic cycle if $S' \not\sqsubseteq S''$. First note that due to the precondition $Z = Z' = Z''$, this is the same as showing Π is a negative weight symbolic cycle if there exists a valuation u, s.t. $w''(u) < w'(u)$. We prove the contrapositive, i.e. if there is no negative cycle then $w'(u) \leq w''$. We abstract away whatever happens inside of the cycle and only concern ourselves with the states between applications, i.e. only the states $S_0, S_1, \ldots, S_{i+1}$ where $S_0 \overset{\Pi}{\rightsquigarrow} S_1 \overset{\Pi}{\rightsquigarrow} \cdots \overset{\Pi}{\rightsquigarrow} S_{i+1}$.

Firstly, recall that there always exists a minimum cost path between two integer valuations that is discrete (i.e. integer valued). Let $\pi \in \Pi^{i+1}$ be a discrete concrete path that obtains the minimum weight $w''(u)$. Since $i \geq N$, we know that π must contain a concrete cycle, because it needs to go through the same zone more times than it has integer points. Therefore, we write $\pi = \pi_1 \sigma \pi_2$, where σ is the concrete cycle that it contains. By the assumption that there is no negative weight concrete cycle, we have that $w(\sigma) > 0$. Now let k be the number of revolutions σ makes in Π, i.e. $\sigma \in \Pi^k$. From Corollary 4, we know that there is also a $\sigma^1 \in \Pi^1$, where $w(\sigma^1) = \frac{1}{k}\sigma$. By applying this $k-1$ times we can get a cycle $\sigma^{k-1} \in \Pi^{k-1}$ with $w(\sigma^{k-1}) = \frac{k-1}{k} w(\sigma)$. We then construct a new path $\pi^* = \pi_1 \sigma^{k-1} \pi_2$. Importantly, $\pi^* \in \Pi^i$ and will also end up in the valuation u, but in S'. The weight of π^* is $w(\pi^*) = w(\pi) - \frac{1}{k} w(\sigma)$, and since we know that $w(\sigma) > 0$, this leads to

the conclusion that $w(\pi^*) < w(\pi)$, and thus $w'(u) < w''(u)$, which proves the contrapositive statement.

Below is an illustrated example of this principle. Here we show the states as ellipses with their discrete points inside. The blue/solid arrows are the path π and the red/dashed arrows are the path π^*.

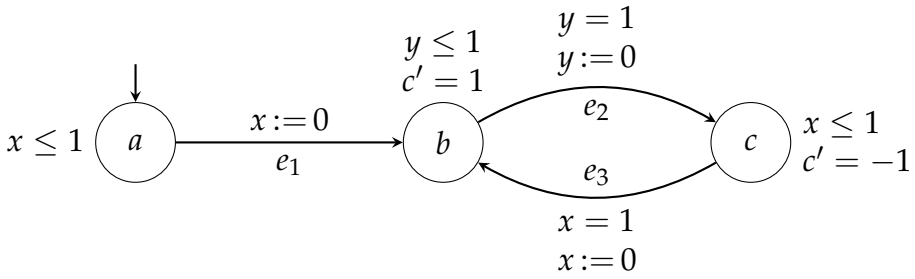


□

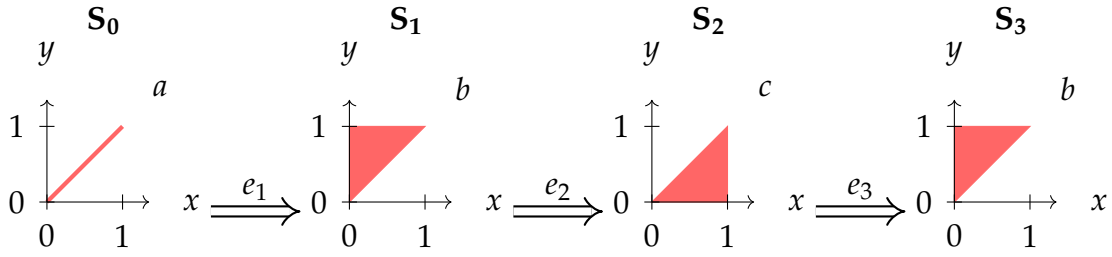
We can calculate the states S' and S'' by raising the compound transformation matrix to the power of N and $N+1$, respectively. Particularly, from a stabilised cycle $S^0 \overset{\Pi}{\rightsquigarrow} S^1$, where $S = (\ell, Z, c)$, then $S^N = (\ell, Z, M_\Pi^N c)$ and $S^{N+1} = (\ell, Z, M_\Pi^{N+1} c)$, where $M_\Pi^N c$ means the translation of the cost-plane into the affine vector representation, then multiplying by the matrix M_Π^N and finally translating it back to the normal cost-plane representation.

The complexity of calculating S^N and S^{N+1} is $\mathcal{O}((|\Pi| + \log N) \cdot |\mathbb{C}|^3)$. This comes from first creating the matrix M_Π, which can be done in $\mathcal{O}(|\Pi| \cdot |\mathbb{C}|^3)$ operations. Each transition in Π may have $\mathcal{O}(|\mathbb{C}|)$ cost-plane operations, however, these can be constructed efficiently in $\mathcal{O}(|\mathbb{C}|^3)$ operations, by constructing several cost-plane operation in the same matrix. Then the $|\Pi|$ cost-plane operation matrices are multiplied—assuming matrix multiplication in $\mathcal{O}(n^3)$ for a $n \times n$ matrix. Finally, raising M to the power of N can be done in $\log_2(N)$ matrix multiplications, meaning this can be done in $\mathcal{O}(\log N \cdot |\mathbb{C}|^3)$ operations.

***Example* 5** Consider the cost-reward timed automaton below.



An exploration of the state-space is shown below, where each arrow shows the result of applying Post_e and afterwards Post_ϵ for the edge e.



$\mathbf{S_0}$	$\mathbf{S_1}$	$\mathbf{S_2}$	$\mathbf{S_3}$
$r(x) = 0$	$r(x) = 1$	$r(x) = 1$	$r(x) = 3$
$r(y) = 0$	$r(y) = 0$	$r(y) = -2$	$r(y) = -2$
$c = 0$	$c = 0$	$c = 0$	$c = 1$

We can identify the symbolic cycle $S_1 \to S_2 \to S_3$. From this cycle, we can then extract the cost-plane operations

$$\xrightarrow{\Delta y,1} \xrightarrow{\uparrow -1,y} \xrightarrow{\Delta x,1} \xrightarrow{\uparrow 1,x}.$$

These represent, respectively, the guard $y \geq 1$ increasing the offset by 1 in the y direction; delaying from a y-facet with rate -1; the guard $x \geq 1$ increasing the offset by 1 in the x direction; and delaying from an x-facet with rate -1. From these, we can construct the affine transformation matrices (appearing in reverse

order)

$$M = \begin{bmatrix} 0 & -1 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 2 \\ -1 & 0 & 0 & -1 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

As a sanity check, we can see that if we apply this matrix on the cost-plane of the first state at b, i.e. $r(x) = 1, r(y) = 0, o = 0$, we get the cost-plane of the second state at b, as expected, i.e.

$$\begin{bmatrix} 1 & 0 & 0 & 2 \\ -1 & 0 & 0 & -1 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 3 \\ -2 \\ 1 \\ 1 \end{bmatrix}.$$

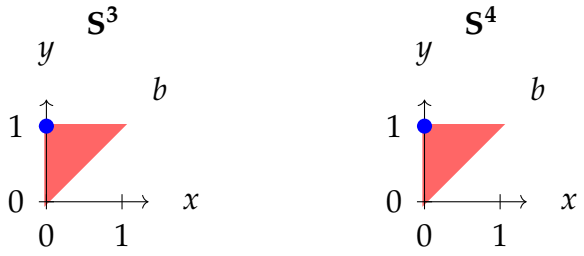
Now we calculate the matrices M^N and M^{N+1}, where, by counting the integer-valued points in S_1, $N = 3$,

$$M^3 = \begin{bmatrix} 1 & 0 & 0 & 6 \\ -1 & 0 & 0 & -5 \\ 1 & 1 & 1 & 2 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{and} \quad M^4 = \begin{bmatrix} 1 & 0 & 0 & 8 \\ -1 & 0 & 0 & -7 \\ 1 & 1 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Applying these matrices yields the cost-planes

$$c_3 = M^3 c = \begin{bmatrix} 1 & 0 & 0 & 6 \\ -1 & 0 & 0 & -5 \\ 1 & 1 & 1 & 2 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 7 \\ -6 \\ 3 \\ 1 \end{bmatrix} \quad \text{and} \quad c_4 = M^4 c = \begin{bmatrix} 9 \\ -8 \\ 4 \\ 1 \end{bmatrix} = \begin{bmatrix} 9 \\ -8 \\ 4 \\ 1 \end{bmatrix}.$$

We can now study the states S^3 and S^4, i.e. S_1 after applying the cycle 3 and 4 times.



$r(x) = 7$
$r(y) = -6$
$o = 3$

$r(x) = 9$
$r(y) = -8$
$o = 4$

We can now observe that the valuation $(0,1)$—the blue dot—has cost -3 in S^3 and -4 in S^4. Thus, $S^3 \not\sqsubseteq S^4$ and, by Theorem 12, there is a negative weight concrete cycle. By analysing the symbolic cycle, we can then surmise that $(0,1) \xrightarrow{e_2} (0,0) \xrightarrow{1} (1,1) \xrightarrow{e_3} (0,1)$ is a negative weight concrete cycle, with weight -1. □

7.4 Optimisations

In this section, we will describe two optimisation techniques for Algorithm 2.

7.4.1 Continuing With the Same Waiting List

We propose the idea of reusing the waiting list between calls of FIND-NEGATIVE-CYCLE of Algorithm 2, instead of re-exploring from the initial state anew. Concretely, we remove the reset of the waiting list on line 9 in Algorithm 2. This still guarantees that all negative cycles will be found.

Theorem 13. *Algorithm 2 without line 9 still terminates and returns a ratio-optimal cycle, if one exists, otherwise* NO CYCLE.

Proof. For termination, the only difference in Lemma 9 (on termination) is that Lemma 8 (on *Parent* comprising a directed in-tree) no longer holds, because there is not necessarily a single root of the parent. However, it is still a directed in-forest, i.e. consisting of one or more mutually disconnected in-trees. Therefore, the PARENT-NEGATIVE-CYCLE subroutine still terminates, as all paths in *Parent* lead to a (non-unique) root node.

The proof for soundness is the same as before (Lemma 10), so we now just need to prove completeness. We do this by showing that at each call to FIND-NEGATIVE-CYCLE, if there is a reachable negative cycle, then there is a state in *Waiting* that can reach the negative cycle. In the base case, this is true because by definition all reachable negative cycles can be reached from the initial state. Now, assuming all negative cycles are reachable from *Waiting* before the call, then we show that all negative cycles are still reachable after updating λ to λ' with $\lambda' < \lambda$. Exploration is only stopped by not passing the if-statement on line 14, when it is dominated by a previously seen state. A cycle is therefore only stopped being explored when it does not become any cheaper, which means it is a non-negative cycle. Observe that when $\lambda' < \lambda$ then any non-negative cycle in the λ-deducted automaton is also non-negative in the λ'-deducted automaton. Thus, any cycle that has been explored and concluded to be non-negative, will still be non-negative with the new, smaller λ'-value. Conclusively, all negative cycles will still be reachable. □

7.4.2 Pruning the Parent Pointer

We now propose to prune the states in the *Parent* pointer to reduce the memory usage. The idea is that we do not need to store all of the explored states in the *Parent* pointer; we can remove those where we have found a cheaper state because we might as well explore that state instead of the latter. Formally, we prune the *Parent* function s.t. $\forall S, S' \in \text{dom}(\mathit{Parent}).\ S \sqsubseteq S' \implies S = S'$. In order to maintain this invariant, we only insert a state S into the parent if $S' \not\sqsubseteq S$ for all $S' \in \text{dom}(\mathit{Parent})$, and after inserting S we remove from *Parent* all states S'' where $S \sqsubseteq S''$.

Specifically, we include the following lines, 14a and 14b, inserted into Algorithm 2:

```
13      forall S' ∈ Post_α(S) for all α ∈ E ∪ {ε} do
14          if ∀S'' ∈ dom(Parent). S'' ⋢ S' then
14a             forall S'' ∈ dom(Parent) where S' ⊑ S'' do
14b                 undefine Parent(S'');
15              Parent(S') := (S, α);
16              insert S' into Waiting;
17              if PARENT-NEGATIVE-CYCLE(S --α--> S') returns cycle C then
18                  return C;
```

They are inserted in the FIND-NEGATIVE-CYCLE subroutine after checking that the successor S' of S (the expanded state from the *Waiting* list) is not dominated in *Parent*. It undefines all $S'' \in \mathrm{dom}(\mathit{Parent})$ that S' dominates, i.e. it removes $\mathit{Parent}(S'')$ s.t. now $S'' \notin \mathrm{dom}(\mathit{Parent})$.

We leave the correctness of this optimisation as a conjecture, as we have not yet been able to find a counter-example nor a proof of its correctness. The main conundrum is whether this pruning can cause an infinite sequence of pruning in the *Parent* s.t. no cycle is left intact whenever PARENT-NEGATIVE-CYCLE is called.

Conjecture 14. *Adding the lines 14a and 14b to Algorithm 2, which prune the Parent function, preserves its correctness.*

8 Experimental Evaluation

In this section, we present our experimental evaluation of the approaches presented in the paper. First, the models used for the experiments are presented. Next, we discuss some problems with the BDD algorithm from Section 4 that makes it perform poorly, and why it is not present in the remaining results. Lastly, the results of the evaluation of symbolic λ-deduction from Section 7 and corner-point minimal cycle ratio solving with Howard's from Section 3.2 are presented and discussed. As a shorthand in the figures and tables, we will refer to the 3 algorithms as *CP-BDD* for corner-point BDD, *S-λD* for symbolic λ-deduction, and

CP-MCR for the ground truth corner-point minimum cycle ratio technique using Howard's algorithm.

8.1 Methodology

The algorithms have been implemented on top of UPPAAL 4.1 [10]. The data from the experiments and the scripts for generating the figures and tables are available on Github[1].

All experiments were conducted on a compute cluster with an AMD EPYC 7551 32-core Processor running Debian with a Linux 5.8 kernel. All experiments were conducted with a 30-minute time limit and 10GB memory limit. The experiments include three different scheduling problems: surveillance, job scheduling and strandvejen.

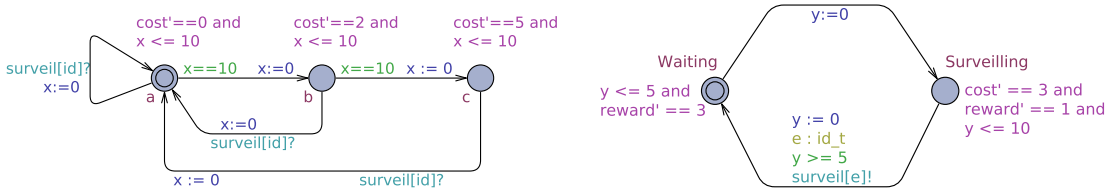
The problems are modelled in UPPAAL as networks of timed automata. We have extended the UPPAAL model parsing module to allow for both costs and rewards. In [10], they give a thorough description of the syntax they use. A network of timed automata consists of template modules, which can be instantiated into actual automata, allowing also parameters for constants. A regular cost-reward weighted timed automaton can be constructed as the cross-product of the individual automata. Additionally, synchronisation between instances is achieved by synchronisation channels. For a synchronisation channel c, the synchronisation is shown on edges as $c!$ and $c?$, and it means that a $\xrightarrow{c!}$ edge can only be fired in conjunction with a $\xrightarrow{c?}$ edge. Additionally, these can be represented as arrays of channels, s.t. the i'th channel of a channel array a is $a[i]$. The initial state is marked as a double circle.

8.1.1 Surveillance

The surveillance scheduling problem has one or more agents that must surveil different places. Figure 3 shows the two templates of the model. If a place has not been surveilled for 10 units of time, the cost of the place is increased. The agents can either be waiting or surveilling, with the former giving more reward.

[1] `https://github.com/Slorup/uppaal_roc_artefact`

At some point the agent can surveil a place which takes between 5 and 10 units of time and resets the cost of the place to its initial value.



(a) Place. (b) Agent.

Fig. 3: Surveillance scheduling problem.

8.1.2 Job Scheduling

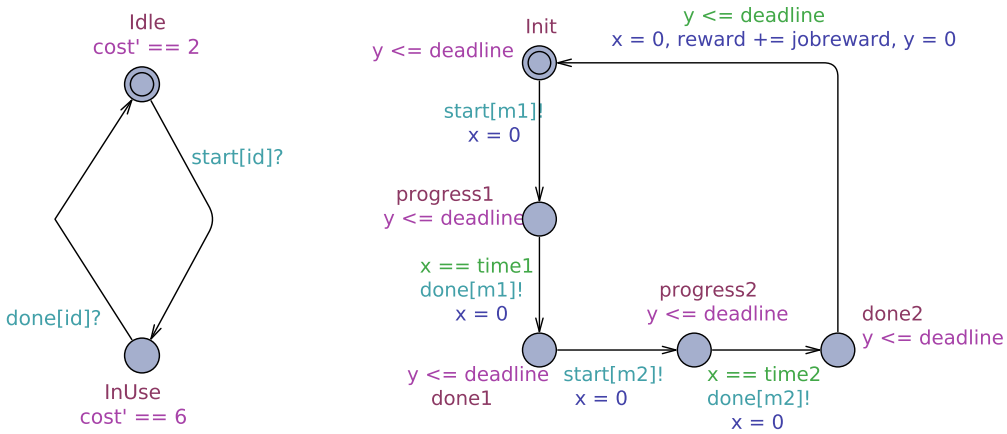
Our job scheduling problem is a small extension to the scheduling problem presented in [29]. The model consists of two templates, machines and jobs, and concerns with scheduling jobs on specific machines. The templates are shown in Figure 4. The machine template only consists of two states that indicate whether it is in use or not. The cost of the machine is higher if it is currently in use. Each job needs to serially perform two tasks on two specific machines, `m1` and `m2`. The time to perform the tasks are specified by `time1` and `time2`, respectively. Each job needs to be repeated at least every `deadline` units of time, and completing the task results in `jobreward` amount of reward.

8.1.3 Strandvejen

Strandvejen is a scheduling problem which concerns volunteers balancing their university work while maintaining a supply of cold drinks in the refrigerators. The UPPAAL model consists of three templates: Volunteer, Consumer and Refrigerator. The templates are shown in Figure 5.

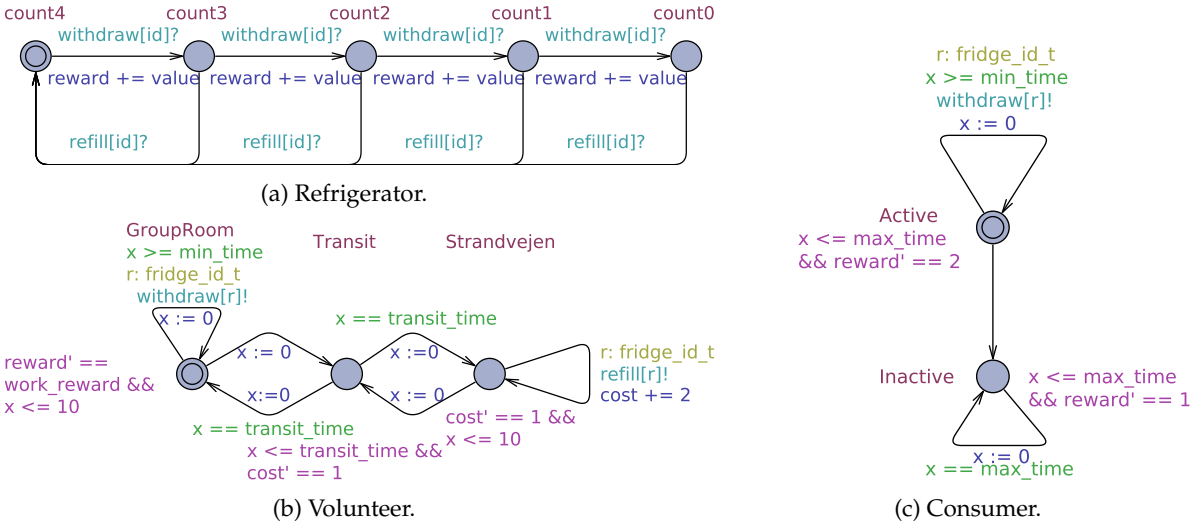
The refrigerator can store up to 4 items, which can be withdrawn by the consumers and volunteers. Withdrawing an item gives a reward specified by `value`. It is also possible for a volunteer to fully refill a refrigerator.

The consumers have two possible states. They start in the active state, which indicates they are a part of the social group at the university. Here they are able



(a) Machine. (b) Job.

Fig. 4: Job scheduling problem.



(a) Refrigerator.

(b) Volunteer.

(c) Consumer.

Fig. 5: Strandvejen scheduling problem.

to withdraw items from the refrigerator. If they do not receive an item within the interval of [min_time, max_time], they leave the social group, which results in a lower reward rate and them no longer being able to withdraw items.

The volunteers are the core of the scheduling problem. They earn reward by working in their group room, where they are also able to withdraw items every min_time units of time. Every so often, they can travel to Strandvejen to refill the refrigerators. Travelling time is based on the volunteer's distance to Strandvejen, which is indicated by transit_time. Traversing and refilling the refrigerators adds to the cost.

8.1.4 Benchmark Suite

For our experiments, we created a collective benchmark with instances from the three scheduling problems presented in Section 8.1. This benchmark includes 7 instances of the job scheduling problem, 9 instances of the strandvejen problem and 14 instances of the surveillance scheduling problem, totalling 30 instances. The job scheduling instances vary in the number of jobs and machines, the time to execute the jobs on each machine is 3 units of time and the deadline for each job is between 12 and 15 units of time. The number of components is indicated in the instance names, i.e. the instance name job_j3_m2 is run with 3 jobs and 2 machines. In Strandvejen, the highest constant is 13 and all templates are scaled. The surveillance instances are scaled in the number of agents and places, and the constants remain as depicted in Figure 3.

8.2 BDD Approach

In Section 4, we described an approach to solve the cost-reward optimal cycle problem using BDDs. However, during initial experiments, we found that it is extremely slow, and takes too long on even very small problems.

One of the main culprits is the BDD multiplication and BDD addition of the many cost and reward variables. Here we tried multiple things to speed up the operations such as changing the variable ordering and doing conjunctions midway to avoid large intermediate BDDs. This did give improvements, but the operations remained too heavy.

Another problem with the approach was the compounding of the transition relation. This step in itself showed to perform slowly due to the large number of variables. To improve on the compounding of the transition relation, we developed a reduction method, where we restrict the relation to only deal with a subset of states that has the property that for all infinite executions, at least one of them is visited infinitely often in that execution. This technique was a great improvement, but unfortunately, it was not enough to render the BDD algorithm remotely applicable.

In the rest of this section, we will not include the BDD algorithm as severely under-performs compared to *CP-MCR* and symbolic λ-deduction.

8.3 Results

This section contains the results of our experimental evaluation. First, we present the results of symbolic λ-deduction and its various optimisations. Afterwards, we compare symbolic λ-deduction to *CP-MCR* . Lastly, we analyse how the algorithms perform when the clock space is up-scaled.

8.3.1 Symbolic λ-deduction Optimisations

We now present and compare the different versions of symbolic λ-deduction. This includes the idea in Section 7.4.1 of reusing the same waiting list through iterations, the pruning of the parent pointer from Section 7.4.2, and the transformation matrix technique from Section 7.3 to efficiently determine if a symbolic cycle is negative.

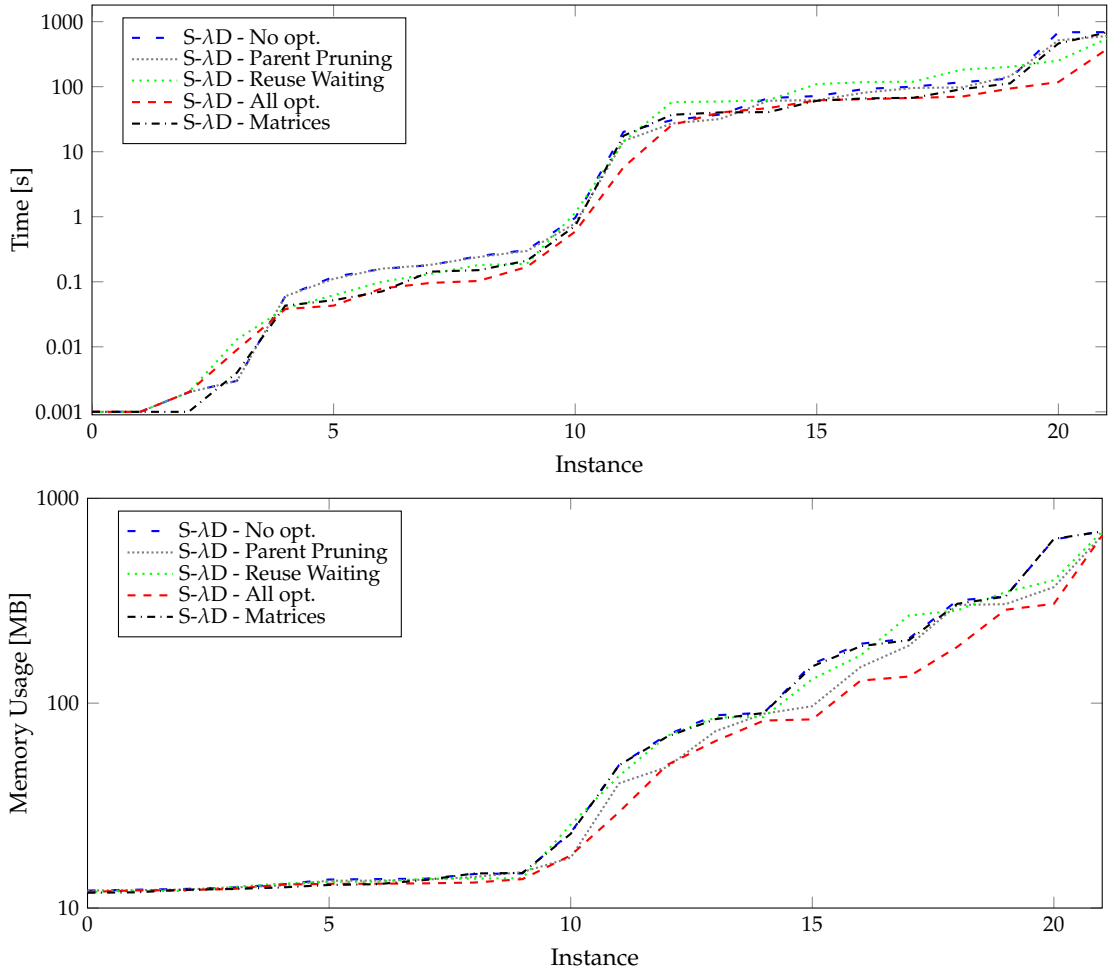


Fig. 6: Cactus plots of the time and memory it took for symbolic λ-deduction and its variants to prove the optimal ratio of the instances. All variants solved 22 of the 30 instances.

Figure 6 shows the time and memory usage of the different versions of symbolic λ-deduction, respectively. This includes a variant without any optimisations, three variants for each optimisation enabled individually, and a variant with all three optimisation techniques enabled together. In the two figures, the instances are ordered by time and memory usage, respectively, for each version of the algorithm. If a variant did not finish on an instance, then the data point is not showed in the cactus plot.

All variants of symbolic λ-deduction terminate with the optimal cycle ratio in 22 of the 30 instances. Pruning of the parent pointers shows a tiny increase in speed at median, but at median reduces the memory usage by 2.1%, however, in the best cases, it achieves a reduction of 42.1%. In the majority of the instances, the use of transformation matrices improves the performance of symbolic λ-deduction and it has a median time reduction of 12.9%.

Reusing the waiting list through iterations shows a large variance in performance. On some instances, it decreases the total time spent by up to 67%, while on other instances it triples the total time. The median time improvement of enabling the technique is exactly 0%. A similar story is seen for memory usage, ranging from 55.0% memory reductions on some instances while increasing the memory usage with up to 71.5% in other cases, with a median memory reduction of 1.4%. This large variance is likely due to the technique changing the order of which the symbolic states are expanded from the waiting list. The problems instances we have run on do not present a lot of gain from this optimisation, because in many cases it is always possible to reach back to the initial state, which means there will not be much opportunity for pruning.

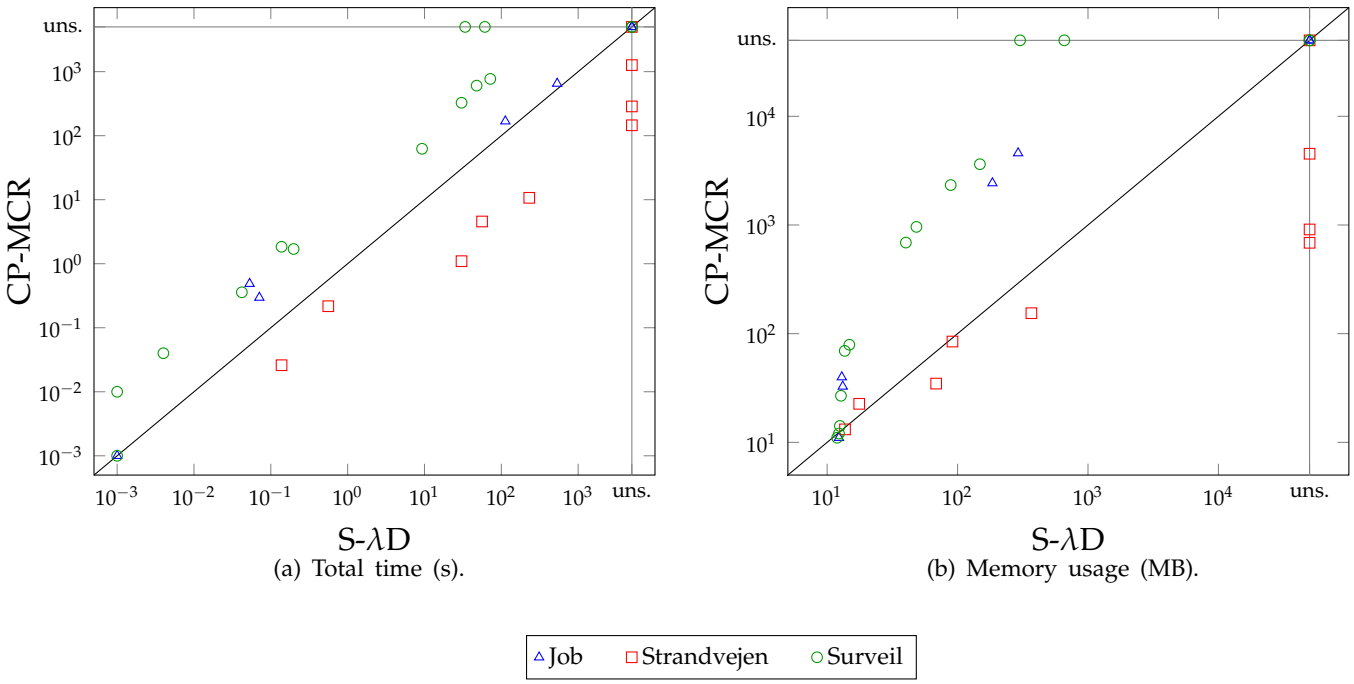
The version with all optimisations techniques enabled shows that the techniques can be combined into an improved version of the algorithm with great effect. Here we get a 38.2% median decrease in time and a 6.5% decrease in memory usage compared to the version without any optimisations.

8.3.2 Comparison Between Symbolic λ-deduction and *CP-MCR*

In this section, we compare the results of *CP-MCR* and symbolic λ-deduction on our benchmark. First, we present the results of scaling the number of components in our model, and afterwards by scaling the size of the constants in the models. We consider only the version of symbolic λ-deduction with parent pruning and transformation matrices enabled, and leave the technique of reusing the waiting list disabled due to its chaotic performance.

Figure 7 show two scatter plots comparing the time and memory usage for *CP-MCR* and symbolic λ-deduction. A table with all the results is shown in Table 1. On two instances, symbolic λ-deduction terminates with the optimal value

where Concrete-MCR does not find any cycle. Contrarily, there are three instances on which Concrete-MCR terminates where symbolic λ-deduction did not. However, on these instances, symbolic λ-deduction was able to find a near-optimal ratio before running out of time. We also note that on *surveil_a1_p7*, symbolic λ-deduction finds that there is no cycle as a single agent cannot surveil seven places within the time constraints.



(a) Total time (s).

(b) Memory usage (MB).

Fig. 7: Scatter plots comparing the performance of symbolic λ-deduction and *CP-MCR* both in total time and memory usage.

On the surveillance and job scheduling instances, symbolic λ-deduction clearly performs better in both time and memory. In contrast, *CP-MCR* outperforms symbolic λ-deduction in time on all strandvejen instances (except for `strdvj_f1_v2_c3` where neither algorithm terminates). At median, the time of *CP-MCR* is 4.17 times greater than symbolic λ-deduction and the memory usage is 2.10 times greater. Notice that symbolic λ-deduction always found a cycle within time limit. This is an advantage of symbolic λ-deduction as it immedi-

	CP-MCR				S-λD			
Instance	Time (s)	Memory (MB)	Ratio	RatioTime (s)	Time (s)	Memory (MB)	Ratio	RatioTime (s)
job_m2_j1	0.001	**11.024**	**48.0000**	0.001	**0.000**	12.320	**48.0000**	**0.000**
job_m2_j2	0.296	32.664	**36.0000**	0.235	**0.071**	**13.172**	**36.0000**	**0.027**
job_m2_j3	168.044	2427.576	**30.0000**	149.242	**112.782**	**185.044**	**30.0000**	**0.023**
job_m2_j4	-	OOM	-	-	OOT	-	**24.0000**	**0.076**
job_m3_j2	0.487	39.800	**42.0000**	0.350	**0.053**	**12.952**	**42.0000**	**0.008**
job_m3_j3	651.253	4583.680	**42.0000**	623.498	**533.127**	**291.316**	**42.0000**	**0.492**
job_m3_j4	-	OOM	-	-	OOT	-	**37.5000**	**142.466**
strdvj_f1_v1_c1	**0.026**	**13.164**	**0.0814**	**0.026**	0.138	13.712	**0.0814**	0.099
strdvj_f1_v1_c2	**1.096**	**34.728**	**0.0598**	**1.047**	30.360	68.548	**0.0598**	18.246
strdvj_f1_v1_c3	**145.386**	**685.652**	**0.0486**	**138.633**	OOT	-	0.1034	531.575
strdvj_f1_v2_c1	**4.566**	**84.644**	**0.0980**	**3.504**	56.174	91.396	**0.0980**	25.440
strdvj_f1_v2_c2	**286.337**	**909.528**	**0.0796**	**253.417**	OOT	-	0.0821	1514.877
strdvj_f1_v2_c3	OOT	-	-	-	OOT	-	**0.0843**	**335.026**
strdvj_f2_v1_c1	**0.217**	22.616	**0.0533**	**0.194**	0.560	**17.616**	**0.0533**	0.440
strdvj_f2_v1_c2	**10.699**	**154.328**	**0.0398**	**9.998**	231.075	368.192	**0.0398**	169.301
strdvj_f2_v1_c3	**1265.142**	**4524.200**	**0.0329**	1195.115	OOT	-	0.1034	**524.100**
surveil_a1_p1	**0.000**	**11.024**	**0.7500**	**0.000**	**0.000**	11.940	**0.7500**	**0.000**
surveil_a1_p2	0.040	14.152	**1.7500**	0.037	**0.004**	**12.520**	**1.7500**	**0.003**
surveil_a1_p3	1.699	79.096	**3.5000**	1.482	**0.198**	**14.800**	**3.5000**	**0.009**
surveil_a1_p4	62.441	688.844	**6.8000**	56.719	**9.302**	**40.140**	**6.8000**	**0.005**
surveil_a1_p5	604.830	3624.020	**10.6250**	582.922	**47.719**	**148.308**	**10.6250**	**0.033**
surveil_a1_p6	OOT	-	-	-	**33.827**	**301.364**	**17.0000**	**3.175**
surveil_a1_p7	OOT	-	-	-	**61.034**	**658.200**	**No sol.**	-
surveil_a2_p1	0.010	**12.052**	**0.7500**	0.010	**0.001**	12.320	**0.7500**	**0.000**
surveil_a2_p2	1.841	69.432	**0.7500**	1.639	**0.138**	**13.648**	**0.7500**	**0.001**
surveil_a2_p3	770.296	2330.464	**1.2500**	743.820	**71.859**	**88.572**	**1.2500**	**0.005**
surveil_a2_p4	OOT	-	-	-	OOT	-	**1.7500**	**0.183**
surveil_a3_p1	0.358	26.828	**0.7500**	0.299	**0.042**	**12.764**	**0.7500**	**0.001**
surveil_a3_p2	326.750	960.452	**0.7500**	319.026	**30.381**	**48.384**	**0.7500**	**0.002**
surveil_a3_p3	OOT	-	-	-	OOT	-	**0.7500**	**0.003**

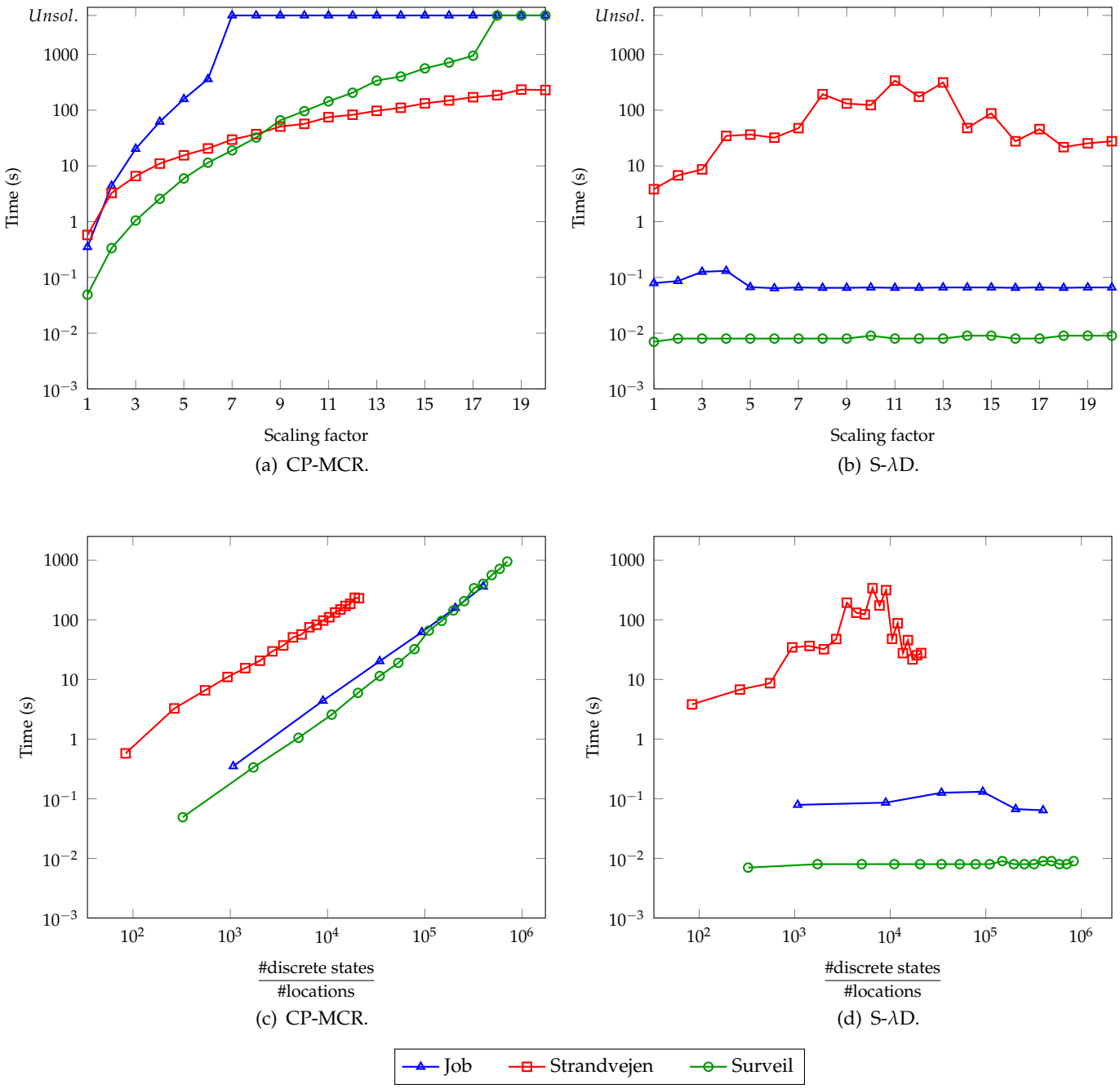
Table 1: Results comparison between symbolic λ-deduction and *CP-MCR* . Run with a 30-minute time limit and 10GB memory limit—OOT and OOM mean that these were exceeded, respectively. *Ratio* is the best ratio that the algorithm found. Note that this may not be optimal in the case that the algorithm did not finish due to exceeding the time or memory limit. *RatioTime* is the time it took for the algorithm to find *Ratio*.

ately starts searching for cycles. *CP-MCR* also incrementally improves on the best found ratio, however before being able to do so, it needs to spend a lot of time expanding the entire state-space.

8.3.3 Scaling the Size of the Clock Space

The previous results showed how the algorithms scaled in the number of locations, components and clocks on our three scheduling models. The constants in these models are relatively small with the highest being 15. We now study the performance of the algorithms when scaling the size of the constants.

Figure 8 shows the run-time of *CP-MCR* and symbolic λ-deduction when scaling all the constants in the model. Additionally, after scaling we add 1 to the constants to remove large common divisors, which means that the problem can no longer be reduced to the original by simply dividing by the scaling factor. The experiment shows that *CP-MCR* suffers an exponential increase in time w.r.t. to the scaling factor. In contrast, this barely affects the time of symbolic λ-deduction. This is not surprising as scaling can result in an exponential increase in the number of discrete states in the underlying transition system, which has a direct proportional increase in the number of vertices in the doubly weighted graph of *CP-MCR* . From Figure 8c, we can see how *CP-MCR* scales linearly w.r.t. the ratio of discrete states to locations. Contrarily, scaling the size of the constants does not have any impact on the number of symbolic states and instead only increases the size of the zones. However, adding 1 to the constants after scaling sometimes changes the solution space, thus we also see some fluctuations for symbolic λ-deduction.



(a) CP-MCR.

(b) S-λD.

(c) CP-MCR.

(d) S-λD.

Fig. 8: Figure 8a and 8b shows the time of *CP-MCR* and symbolic λ-deduction on `job_m2_j2`, `strandvejen_f1_v1_c2` and `surveil_a1_p2` when scaling all constants in the models with a factor and afterwards adding 1. Similarly, Figure 8c and 8d shows the time as a function of the average number of discrete states per location (by scaling constants as before).

9 Conclusion

In this report, we studied two novel symbolic approaches for finding ratio-optimal cycles in cost-reward priced timed automata (CRTA), namely, using binary decision diagrams (BDDs) and using priced zones. We implemented both algorithms in UPPAAL, and conducted an experimental evaluation.

We showed how to encode the transition relation of the underlying discrete transition system as a BDD, and how it can be used to find optimal cycles. We compute the closure of this relation, which is then used to determine the states that are in an optimal cycle. Finally, we construct a transition relation consisting only of transitions that are in an optimal cycle and proved that any cycle in this transition relation is optimal. Unfortunately, the experimental evaluation unveiled that this algorithm has atrocious performance.

As our main contribution, we developed symbolic λ-deduction: an algorithm that merges the cost and reward of edges and rates into a single-priced timed automaton, circumventing some significant theoretical barriers that arise in the doubly priced setting. It incrementally improves the best found cycle, searching with priced clock zones. We proved that it converges on an optimal solution by proving its correctness and termination. Additionally, we showed how the change of the cost function of a symbolic state, from traversing a path (or cycle), can be represented as an affine transformation matrix, allowing us to efficiently determine whether a symbolic cycle is a better solution. The experimental evaluation for this algorithm was far more prosperous: it showed that symbolic λ-deduction outperforms the concrete MCR algorithm using Howard's on some problems. We also experimentally validated the symbolic strength of being completely unaffected by large clock values, where the concrete approach suffers an exponential slowdown. For example, the median for symbolic λ-deduction is ~3.5 times faster than the median for the concrete approach over all experiments. On the other hand, symbolic λ-deduction likely performs poorer than the concrete algorithm if the clock space is small and the different cycles are plenty.

In future work, we believe there can be a lot of gain in using heuristics to decide the order of exploring states in symbolic λ-deduction. We also believe there

can be better ways to avoid information loss when finding a new λ-value. We have already explored this, by reusing the waiting list, however, we believe there can be more benefit by also somehow keeping the parent pointer information.

References

[1] Nicklas Slorup Johansen, Kristian Ødum Nielsen, and Rasmus Tollund. "Towards Efficiently Computing Optimal Infinite Cycles in Cost-Reward Timed Automata". In: (2023).

[2] Gerd Behrmann et al. "Minimum-Cost Reachability for Priced Timed Automata". In: *Hybrid Systems: Computation and Control, 4th International Workshop, HSCC 2001, Rome, Italy, March 28-30, 2001, Proceedings*. Ed. by Maria Domenica Di Benedetto and Alberto L. Sangiovanni-Vincentelli. Vol. 2034. Lecture Notes in Computer Science. Springer, 2001, pp. 147–161. DOI: 10.1007/3-540-45351-2_15.

[3] Rajeev Alur, Salvatore La Torre, and George J. Pappas. "Optimal paths in weighted timed automata". In: *Theor. Comput. Sci.* 318.3 (2004), pp. 297–322. DOI: 10.1016/j.tcs.2003.10.038.

[4] Thomas Hune, Kim Guldstrand Larsen, and Paul Pettersson. "Guided Synthesis of Control Programs Using UPPAAL". In: *Nord. J. Comput.* 8.1 (2001), pp. 43–64.

[5] Ansgar Fehnker. "Scheduling a Steel Plant with Timed Automata". In: *6th International Workshop on Real-Time Computing and Applications Symposium (RTCSA '99), 13-16 December 1999, Hong Kong, China*. IEEE Computer Society, 1999, pp. 280–286. DOI: 10.1109/RTCSA.1999.811256.

[6] Peter Niebert and Sergio Yovine. "Computing Efficient Operation Schemes for Chemical Plants in Multi-batch Mode". In: *Eur. J. Control* 7.4 (2001), pp. 440–454. DOI: 10.3166/ejc.7.440-454.

[7] Ed Brinksma, Angelika Mader, and Ansgar Fehnker. "Verification and optimization of a PLC control schedule". In: *Int. J. Softw. Tools Technol. Transf.* 4.1 (2002), pp. 21–33. DOI: 10.1007/s10009-002-0079-0.

[8] Patricia Bouyer, Ed Brinksma, and Kim Guldstrand Larsen. "Staying Alive as Cheaply as Possible". In: *Hybrid Systems: Computation and Control, 7th International Workshop, HSCC 2004, Philadelphia, PA, USA, March 25-27, 2004, Proceedings*. Ed. by Rajeev Alur and George J. Pappas. Vol. 2993. Lecture Notes in Computer Science. Springer, 2004, pp. 203–218. DOI: 10.1007/978-3-540-24743-2_14.

[9] Rajeev Alur and David Dill. "Automata for modeling real-time systems". In: *Automata, Languages and Programming*. Ed. by Michael S. Paterson. Berlin, Heidelberg: Springer Berlin Heidelberg, 1990, pp. 322–335.

[10] Kim G Larsen, Paul Pettersson, and Wang Yi. "UPPAAL in a nutshell". In: *International journal on software tools for technology transfer* 1.1 (1997), pp. 134–152.

[11] Conrado Daws et al. "The tool KRONOS". In: *International Hybrid Systems Workshop*. Springer. 1995, pp. 208–219.

[12] Gerd Behrmann et al. "Efficient guiding towards cost-optimality in UPPAAL". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2001, pp. 174–188.

[13] Ulrich Fahrenberg and Kim Guldstrand Larsen. "Discount-Optimal Infinite Runs in Priced Timed Automata". In: *Joint Proceedings of the 8th, 9th, and 10th International Workshops on Verification of Infinite-State Systems, INFINITY 2006 / 2007 / 2008*. Ed. by Peter Habermehl and Tomás Vojnar. Vol. 239. Electronic Notes in Theoretical Computer Science. Elsevier, 2009, pp. 179–191. DOI: 10.1016/j.entcs.2009.05.039.

[14] Lewis Tolonen, Tim French, and Mark Reynolds. "Population Based Methods for Optimising Infinite Behaviours of Timed Automata". In: *25th International Symposium on Temporal Representation and Reasoning (TIME 2018)*. Ed. by Natasha Alechina, Kjetil Nørvåg, and Wojciech Penczek. Vol. 120. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, 22:1–22:22. ISBN: 978-3-95977-089-7. DOI: 10.4230/LIPIcs.TIME.2018.22.

[15] Alexandre David et al. “Optimal Infinite Runs in One-Clock Priced Timed Automata”. English. In: Annual Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS); null ; Conference date: 14-10-2011 Through 16-10-2011. 2011.

[16] E.L. Lawler. “Combinatorial Optimization: Networks and Matroids”. In: Holt, Rinehart and Winston, 1976. Chap. 13.

[17] Patricia Bouyer et al. “Infinite Runs in Weighted Timed Automata with Energy Constraints”. In: *Formal Modeling and Analysis of Timed Systems, 6th International Conference, FORMATS 2008, Saint Malo, France, September 15-17, 2008. Proceedings*. Ed. by Franck Cassez and Claude Jard. Vol. 5215. Lecture Notes in Computer Science. Springer, 2008, pp. 33–47. DOI: 10.1007/978-3-540-85778-5_4.

[18] Patricia Bouyer et al. “Average-energy games”. In: *Acta Informatica* 55.2 (2018), pp. 91–127. DOI: 10.1007/s00236-016-0274-1.

[19] Patricia Bouyer, Kim G. Larsen, and Nicolas Markey. “Lower-bound-constrained runs in weighted timed automata”. In: *Perform. Evaluation* 73 (2014), pp. 91–109. DOI: 10.1016/j.peva.2013.11.002.

[20] Kim Larsen et al. “As Cheap as Possible: Efficient Cost-Optimal Reachability for Priced Timed Automata”. In: vol. 2102. July 2001, pp. 493–505. ISBN: 978-3-540-42345-4. DOI: 10.1007/3-540-44585-4_47.

[21] Ali Dasdan. “Experimental Analysis of the Fastest Optimum Cycle Ratio and Mean Algorithms”. In: *ACM Trans. Des. Autom. Electron. Syst.* 9.4 (Oct. 2004), pp. 385–418. ISSN: 1084-4309. DOI: 10.1145/1027084.1027085.

[22] Ali Dasdan, Sandy S Irani, and Rajesh K Gupta. “Efficient algorithms for optimum cycle mean and optimum cost to time ratio problems”. In: *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference*. 1999, pp. 37–42.

[23] Jean Cochet-Terrasson et al. “Numerical Computation of Spectral Elements in Max-Plus Algebra”. In: *IFAC Proceedings Volumes* 31.18 (1998). 5th IFAC Conference on System Structure and Control 1998 (SSC’98), Nantes, France,

8-10 July, pp. 667–674. ISSN: 1474-6670. DOI: https://doi.org/10.1016/S1474-6670(17)42067-2.

[24] Ramón Béjar, César Fernández, and Francesc Guitart. "Encoding Basic Arithmetic Operations for SAT-Solvers". In: *Artificial Intelligence Research and Development - Proceedings of the 13th International Conference of the Catalan Association for Artificial Intelligence, l'Espluga de Francolí, Tarragona, Spain, 20-22 October 2010*. Ed. by René Alquézar, Antonio Moreno, and Josep Aguilar-Martin. Vol. 210. Frontiers in Artificial Intelligence and Applications. IOS Press, 2010, pp. 239–248. DOI: 10.3233/978-1-60750-643-0-239. URL: https://doi.org/10.3233/978-1-60750-643-0-239.

[25] Kim Guldstrand Larsen et al. "As Cheap as Possible: Efficient Cost-Optimal Reachability for Priced Timed Automata". In: *Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings*. Ed. by Gérard Berry, Hubert Comon, and Alain Finkel. Vol. 2102. Lecture Notes in Computer Science. Springer, 2001, pp. 493–505. DOI: 10.1007/3-540-44585-4_47.

[26] A. Charnes and W. W. Cooper. "Programming with linear fractional functionals". In: *Naval Research Logistics Quarterly* 9.3-4 (1962), pp. 181–186. DOI: https://doi.org/10.1002/nav.3800090303.

[27] Michel Gondran and Michel Minoux. "Appendix 5". In: *Graphs and algorithms*. John Wiley & Sons, 1995.

[28] Joseph B Kruskal. "The theory of well-quasi-ordering: A frequently discovered concept". In: *Journal of Combinatorial Theory, Series A* 13.3 (1972), pp. 297–305. ISSN: 0097-3165. DOI: https://doi.org/10.1016/0097-3165(72)90063-5. URL: https://www.sciencedirect.com/science/article/pii/0097316572900635.

[29] Gerd Behrmann, Kim G Larsen, and Jacob I Rasmussen. "Optimal scheduling using priced timed automata". In: *ACM SIGMETRICS Performance Evaluation Review* 32.4 (2005), pp. 34–40.