Population Based Methods for Optimising Infinite Behaviours of Timed Automata

Lewis Tolonen
The University of Western Australia

Tim French
The University of Western Australia
tim.french@uwa.edu.au

Mark Reynolds
The University of Western Australia
mark.reynolds@uwa.edu.au

Abstract

Timed automata are powerful models for the analysis of real time systems. The optimal infinite scheduling problem for double-priced timed automata is concerned with finding infinite runs of a system whose long term cost to reward ratio is minimal. Due to the state-space explosion occurring when discretising a timed automaton, exact computation of the optimal infinite ratio is infeasible. This paper describes the implementation and evaluation of ant colony optimisation for approximating the optimal schedule for a given double-priced timed automaton. The application of ant colony optimisation to the corner-point abstraction of the automaton proved generally less effective than a random method. The best found optimisation method was obtained by formulating the choice of time delays in a cycle of the automaton as a linear program and utilizing ant colony optimisation in order to determine a sequence of profitable discrete transitions comprising an infinite behaviour.

2012 ACM Subject Classification Theory of computation → Automata over infinite objects

Keywords and phrases Timed Automata, Heuristic Search, Ant Colony Optimisation

Digital Object Identifier 10.4230/LIPIcs.TIME.2018.22

1 Introduction

Many systems exhibit behaviour that is dependent not only on the ordering of a sequence of events, but also the precise time at which these events occur. These are known as *real time systems*, and include such things as industrial plants which may carry out a set of sub-processes each of which requires different resources for a specific duration of time, or communication protocols where messages are sent and received with the restriction that at any given time the communication medium can only be used by one of these actions. The analysis of real time systems is important for ensuring the behaviour of these systems meets the required specifications. In particular, for cases such as industrial plants, it is also desirable that the long term behaviour of these systems is profitable, that is the cost to reward ratio of the system behaviour is minimised. Timed automata are powerful models for performing such analysis of these systems. A timed automaton models a real time system as a set of states with transitions between them, alongside a set of real valued clock variables that count up uniformly in real time. The taking of transitions is dependent on the values of the clocks, and can reset some of the clocks when taken. The model has been extended to be able to describe other quantitative aspects of systems behaviours, such as costs and rewards.

The optimal infinite scheduling problem for timed automata is concerned with finding the behaviour of a system that minimises the long term ratio of the operating cost to the obtained rewards. Solving this problem is particularly useful when modelling an industrial process, as the profitable operation of such a system is typically a key goal in its design. However, due to the state-space explosion that can occur when modelling all potential behaviours of a real time system, finding the exact solution for the optimal infinite scheduling problem generally proves infeasible. Instead, this paper focuses on developing methods for finding approximate solutions to the problem. In particular, the use of population based metaheuristics such as ant colony optimisation as a means of efficiently exploring the candidate behaviours of a real time system. By expressing the timed component of the optimal infinite scheduling problem as a linear program that can be exactly solved relatively efficiently, these techniques give an effective method for approximating the optimal behaviour of a given system.

2 Preliminaries

In this section we will present the preliminary definitions required for this paper.

2.1 Timed Automata

Clocks, Valuations, and Constraints. Before formally defining a timed automaton it is necessary to introduce the *clock variables* on which their behaviour depends. A set of clocks $\mathcal{X}$ is a finite set of variables $\{x_0, \ldots, x_n\}$ taking values in the non-negative real numbers $\mathbb{R}_{\geq 0}$. A clock *valuation* is a function $v : \mathcal{X} \to \mathbb{R}_{\geq 0}$ assigning to each clock its current value. $\mathbf{0}$ denotes the valuation that assigns 0 to all clocks. The *reset operation* on a clock valuation v by a subset X of $\mathcal{X}$, denoted $v[X := 0]$, gives the valuation v' where the clocks in X are set to 0, while the other clocks remain unchanged. For a real number δ, the *delay* of v by δ, denoted $v + \delta$ gives the valuation where δ has been added to the value of each clock, i.e. $(v + \delta)(x) = v(x) + \delta$. $\mathcal{C}(\mathcal{X})$ denotes the set of *clock constraints* over $\mathcal{X}$. These are conjunctions of atomic constraints of the form $x \bowtie c$ or $x - y \bowtie c$ where $x, y \in \mathcal{X}$, $c \in \mathbb{Z}$ and $\bowtie \in \{<, >, \leq, \geq\}$. A clock valuation v satisfies a clock constraint g if the value of each clock under v satisfies each atomic constraint comprising g, if so we write $v \vDash g$. A clock constraint is said to be *diagonal free* if it is the conjunction of atomic constraints that only place upper and lower bounds on individual clocks, and does not directly restrict the difference between any pair of clocks.

▶ **Definition 1** (Timed Automaton)**.** A timed automaton $\mathcal{A}$ is defined to be a tuple $(Q, q_0, \mathcal{X}, \Sigma, E, I)$, where

- Q is a finite set of discrete locations.
- $q_0 \in Q$ is the starting location.
- $\mathcal{X}$ is a set of clocks.
- Σ is a finite alphabet of actions.
- $E \subset Q \times \Sigma \times \mathcal{C}(\mathcal{X}) \times 2^{\mathcal{X}} \times Q$ is a relation defining the jumps between locations. $e = (q, \alpha, G, X, q')$ represents a jump from the location q to q' by taking the action α. G is the guard on e: a diagonal free clock constraint that must be satisfied by the current clock valuation in order for the jump to be taken. X is the subset of clocks to be reset to 0 when the jump is taken.
- $I : Q \to \mathcal{C}(\mathcal{X})$ assigns to each location an invariant condition that must be satisfied at all times while in that location. Like the guards on the jumps, each location invariant must be diagonal free.

A *state* of a timed automaton $\mathcal{A}$ is a pair (q, v) with $q \in Q$ and v a valuation over $\mathcal{X}$. The execution of $\mathcal{A}$ gives a transition system over the infinite space of possible states with two types of transitions:

- A *delay transition* from a state (q, v) has the form $(q, v) \xrightarrow{\delta} (q, v + \delta)$. This transition represents the automaton idling in location q for a duration of δ, provided the state continues to satisfy the invariant condition of the location, that is $v + t$ satisfies $I(q)$ for all $0 \leq t \leq \delta$.
- A *discrete transition* with respect to a jump $e = (q, \alpha, G, X, q') \in E$ from a state (q, v) has the form $(q, v) \xrightarrow{e} (q', v')$, where $v' = v[X := 0]$, v satisfies G and v' satisfies $I(q')$. This represents the automaton taking action α in q resulting in the jump to q' provided the guard on e is satisfied by v and the invariant at q' is satisfied by v'.

A finite *run* of $\mathcal{A}$ is a finite sequence $\rho = (q_0, \mathbf{0}) \xrightarrow{\delta_1}\xrightarrow{e_1} s_1 \xrightarrow{\delta_2}\xrightarrow{e_2} \ldots \xrightarrow{\delta_n}\xrightarrow{e_n} s_n$ where each $s_{i-1} \xrightarrow{\delta_i}\xrightarrow{e_i} s_i$ is a delay-transition (possibly of 0 duration) followed by a discrete-transition. This definition can be extended to infinite runs of $\mathcal{A}$ by extending ρ to be an infinite sequence.

2.1.1 Double-Priced Timed Automata and Optimal Infinite Scheduling

The model of timed automata has been extended in various ways to model different aspects of real time systems. A common extension is to associate a price with each location and jump; a cost is accrued for each unit of time spent in a location, as well as for each jump taken [3]. The accumulated cost of runs of the automaton can then be analysed, for example the minimum cost reachability of certain states [2]. In 2008, Bouyer, Brinksma and Larsen proposed the model of *double-priced timed automata*: a timed automaton extended with two price functions: costs and rewards [4]. This model was designed for the purpose of analysing the infinite behaviours of systems via their long term cost to reward ratio. The minimisation of this ratio for long term behaviours is called the *optimal infinite scheduling problem*. The solutions to this problem are of particular interest when considering systems that need to perform a series of operations indefinitely, such as industrial processes where the operator would want the long term rewards to exceed the costs. We now formally define a double-priced timed automaton:

▶ **Definition 2** (Double-Priced Timed Automaton)**.** A double-priced timed automaton (DPTA) $\mathcal{A}$ is a tuple $(Q, q_0, \mathcal{X}, \Sigma, E, I, \mathtt{Cost}, \mathtt{Reward})$ where

- $(Q, q_0, \mathcal{X}, \Sigma, E, I)$ defines a timed automaton as defined in definition 1.
- $\mathtt{Cost} : (\mathtt{Q} \cup \mathtt{E}) \to \mathbb{N}$ is a function assigning to each jump a cost and to each location a cost rate.
- $\mathtt{Reward} : (\mathtt{Q} \cup \mathtt{E}) \to \mathbb{N}$ is a function assigning to each jump a reward and to each location a reward rate.

The transition system of a DPTA is identical to that of a timed automaton. However each transition now has an associated cost and reward. A delay transition $(q, v) \xrightarrow{\delta} (q, v + \delta)$ has cost $\mathtt{Cost}(q) \cdot \delta$ and reward $\mathtt{Reward}(q) \cdot \delta$, that is each time unit spent in location q contributes an amount equal to its cost or reward rate. A discrete transition $(q, v) \xrightarrow{e} (q', v')$ simply has the cost and reward $\mathtt{Cost}(e)$ and $\mathtt{Reward}(e)$ respectively. Given a finite run of a DPTA $\rho = (q_0, \mathbf{0}) \xrightarrow{\delta_1}\xrightarrow{e_1} (q_1, v_1) \xrightarrow{\delta_2}\xrightarrow{e_2} \ldots \xrightarrow{\delta_n}\xrightarrow{e_n} (q_n, v_n)$ we define the cost of the run as the sum of the costs of each transition: $\mathtt{Cost}(\rho) = \sum_{i=1}^{n}(Cost(e_i) + Cost(q_{i-1} \cdot \delta_i)$. The reward of a finite a run is defined similarly. The ratio of a run is simply the cost divided by the reward: $\mathtt{Ratio}(\gamma) = \mathtt{Cost}(\gamma)/\mathtt{Reward}(\gamma)$. For infinite runs, we consider the limit of the ratio as the run progresses. For an infinite run $\rho = (q_0, \mathbf{0}) \xrightarrow{\delta_1} \xrightarrow{e_1} (q_1, v_1) \xrightarrow{\delta_2} \xrightarrow{e_2} \ldots$, let ρ_n be the prefix of ρ consisting of the first n transitions. The ratio of the run is defined as $\texttt{Ratio}(\rho) := \lim_{n\to\infty} \texttt{Ratio}(\rho_n)$. To ensure that the ratios of all infinite runs of the DPTA exist, we require that the automaton be *strongly reward diverging*: for every infinite run ρ, we have $\lim_{n\to\infty} \texttt{Reward}(\rho_n) = \infty$, i.e an infinite number of actions leads to an infinite reward. This effectively means that there are no cycles with zero reward.

We now define the optimal infinite scheduling problem for DPTA. Given a strongly reward diverging DPTA $\mathcal{A} = (Q, q_0, \mathcal{X}, \Sigma, E, I, \texttt{Cost}, \texttt{Reward})$, let Γ be the set of all infinite runs of $\mathcal{A}$. The optimal infinite scheduling problem is to determine the lower bound on achievable cost to reward ratios of infinite runs of the automaton: $\texttt{Ratio}^* = \inf\{\texttt{Ratio}(\rho) \mid \rho \in \Gamma\}$. Note that there may not be a run of $\mathcal{A}$ with this ratio, but instead a family of runs such that their ratio becomes arbitrarily close to $\texttt{Ratio}^*$. Bouyer, Brinksma and Larsen have shown that the optimal infinite scheduling problem is computable and is PSPACE-complete [4].

While the region abstraction defined by Alur and Dill preserves reachability properties [1], it does not preserve the ratios of infinite runs as the duration of timed transitions is abstracted away. In their proof of the computability of the optimal infinite scheduling problem, Bouyer, Brinksma and Larsen introduce an extension of the region abstraction that does preserve this property called the *corner point abstraction* [4]. Let $\mathcal{A} = (Q, q_0, \mathcal{X}, \Sigma, E, I, \texttt{Cost}, \texttt{Reward})$ be a DPTA with $|\mathcal{X}| = k$. We also require that the reachable clock-space of $\mathcal{A}$ is bounded, that is there exists a constant M such that for all reachable clock-valuations v, $v(x) \leq M$ for all clocks x in $\mathcal{X}$. A *corner point* is an element $\mathbf{a} = (a_1, \ldots, a_k)$ of $\mathbb{N}^k$, which can be thought of as a clock valuation. If (q, R) is a region of $\mathcal{A}$, a corner point $\mathbf{a}$ is associated with (q, R) if $\mathbf{a}$ lies in the closure of R. Interpreting R as a polyhedron in the clock space, then the corner points associated with R are the vertices of the closure of that polyhedron. The corner point abstraction of $\mathcal{A}$ is a double-weighted graph with vertices of the form $(q, R, \mathbf{a})$ where (q, R) is a region and $\mathbf{a}$ is a corner point associated with (q, R). The edges of the graph represent either discrete or delay transitions of the automaton. The edges have two weights - a cost and a reward corresponding to the cost and reward of the transition in the original automaton. For each jump $e = (q_{\texttt{src}}, \alpha, G, X, q_{\texttt{dest}})$ of $\mathcal{A}$, for every vertex $(q, R, \mathbf{a})$ such that $q = q_{\texttt{src}}$ and R satisfies the guard G, then there is an edge from this vertex to the unique vertex $(q', R', \mathbf{a}')$ such that $q' = q_{\texttt{dest}}$, $R' = R[X := 0]$ and $\mathbf{a}' = \mathbf{a}[X := 0]$. This edge has weights $\texttt{Cost}(e)$ and $\texttt{Reward}(e)$. There are two types of edges corresponding to delay transitions in the corner point abstraction:

- Given two vertices $(q, R, \mathbf{a})$ and $(q, R, \mathbf{a}+1)$ (i.e $\mathbf{a}+1$ is an immediate time successor of $\mathbf{a}$), there is an edge between them with weights $\texttt{Cost}(q)$ and $\texttt{Reward}(q)$ as this corresponds to a delay of 1 time unit in location q.
- Given two vertices $(q, R, \mathbf{a})$ and $(q, R', \mathbf{a})$ with the region R' being a time successor of R, there is an edge between the vertices with cost and reward equal to 0, as this is a delay of zero duration.

A path through the corner point abstraction corresponds to a run of the automaton, with corresponding accumulated cost and reward. However, the only delay transitions represented in the corner-point abstraction are those of duration $z + \epsilon$ where z is a non-negative integer and ϵ is some real number with $|\epsilon| << 1$. Bouyer, Brinksma and Larsen [4] prove that these transitions are sufficient, and that the corner point abstraction is sound and complete with respect to the optimal infinite scheduling problem for double priced timed automata, i.e. that the optimal ratio of infinite paths in the corner point abstraction is equal to $\texttt{Ratio}^*$ in the corresponding DPTA. The optimal infinite scheduling problem can therefore be reduced to finding the minimum cost to reward ratio cycle in the corner point abstraction. If the corner point abstraction has V vertices and E edges, then the best known algorithm for this problem - Burn's Algorithm has time complexity $O(V^2E)$ [7]. As the number of vertices in the corner point abstraction is proportional to the number of regions of the automaton, which has an upper bound of $|Q| \cdot |\mathcal{X}|! \cdot 2^{|\mathcal{X}|} \cdot \prod_{x\in\mathcal{X}}(2c_x + 2)$ as seen above, applying this method does not give a feasible algorithm for most automata.

A possible approach to effectively solving the optimal infinite scheduling problem is to use a zone-based abstraction that preserves the optimal ratio of runs, rather than a region-based one. David et al. have constructed such a method specifically for the case of timed automata with only one clock [8]. Their method is based on *strong time abstracting bisimulations* (STABs)- a zone based method of partitioning the state space of an automaton into zones that have equivalent behaviour [16]. In the one clock case zones are simply intervals of non-negative real numbers. David et al. [8] define an abstraction based on the end points of these intervals and prove that it preserves the optimal infinite ratio. The procedure of constructing this abstraction and finding the optimal ratio has complexity $O(|Q|\cdot(|Q|+|E|)^6)$, which is much more feasible than using the corner point abstraction. However, most real time systems require the use of several clocks to effectively model their behaviour. This work has not currently been extended to the case with more than one clock, and so there is a lack of effective algorithms for solving the optimal infinite scheduling problem for timed automata with two or more clocks. Additionally, in the worst case the size of a STAB of a timed automaton is as large as that of the region abstraction, so even given a STAB based method, the computation of the optimal infinite ratio will still be infeasible.

2.2 Population Based Metaheuristics

For many optimisation problems, especially those of high complexity, an exact solution is not required. Instead an approximate solution is often sufficient. In this case, we can make use of heuristic algorithms to rapidly explore the state-space of the problem to find a solution that is good enough, but not provably optimal. As demonstrated in the previous section, the complexity of the optimal infinite scheduling problem for double-priced timed automata is typically too high to admit an exhaustive search of the state-space, and so the use of heuristics is a suitable method for compromising precision for computational efficiency in this domain.

2.2.1 Ant Colony Optimisation

Ant colony optimisation is a population based metaheuristic first developed by Marco Dorigo in 1996 as a method of determining approximate solutions to the travelling salesman problem [9]. The method is inspired by the way ants utilize pheromone in order to communicate the best path between their nest and a food source. Ant colony optimisation is typically used for combinatorial optimisation problems consisting of a finite set of discrete decisions, such as finding traversals of graphs, but it has been extended to problems in continuous domains [12]. While these problems are typically NP, Ant Colony optimisation has been applied to some PSPACE problems such as the Canadian Traveller Problem [13].

Ant colony optimisation operates by simulating a population of agents representing artificial ants. To illustrate we will suppose we have a combinatorial optimisation problem given as finding a traversal of a graph $G = \{V, E\}$ with V being the set of vertices and E the set of edges. We have some objective function over the traversals of the graph that we are trying to optimize. As each possible solution is a traversal, the discrete decisions are the choice of which edge to take at each point. The artificial ants communicate and make

Algorithm 1: High Level Ant Colony Optimisation.

```
repeat
    place ants;
    repeat
        foreach ant do
            if path not complete then
                follow state transition rule to choose next edge;
                take edge and apply local pheromone update;
            end
        end
    until all paths complete;
    apply global pheromone update;
until terminated;
return best path;
```

decisions based on a *pheromone function*: $\texttt{Pheromone} : \texttt{E} \rightarrow \mathbb{R}_{\geq 0}$, a function assigning to each edge a non-negative real number which represents the concentration of pheromone on that edge. A high pheromone concentration means that traversals including that edge evaluated well under the objective function. At a decision point a traversal is more likely to take an edge with a higher concentration of pheromone.

The overall algorithm operates over a series of iterations. Within each iteration each ant in the population builds a solution by first being placed at a starting vertex, and then taking edges one by one according to a *state transition rule*. After taking an edge, the ant updates that edge's pheromone concentration according to a *local pheromone updating rule*. The ant continues to take edges until a complete solution is formed. Once all ants have complete solutions, the pheromone along all edges is updated. This is called the *global pheromone update*. After a number of iterations the process completes, and the overall best found traversal is returned as the solution to the optimisation problem.

State Transition Rule. The state transition rule is how ants determine what edge to add to their solution. If an ant is at a vertex v, with outgoing edges $\texttt{Out(v)}$, the simplest used state transition rule is to choose a particular edge with probability corresponding to its relative pheromone concentration. For an edge e in $\texttt{Out}(v)$ the probability of taking this edge, $P(e)$ is given by:

$$P(e) = \frac{\texttt{Pheromone}(e)}{\sum\limits_{e' \in \texttt{Out}(v)} \texttt{Pheromone}(e')}$$

This means that an ant is more likely to take an edge known to give results, but the probabilistic factor means that alternate solutions will still be explored. It is also common to apply a heuristic factor, i.e. an approximation of that edge's quality, to the probability of taking an edge. The effect of the pheromone relative to the effect of the heuristic can then be adjusted using real-valued weighting parameters [9].

Pheromone Updates. In combination with the state transition rule, pheromone updates act as a mechanism for encouraging ants in future iterations to explore in the neighbourhood of previously found good solutions. The global pheromone update occurs after each ant has built its solution. First, some amount of the pheromone along all edges *evaporates*, that is it is reduced by a linear factor $\lambda \in [0,1]$ called the *evaporation rate*. For each edge e in E this is characterized by the update:

$$\texttt{Pheromone}(e) \leftarrow (1-\lambda) \cdot \texttt{Pheromone}(e)$$

This is a mechanism in place to make ants weight more recent information more heavily than older information. Then, each ant applies to each of the edges involved in its solution an amount of pheromone proportional to that solution's quality under the objective function. If f is the objective function, and P_k is the solution found by the kth ant, the update for an edge e is:

$$\texttt{Pheromone}(e) \leftarrow \texttt{Pheromone}(e) + \sum_{k \in Ants} \begin{cases} f(P_k) & e \in P_k \\ 0 & e \notin P_k \end{cases}$$

In his revised version of the ant colony optimisation [10], Dorigo found that using an *elitist* global update gave better performance in general. This is a modification where only the best n ants of an iteration are allowed to place pheromone along their path. In the same revision, Dorigo introduced the local pheromone update, where upon adding an edge to its partial solution an ant will slightly reduce the taken edge's pheromone concentration. This discourages ants within the same iteration from taking the same set of edges, ensuring that a diverse set of solutions is tested in each iteration. Later work by Stüzle and Hoos found that performance is also improved if pheromone concentration is capped between a minimum and maximum level [15].

3 Approximate Optimisation

In the following section we detail the application of population based metaheuristic methods to the optimal infinite scheduling problem for a double priced time automata $\mathcal{A} = (Q, q_{start}, \mathcal{X}, \Sigma, E, I, \texttt{Cost}, \texttt{Reward})$. There are several assumptions we will make regarding $\mathcal{A}$:

1. $\mathcal{A}$ is strongly reward diverging - This is to ensure that every infinite run of $\mathcal{A}$ has a defined cost to reward ratio.
2. The invariants I of $\mathcal{A}$ in each location bound the values of every clock. This ensures that the reachable clock space is bounded and hence the corner point abstraction [4] can be used.
3. $\mathcal{A}$ is *deadlock free*. That is for all reachable states $s = (q, v)$ of $\mathcal{A}$ there exists some positive delay δ such that the state $(q, v + \delta)$ satisfies the location invariant at q and the guard condition of at least one jump from q. Intuitively this means that at all states, by letting time pass the automaton is guaranteed to be able to take a discrete transition.

When considering the corner point abstraction, conditions 2 and 3 together ensure that starting from a particular vertex of the corner point an infinite execution will eventually cycle and repeat a corner point without reaching any 'dead ends'. This fact is useful for generating cyclic executions of the automaton on the fly, without the need for backtracking.

Our approach will consist of applying metaheuristic techniques in order to find cycles in the corner point abstraction that have a ratio that is as small as possible. The goal is to explore the state-space as efficiently as possible to give a good solution, although the solutions that are found will not be provably optimal.

In order to generate these cycles, we use a simple representation for the vertices of corner point abstraction, which we will refer to as a *CP-state*. For an automaton with k clocks, a CP-state $\underline{s} = (q, \mathbf{z}, \mathbf{d})$ is a location q, alongside a pair of integer-valued vectors $\mathbf{z}$ and $\mathbf{d}$ with $\mathbf{z} = (z_1, \dots, z_k)$ and $\mathbf{d} = (d_1, \dots, d_k)$. The ith component of each vector is related to the ith clock of the automaton. The vector $\mathbf{z}$ gives the corner-point with which the CP-state is associated, while $\mathbf{d}$ defines the relative ordering of the fractional components of each clock. If ϵ is a very small positive real number, the CP-state $(\mathbf{z}, \mathbf{d})$ corresponds to the vertex of the corner point abstraction associated with the corner point $\mathbf{a} = \mathbf{z}$, and with the region to which the point $\mathbf{z} + \epsilon \cdot \mathbf{d}$ belongs.

3.1 Ant Colony Optimisation

As the corner point abstraction is a graph, we could theoretically apply ant colony optimisation directly in order to find a good cycle. However for this approach, as part of the pheromone function representation, we would need to store a value for each individual edge in the corner point abstraction. The upper bound on the number of vertices is of the order $|Q| \cdot |\mathcal{X}|! \cdot 2^{|\mathcal{X}|} \cdot \prod_{x \in \mathcal{X}} (2c_x + 2)$ [1], with the number of edges being even larger, so memory-wise this is not feasible. Instead we will approach the problem as a multi-step problem with two components. The first component is the problem of determining what sequence of discrete transitions of the automaton will compose the cycle. The second component is the problem of determining what delays to take between the discrete transitions.

3.1.1 Simple Ant Colony Optimisation with Random Delays

The simplest approach under this interpretation is to have the pheromone function assign values to the jumps of the automaton. Given a DPTA $\mathcal{A} = (Q, q_{start}, \mathcal{X}, \Sigma, E, I, \texttt{Cost}, \texttt{Reward})$, we have a pheromone function $\texttt{pheromone} : E \to \mathbb{R}_{\geq 0}$. Initially, before performing any exploration we set $\texttt{pheromone}(e) = 1$ for all e in E. In each iteration of the algorithm, each ant will be placed at a random CP-state of $\mathcal{A}$. The ant will then build a solution by sequentially taking a delay transition (randomly) and then a discrete transition (based on pheromone), until it reaches a CP-state it has already visited. By our assumptions listed at the start of the chapter, this is guaranteed to occur and as such we will have a cycle from the repeated CP-state to itself. As we are only interested in the cycle itself, the path leading up to the cycle can be discarded.

To choose a delay from a particular CP-state, the ant considers the set of delays that lead to a subsequent CP-state where at least one discrete-transition is enabled, as the next step requires choosing a discrete transition from that state. Due to the assumption of the automaton being deadlock-free this set of delays is guaranteed to be non-empty. A delay from this set is chosen with uniform probability.

For this algorithm the state-transition rule will determine which jump of the automaton an ant will take from a given CP-state $\underline{s}$. The probability of taking a jump e depends on both the pheromone concentration as well as the cost/reward ratio contributed by e, with a smaller ratio leading to a higher probability. Let $\texttt{Out}(\underline{s})$ be the set of jumps that can be taken from $\underline{s}$, the probability of taking a jump e in $\texttt{Out}(\underline{s})$, $P(e)$ is given by:

$$P(e) = \frac{\texttt{pheromone}(e)^{\alpha} \cdot (\frac{\texttt{Reward}(e)}{\texttt{Cost}(e)})^{\beta}}{\sum\limits_{e' \in \texttt{Out}(\underline{s})} \texttt{pheromone}(e')^{\alpha} \cdot (\frac{\texttt{Reward}(e')}{\texttt{Cost}(e')})^{\beta}}$$

α and β are real valued parameters that allow for weighting the effect of the pheromone and the jump's ratio when determining the probability.

As per the standard implementation of ant colony optimisation, we have local and global pheromone updates. As the local update, when an ant takes a jump e, the pheromone concentration of e is scaled towards the initial value 1 by a linear *decay factor* μ:

$$\texttt{Pheromone}(e) \leftarrow (1-\mu)\cdot\texttt{Pheromone}(e) + \mu\cdot 1$$

The global update first has the pheromone along all edges reduced according to the evaporation factor λ:

$$\texttt{Pheromone}(e) \leftarrow (1-\lambda)\cdot\texttt{Pheromone}(e)$$

Then we perform an elitist update based on the paths found. Let ρ_i be the ith best cycle found in this iteration, and n be the elitism factor, i.e. the number of top ants we are considering. We update by:

$$\texttt{Pheromone}(e) \leftarrow \texttt{Pheromone}(e) + \sum_{i=1}^{n} \begin{cases} (\texttt{Ratio}(\rho_i))^{-1} & e \in \rho_i \\ 0 & e \notin \rho_i \end{cases}$$

where $(\texttt{Ratio}(\rho))^{-1}$ is the reciprocal of the ratio of ρ, so a lower cost to reward yields a higher pheromone concentration. We also cap all pheromone values to be below a bound $\texttt{Pheromone}_{\texttt{max}}$. If we ever evaluate a reward/cost fraction where the cost is 0, we interpret this value as $\texttt{Pheromone}_{\texttt{max}}$ to ensure all values are well defined.

Using these steps as part of a standard ant colony optimisation algorithm we get a process that attempts to optimise the discrete transitions of runs in our DPTA. However, our pheromone function stores no information about the delay transitions being used, so these cannot be effectively optimised by this method. The next section introduces an approach to optimise both the discrete and delay transitions.

3.1.2 Hybrid Ant Colony Optimsation

Research has shown that with a modification to pheromone function representation, ant colony optimisation can be applied to optimisation in continuous domains [12]. With further modification we can use this approach to solve hybrid discrete-optimisation problems. The optimal infinite scheduling problem can be constructed in this way. Choosing the jumps that form the cycle is the discrete problem, and choosing the delays is the continuous problem. This section will extend the previous method to utilize this hybrid approach. We are still considering paths within the corner point abstraction, but instead of choosing a delay and then a jump we choose a *joint action* consisting of both these parts: (δ, e) where δ is the positive delay, and e is the jump.

When considering what joint action to take, while the delay itself determines the cost and reward contributed by the current location, the state that we delay to and take the jump from has more of an impact over the automaton's potential future behaviour. Therefore, our new pheromone function should act as an estimator of the utility of taking a jump from a particular state. As a jump can only be taken from its source location, we only need to consider the valuation of the state and not its location. So for a k clock automaton, we have $\texttt{Pheromone} : \mathbb{R}^k_{\geq 0} \times E \to \mathbb{R}_{\geq 0}$. A challenge here is the effective representation of the pheromone function, as the domain is partially continuous. We would also like the placement pheromone to affect not only the exact state the jump was taken from, but also

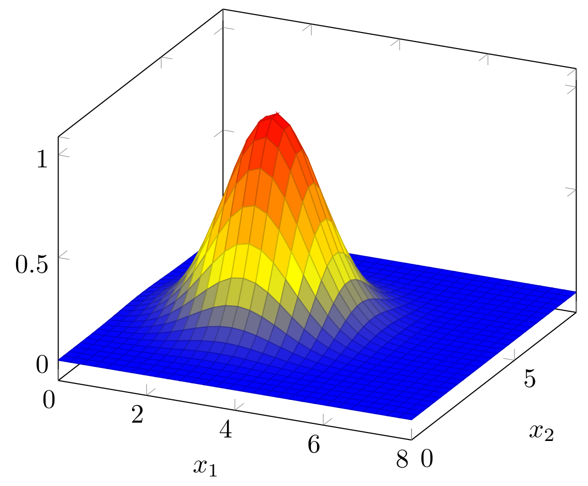

Figure 1 2D Gaussian kernel $G_{\mathbf{v}}(\mathbf{x})$ with $\mathbf{v} = (3, 4)$ and $\sigma = 1$.

the neighbourhood around that state so as to inform later ants that similar states to this might also be good candidates. As explored by Socha and Dorigo [12], a solution is to have the pheromone function be a sum of k-dimensional Gaussian kernels. Given a center-point $\mathbf{v}$ the Gaussian kernel $G_{\mathbf{v}} : \mathbb{R}^k \to \mathbb{R}$ is of the form:

$$G_{\mathbf{v}}(\mathbf{x}) = \exp\left(-\frac{|\mathbf{v} - \mathbf{x}|^2}{2\sigma^2}\right)$$

where $|\mathbf{v} - \mathbf{x}|$ is the Euclidean distance between $\mathbf{v}$ and $\mathbf{x}$ and σ is a constant defining the width of the curve. As seen in Figure 1, the function gives a maximum of 1 at $\mathbf{x} = \mathbf{v}$ and approaches 0 as $\mathbf{x}$ moves further away.

We can then define our pheromone function as a linear combination of these curves centered at the points pheromone has been applied. For a particular jump e, let $\mathbf{v}_1, \ldots, \mathbf{v}_m$ be the points at which pheromone has been applied, and $\tau_1, \ldots, \tau_m$ be the corresponding concentration that was applied at each point. Each value τ_i is the maximum height of the corresponding curve, We also add the constant function 1 so that the initial pheromone concentration at each point is 1 like in the previous method. Our pheromone function is then given by:

$$\texttt{Pheromone}(\mathbf{x}, e) = 1 + \tau_1 \cdot G_{\mathbf{v}_1}(\mathbf{x}) + \ldots + \tau_m \cdot G_{\mathbf{v}_m}(\mathbf{x})$$

Figure 2 shows an example of a pheromone function at a particular jump in a 1 clock automaton, where varying concentrations of pheromone have been placed at $x = 2$, $x = 6$ and $x = 12$.

This method is easily represented by a list of centre points and concentrations for each jump of the automaton. Evaluation of the function simply requires taking the value 1 and adding the corresponding Gaussian functions at the particular point, restricting the value if it exceeds our $\texttt{Pheromone}_{max}$ value. As the computation time for this is linear in the number of curves present, we restrict the number of curves allowed to a number n, keeping only the most recent n curves for each jump.

The rest of the algorithm operates mostly the same as the previous with a few modifications. As our pheromone function has changed, the state-transition rule is slightly modified. At a given CP-state $\underline{s}$ the ant considers the set of all legal joint actions $\texttt{Actions}(\underline{s})$ which is

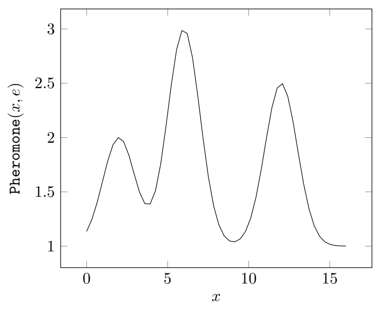

Figure 2 Example 1D pheromone function for a particular jump.

the set of actions (δ, e) such that $\underline{s} + \delta$ satisfies both the current location invariant and the guard of e. The probability of taking an action (δ, e) depends on the pheromone value at the state the discrete transition is taken from: $\texttt{Pheromone}(\underline{s} + \delta, e)$, and the reward and cost that this action contributes. We define the reward-cost ratio of an action (δ, e) from a CP-state $\underline{s} = (q, \mathbf{z}, \mathbf{d})$ as:

$$\texttt{Ratio}(\delta, e, \underline{s})^{-1} := \frac{\texttt{Reward}(q) \cdot \delta + \texttt{Reward(e)}}{\texttt{Cost}(q) \cdot \delta + \texttt{Cost(e)}}$$

The state -transition rule then gives the probability of taking an action (δ, e) in $\texttt{Actions}(\underline{s})$:

$$P(\delta, e) = \frac{\texttt{Pheromone}(\underline{s} + \delta, e)^{\alpha} \cdot \texttt{Ratio}(\delta, e, \underline{s})^{-\beta}}{\sum\limits_{(\delta', e') \in \texttt{Actions}(\underline{s})} \texttt{Pheromone}(\underline{s} + \delta', e')^{\alpha} \cdot \texttt{Ratio}(\delta', e', \underline{s})^{-\beta}}$$

The parameters α and β are weightings of each component as before. Local and global pheromone updates act as before, with the change that whenever pheromone is scaled, instead the height parameter of each component Gaussian curve is scaled, and whenever a value is added, a Gaussian curve with that height is added instead. To keep the number of Gaussian curves low, we set a small real valued constant $\texttt{height}_{min}$. Whenever the scaling of a Gaussian function results in its height being less than $\texttt{height}_{min}$, the curve is removed from that pheromone function.

The remaining parts of the algorithm: iteration, elitist updating and returning of the best solution are identical to that of the simple ant colony optimisation method. The modified pheromone function and state transition rule gives a method for approximating the optimal infinite ratio of the automaton, that takes into account more information than the previous method, but has more significant time and memory requirements due to the pheromone function representation.

3.2 Cycle Optimisation

In the previous methods the delays for a given cycle of the automaton were determined approximately. In this section we introduce a method that, given a set of jumps forming a cycle in the automaton, will exactly determine the delays that give the best possible cost to reward ratio for that cycle. With this method, the optimal infinite scheduling problem only requires the trial of different combinations of discrete transitions.

Let $\mathcal{A} = (Q, q_{start}, \mathcal{X}, \Sigma, E, I, \texttt{Cost}, \texttt{Reward})$ be a double-priced timed automaton, and let $(e_i)_{i=1}^n = e_1, e_2, \ldots, e_n$ be a sequence of jumps such that $e_i = (q_{i-1}, \alpha_i, G_i, X_i, q_i) \in E$ with $q_0 = q_n$, that is the edges of $(e_i)_{i=1}^n$ form a cycle in $\mathcal{A}$. Given this sequence we wish to find a cyclic execution of $\mathcal{A}$, $\rho = (q_0, v_{0,0}) \xrightarrow{\delta_0} (q_0, v_{0,1}) \xrightarrow{e_1} (q_1, v_{1,0}) \xrightarrow{\delta_1} \dots \xrightarrow{e_{n-1}} (q_{n-1}, v_{n-1,0}) \xrightarrow{\delta_n-1} (q_{n-1}, v_{n-1,1}) \xrightarrow{e_n} (q_0, v_{0,0})$ with $\texttt{Ratio}(\rho)$ being minimal. As the jumps involved are fixed, this problem consists of finding optimal delays δ_i such that ρ is a valid execution. The valuations $v_{i,j}$ can then be derived from the delays. Note that unless all delays are 0, then for a solution to exist each clock must be reset by at least one jump, as such we will only consider sequences in which all clocks are reset at least once. To begin with, we will also assume that all invariant and guard conditions are closed, that is they only make use of the weak inequalities $\geq$ and $\leq$.

To solve this problem we will first derive a set of constraints that will ensure that ρ is a valid execution. We then redefine the objective function and constraints with respect to a new set of variables such that the constraints define a bounded polyhedron and the objective function is a quotient of affine functions. This will give a linear-fractional program that we can transform into a linear program to then solve. The constraints, objective function and solution are described in the appendix (Appendix A). This method gives a computable function $\texttt{CycleOpt} : \{e_n\} \to \mathbb{R}$ that maps to each sequence of jumps in the automaton forming a cycle, a real number that is the minimum achievable cost to reward ratio for that particular cycle. If a sequence $e_1, \dots, e_n$ is infeasible, that is no valid execution of the sequence of jumps exists, we set $\texttt{CycleOpt}(e_1, \dots, e_m) = \infty$.

The cycle optimisation can be extended to handle strict inequalities in guards, as described in the appendix (Appendix A.1).

4 Ant Colony Optimisation with Cycle Optimisation

The cycle optimisation algorithm can be applied in conjunction with ant colony optimisation to the optimal infinite scheduling problem. The ants are responsible for testing cycles consisting of different jumps of the automaton, while the cycle optimisation algorithm returns the optimal delays as well as the best achievable cost reward ratio for a cycle composed of those jumps.

The implementation of this algorithm operates largely the same as the simple ant colony optimisation algorithm discussed earlier. Ants traverse the CP-states of the automaton until they reach a cycle, informed by the pheromone function over the set of jumps. The difference occurs around the time of the global pheromone update. In the orginal algorithm each of the best k ants would place pheromone corresponding to the cycle it built. Instead, for each of these ants, the sequence of discrete-transition used in its solution $e_1, \dots, e_n$ is passed into the cycle optimisation method. $\texttt{CycleOpt}(e_1, \dots, e_n)$ is calculated, with the result being used to update the concentration of pheromone along each jump in the sequence. This reduces the drawback of using random delays during exploration, while still having the benefits of the simple pheromone function this enables. During the exploration ants are essentially finding *feasible* sequences of jumps, while the $\texttt{CycleOpt}$ function finds the best ratio over a subset of these feasible cycles. This is beneficial as the $\texttt{CycleOpt}$ function, the most computationally expensive part of the iteration, only needs to be executed for a small number of the candidate solutions. A drawback that remains in this method is that the pheromone function only informs what set of jumps to use, while the optimal achievable ratio depends on the exact sequence of jumps. The ratio varies on what order the jumps appear in as well as on how many times a jump is taken.

Table 1 Rates of finding the overall minimal ratio on each of the 175 timed automata test cases.

Method	Rate
Ant Colony with CO	61.5%
Simple Ant Colony	44.6%
Random	43.5%
Hybrid Ant Colony	38.2%

Table 2 Normalized average ratios across all test cases. This is given by dividing the ratio obtained in each trial by the best ratio found for that automaton.

Method	Normalized Avg.
Ant Colony with CO	1.131
Random	1.161
Simple Ant Colony	1.202
Hybrid Ant Colony	1.297

5 Results

This section details the execution and results of several experiments designed to test the effectiveness of each of the above algorithms for solving the optimal infinite scheduling. Of interest are the ratios obtained, the time taken to do so, and the types of cycle structures each algorithm is best suited finding. Each algorithm was implemented in Java 8.

As a baseline for comparison with the other algorithms, a method that simply explores the CP-states randomly was implemented. This method generates some number of random cycles in the corner-point abstraction, with the best found cycle being returned as the solution. At each decision point, which delay or jump to take is determined using a uniform distribution. The parameter tuning is described in the appendix (Appendix B).

The primary experiment for evaluation consisted of running each algorithm on 175 randomly generated double priced timed automata. Each method was performed 10 times per automaton with the best found ratio, the average ratio and the average time taken to terminate being recorded. The test case generation (Appendix C) and optimal ratio calculation (Appendix D) are described in the appendix. Even on these small instances it was not feasible to run the deterministic algorithm [4] to confirm the optimal ratio.

5.1 Results and Discussion

For each automaton, the best ratio that either the optimal ratio search or any of the five methods returned was recorded and used as a point of comparison. How frequently each method returned the best ratio for a given automaton was recorded. This is shown in Table 1.

The algorithms making use of cycle optimisation performed the best by this measure, being the only methods to find the optimum in over 50% of the trials. Notably, the hybrid ant colony algorithm found the optimum less frequently than even the random method.

The normalized average ratio found by each method was also recorded. This value is given by taking the ratio found in a particular trial and dividing by the best ratio found across all trials for the same automaton. A normalized average of 1.0 would suggest that the method found the optimum in every trial, with higher values suggesting worst performance. The normalised ratio gives a measure of how close to the optimum the ratios returned by each method were. These results are shown in Table 2. Ant colony optimisation with cycle

Table 3 Rates of taking the optimal loop.

Method	Rate
Ant Colony with CO	52.4%
Simple Ant Colony	50.4%
Hybrid Ant Colony	15.4%
Random	10.4%

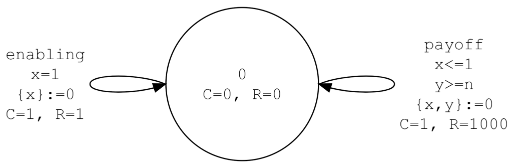

Figure 3 An example loop automaton. The optimal behaviour is given by taking **`enabling`** $n - 1$ times and then taking **`payoff`**.

optimisation gave the best performance by this measure. Interestingly the random method had the second best performance. However, this performance comes at a price, taking on average 9 seconds per trial, compared with 1 second per trial for the simple ant colony methods. The hybrid and random average 1.5 seconds per trial.

Based on the results of this experiment, ant colony optimisation with cycle optimisation appear to be the best method for finding a profitable cyclic behaviour of a timed automaton. Although this method has a greater time requirement than the others, the improvement to the behaviours found is likely to make this time increase worthwhile. Overall the hybrid ant colony optimisation algorithm was the worst at optimising the behaviour of the automaton, performing even worse than the baseline random method. It is possible that the pheromone representation more frequently led the algorithm astray rather than allowing it to improve upon its solutions in each iteration.

6 Additional Experiments

In addition to the primary experiment, the methods were tested on automata with a particular structure designed to stress the algorithms and find potential weaknesses.

6.1 Jump Repetition

This experiment tested how frequently each method found the optimal cycle when that cycle consisted of taking a particular jump n times followed by a high reward jump. Each test case had 2 clocks and a single location with 0 cost and reward. The jumps consisted of an enabling jump which allows the difference between the two clock values to increase by at most 1 each time it is taken, a payoff jump which can only be taken when the difference between the clocks is at least n and a set of k “noise” jumps. All jumps have cost 1, with the enabling jump having reward 1, the payoff jump having reward 1000 and the noise jumps having rewards in the range 1 to 10. The payoff jumps reward dominates that of all others and hence the optimal ratio is obtained by the cycle where the enabling jump is taken n times and then the payoff jump is taken. Figure 3 shows the structure of the automata.

There were 50 test cases with n ranging from 1 to 10 and k ranging from 1 to 5. In each test case, each method was run 10 times. The percentage of these runs that found the

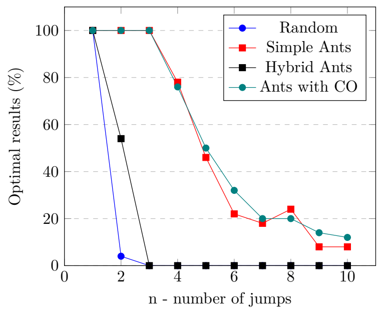

Figure 4 Rate of taking the optimal loop, with respect to number of jumps.

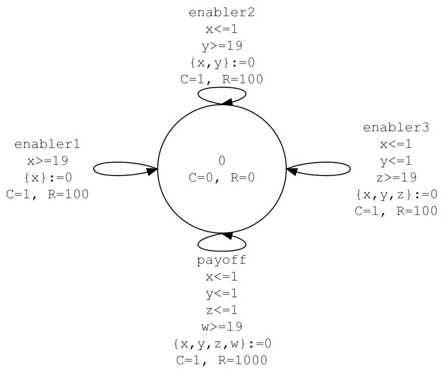

Figure 5 An example sequence automaton of length 4. The optimal behaviour is given by `enabler1` $\to$ `enabler2` $\to$ `enabler3` $\to$ `payoff`.

optimal ratio were recorded. These results are shown in Table 3. Figure 4 shows how the number of times the repeated jump was required to be taken affects how frequently each method found the optimal behaviour.

The two best performing methods were the simple ant colony optimisation, and ant colony optimisation with cycle optimisation. Notably these two methods use a pheromone function associated only with individual jumps, so a high pheromone concentration on the enabling jumps would cause these algorithms to take it more frequently.

6.2 Sequence Length

This experiment consisted of test cases with one location where the optimal cycle consisted of taking a specific sequence of jumps of length n, again in the presence of k "noise" jumps. All jumps had cost 1. Noise jumps had rewards between 1 and 10, the first $n-1$ jumps in the sequence had reward 100 and the final jump has reward 1000. The test case has n clocks, with the ith jump resetting the first i clocks and requiring that the first $i-1$ clocks, but not the ith have been reset, making it so the final jump is only possible to take if the first $n-1$ jumps have been taken in the correct order with no noise jumps (which reset all clocks) in between. The final jump's high reward again dominates, making the set sequence the optimal cycle for the automaton. Figure 5 shows an example with an optimal sequence of length 4.

There were 50 test cases with n ranging from 1 to 10 and k ranging from 1 to 5. In each test case, each method was run 10 times. The percentage of these 10 that found the optimal

Table 4 Rates of taking the optimal sequence.

Method	Rate
Hybrid Ant Colony	100%
Ant Colony with CO	76.6%
Simple Ant Colony	73.6%
Random	27.2%

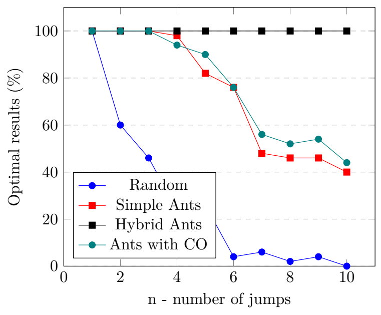

Figure 6 Rate of taking the optimal sequence, with respect to number of jumps.

ratio were recorded. These results are shown in Table 4. Figure 6 shows how the length of the optimal sequence affects how frequently each method found the optimal behaviour.

In this experiment the ant colony optimisation based methods greatly outperformed the other methods. Notably the hybrid ant colony optimisation algorithm found the optimal sequence in every single trial, despite performing very poorly in the other experiments. As the hybrid ant colony optimisation pheromone function varies over where in the clock space a jump is taken from. As what jumps can currently be taken depends on the resets of the jumps taken previously, the function is able to express some heuristic information about the impact of the order of the jumps. This result suggests that while in the general case the hybrid ant colony is less efficient at finding a good behaviour, this pheromone representation is well suited to the cases where a very specific sequence of actions is required.

7 Conclusion

This investigation shows that in many cases it is possible to achieve optimal or near optimal strategies for double priced timed automata using variations of ant colony optimisation, and linear programming for cycle optimisation. However, it also shows that it is possible to engineer test cases with specific solutions that are difficult for some heuristic search methods to find. Future work will work towards finding robust variations that are able to consistently produce near optimal strategies.

References

1 R Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

2 Gerd Behrmann, Ansgar Fehnker, Thomas Hune, Kim Guldstrand Larsen, Paul Pettersson, Judi Romijn, and Frits W. Vaandrager. Minimum-cost reachability for priced timed automata. In *Proceedings of the 4th International Workshop on Hybrid Systems: Computation and Control*, HSCC '01, pages 147–161, 2001.

3 Gerd Behrmann, Kim G. Larsen, and Jacob I. Rasmussen. Priced timed automata: Algorithms and applications. In *in International Symposium Formal Methods for Components and Objects (FMCO*, pages 162–182, 2005.

4 Patricia Bouyer, Ed Brinksma, and Kim G. Larsen. Optimal infinite scheduling for multi-priced timed automata. *Formal Methods in System Design*, 32(1):3–23, 2008.

5 A. Charnes and W. W. Cooper. Programming with linear fractional functionals. *Naval Research Logistics Quarterly*, 9(3-4):181–186, 1962. `doi:10.1002/nav.3800090303`.

6 George B Dantzig. Maximization of a linear function of variables subject to linear inequalities. *Activity analysi of production and allocation*, 1951.

7 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*, DAC '99, pages 37–42, 1999.

8 Alexandre David, Daniel Ejsing-Duun, Lisa Fontani, Kim G. Larsen, Vasile Popescu, and Jacob Haubach Smedegård. Optimal infinite runs in one-clock priced timed automata. Annual Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS), 2011.

9 M. Dorigo, V. Maniezzo, and A. Colorni. Ant system: Optimization by a colony of cooperating agents. *Trans. Sys. Man Cyber. Part B*, 26(1):29–41, 1996.

10 Marco Dorigo and Luca Maria Gambardella. Ant colony system: A cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, 1(1):53–66, 1997.

11 Peter Niebert, Stavros Tripakis, and Sergio Yovine. Minimum-time reachability for timed automata. In *IEEE Mediteranean Control Conference*, 2000.

12 Krzysztof Socha and Marco Dorigo. Ant colony optimization for continuous domains. *European Journal of Operational Research*, 185(3):1155–1173, 2008. `doi:10.1016/j.ejor.2006.06.046`.

13 P Soustek, R Matousek, J Dvorak, and J Bednar. Canadian traveller problem: A solution using antcolony optimization. In *Proceedings of 19th International Conference on Soft Computing – MENDEL 2013*, page 439–444, 2013.

14 Daniel A. Spielman and Shang-Hua Teng. Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time. *J. ACM*, 51(3):385–463, 2004. `doi:10.1145/990308.990310`.

15 Thomas Stützle and Holger H. Hoos. Max–min ant system. *Future Generation Computer Systems*, 16(8):889–914, 2000. `doi:10.1016/S0167-739X(00)00043-1`.

16 Stavros Tripakis and Sergio Yovine. Analysis of timed systems using time-abstracting bisimulations. *Form. Methods Syst. Des.*, 18(1):25–68, 2001.

A Cycle Optimisation

Constraints. We first derive a set of constraints over the clock valuations $v_{i,j}$ that limit the choice of delays to only those that result in a valid execution:

1. $v_{i,1}$ satisfies $G_{(i+1) \pmod n}$ as the jump $e_{(i+1) \pmod n}$ is taken from $v_{i,1}$.
2. If $x \in X_i$ then, $v_{i,0}(x) = 0$ as jump e_i resets clock x.
3. $v_{i,j}$ satisfies $I(q_i)$, as all valuations in the run must satisfy the corresponding invariant condition. As the location invariant defines a convex region in the clock-space, all valuations between $v_{i,0}$ and $v_{i,1}$ will also satisfy the invariant.
4. $v_{i,1}$ is a direct time successor of $v_{i,0}$. That is, for each i there is a non-negative constant δ (corresponding to the delay taken from $v_{i,0}$) such that for x in $\mathcal{X}$, $v_{i,1}(x) = v_{i,0}(x) + \delta$.

Note that while the first three conditions can easily be expressed in terms of clock difference constraints between the valuations, the final constraint cannot as the constant δ depends on the delay chosen for that step of the run. To get around this issue, we need to reframe the constraints with respect to the delays, rather than the valuations. To achieve this, we define a vector of variables $\mathbf{t} = (t_0, \ldots, t_n)$ with $t_i := \sum_{j=0}^{i-1} \delta_j$. t_i can be thought of the total elapsed time between the beginning of the execution and taking jump e_i. We then define each valuation with respect to these variables. In order to do so we first define a function $\texttt{lastReset} : (0, \ldots, n) \times \mathcal{X} \to 0, \ldots, n$, which given an index and a clock gives the index of the most recent jump that reset that clock:

$$\texttt{lastReset}(i, x) := \begin{cases} max(j \leq i \,|\, x \in X_j) & \text{if such a } j \text{ exists} \\ max(j > i \,|\, x \in X_j) & \text{otherwise} \end{cases}$$

As we are considering only sequences of jumps in which all clocks are reset, $\texttt{lastReset}(i, x)$ is defined for all $i = 0, \ldots, n$ and all $x \in \mathcal{X}$. The value of a clock at a given point of our sequence is simply equal to the total time elapsed since that clock was last reset, so we can use this function to define each valuation in terms of $\mathbf{t}$, being careful to take into account the cyclical nature of the sequence:

$$v_{i,0}(x) := \begin{cases} t_i - t_{\texttt{lastReset}(i,x)} & \texttt{lastReset}(i, x) \leq i \\ t_i - t_{\texttt{lastReset}(i,x)} + t_n & \texttt{lastReset}(i, x) > i \end{cases}$$

$$v_{i,1}(x) := \begin{cases} t_{i+1} - t_{\texttt{lastReset}(i,x)} & \texttt{lastReset}(i, x) \leq i \\ t_{i+1} - t_{\texttt{lastReset}(i,x)} + t_n & \texttt{lastReset}(i, x) > i \end{cases}$$

This definition of the valuations conveniently ensures that a run given in terms of $\mathbf{t}$ will satisfy several of the above constraints. Constraint 2 is satisfied as if $x \in X_i$, then $\texttt{lastReset}(i, x) = i$ which implies $v_{i,0}(x) = t_i - t_i = 0$. Constraint 4 is almost satisfied, as for all x in $\mathcal{X}$, $v_{i,1}(x) - v_{i,0}(x) = t_{i+1} - t_i$, which does not depend on x, however we need to ensure that the delay this corresponds to is non-negative. To satisfy the rest of constraint 4, as well as constraints 1 and 3 we build a set of constraints $\mathcal{C}$ on $\mathbf{t}$. To ensure the non-negativity of delays we add the constraints that $t_{i+1} - t_i \geq 0$ as $\delta_i = t_{i+1} - t_i$. Each clock value of a valuation $v_{i,j}$ is expressed as a linear combination of the elements of $\mathbf{t}$, and each location invariant and jump guard is the conjunction of upper or lower bounds on these clock values, so adding these conditions expressed in terms of $\mathbf{t}$ to $\mathcal{C}$ ensures that all runs in our system will be valid. All the conditions in $\mathcal{C}$ are bounds on linear combinations of elements in $\mathbf{t}$, and as the invariants bound the clock values, each t_i is also bounded, hence $\mathcal{C}$ defines a bounded convex polyhedron over $\mathbf{t}$.

Objective Function. The function we are trying to minimize is $\texttt{Ratio}(\rho) = \texttt{Cost}(\rho)/\texttt{Reward}(\rho)$, which we wish to express in terms of $\mathbf{t}$. The cost and reward functions are:

$$\texttt{Cost}(\gamma) = \sum_{i=0}^{n-1} \texttt{Cost}(q_i) \cdot \delta_i + \sum_{i=1}^{n} \texttt{Cost}(e_i)$$

$$\texttt{Reward}(\gamma) = \sum_{i=0}^{n-1} \texttt{Reward}(q_i) \cdot \delta_i + \sum_{i=1}^{n} \texttt{Reward}(e_i)$$

To get our objective function $\texttt{Ratio}(\mathbf{t})$ we substitute $\delta_i = t_{i+1} - t_i$, giving:

$$\texttt{Ratio}(\mathbf{t}) = \frac{\sum_{i=0}^{n-1} \texttt{Cost}(q_i) \cdot (t_{i+1} - t_i) + \sum_{i=1}^{n} \texttt{Cost}(e_i)}{\sum_{i=0}^{n-1} \texttt{Reward}(q_i) \cdot (t_{i+1} - t_i) + \sum_{i=1}^{n} \texttt{Reward}(e_i)}$$

Solving the System. The set of polyhedral constraints $\mathcal{C}$ can be represented by the inequality $A\mathbf{t} \leq \mathbf{b}$, where A is a matrix with a column for each clock. Each row of A corresponds to one of the constrains in C, with the matching entry in $\mathbf{b}$ being the bound on that linear combination of members of $\mathbf{t}$. As the objective function $\texttt{Ratio}(\mathbf{t})$ is a quotient of affine functions of $\mathbf{t}$, this function in conjunction with our constraint matrix A, as well as the implicit constraint that each t_i be non-negative gives a *linear-fractional program.* To solve the system we can first apply the Charnes-Cooper transformation [5] to give an equivalent linear program. If we rewrite our objective function as

$$\texttt{Ratio}(\mathbf{t}) = \frac{\mathbf{c}\cdot\mathbf{t} + \alpha}{\mathbf{d}\cdot\mathbf{t} + \beta}$$

Our transformed system has variables $\mathbf{y}$ and λ with:

$$\mathbf{y} = \frac{1}{\mathbf{d}\cdot\mathbf{t} + \beta}\cdot\mathbf{t};\ \lambda = \frac{1}{\mathbf{d}\cdot\mathbf{t} + \beta}$$

The equivalent linear program is then given by:

$$\begin{array}{ll} \text{minimize} & \mathbf{c}\cdot\mathbf{y} + \alpha\lambda \\ \text{subject to constraints} & A\mathbf{y} \leq \alpha\lambda \\ & \mathbf{d}\cdot\mathbf{y} + \beta\lambda = 1 \end{array}$$

If $\mathbf{y}^*, \lambda^*$ is the solution to the linear program, the solution to the linear fractional program is $\mathbf{t} = \frac{1}{\lambda^*}\mathbf{y}^*$. From this we can extract the optimal delays as $\delta_i = t_{i+1} - t_i$.

The linear program can be solved with a method such as the simplex algorithm [6]. In the worst case this has time complexity $O(2^n)$, but it has been shown that on average the simplex algorithm terminates in polynomial time [14].

A.1 Strict Inequalities in Guards

As described, the cycle optimisation algorithm only works for DPTA whose guards and invariants contain weak inequalities ($\leq$ and $\geq$). This is because linear programs require the feasible region to be closed for the precise optimum to exist. However we are interested in the minimum cost to reward ratio that infinite runs of the automaton can approach arbitrarily close to, even if this limit itself is not obtained from an actual run. This situation can arise when strict inequalities ($<$ and $>$) are present in the guards on invariants of the DPTA. To solve the system in this case we replace all strict inequalities by weak inequalities and construct and solve the linear program as before, obtaining the solution vertex $\mathbf{t}^*$. We then reintroduce the strict inequalities and check if their exists a feasible solution within an arbitrarily small neighbourhood around $\mathbf{t}^*$. If such a solution exists, runs of the automaton can approach the ratio obtained at $\mathbf{t}^*$ by taking delays arbitrarily close to those given by this solution that are inside the feasible region. If not, this means that the feasible region is empty and that this sequence of jumps does not give rise to any valid cycles.

Let $\texttt{active}_{\leq}$ be the set of weak constraints in $\mathcal{C}$ that are active at $\mathbf{t}^*$, that is the constraints $\mathbf{a}_i\cdot\mathbf{t} \leq b_i$ such that $\mathbf{a}_i\cdot\mathbf{t}^* = b_i$. Let $\texttt{active}_{<}$ be the be the equivalent set of strong constraints $\mathbf{a}_i\cdot\mathbf{t} < b_i$. If this set is non-empty, the current solution is not feasible. Each active constraint defines a hyperplane in $\mathbb{R}^{|\mathcal{X}|}$ on which $\mathbf{t}^*$ lies. To check if a neighbouring feasible solution exists, we determine if there exists a direction $\mathbf{v}$ that we can move in from $\mathbf{t}^*$ that moves off of all of the hyperplanes in $\texttt{active}_{<}$ without moving onto the wrong side of any other hyperplane. Such a direction requires that for all constraints $\mathbf{a}_i\cdot\mathbf{t} < b_i$ in $\texttt{active}_{<}$, $\mathbf{v}\cdot\mathbf{a}_i > 0$ and for all constraints $\mathbf{a}_i\cdot\mathbf{t} < b_i$ in $\texttt{active}_{\leq}$, $\mathbf{v}\cdot\mathbf{a}_i \geq 0$. Such a $\mathbf{v}$ does not exist if for some active strict constraint the other active inequalities can be combined to derive a contradictory constraint. If $\mathbf{a}$ is the normal vector of a constraint in $\texttt{active}_<$, the other constraints are contradictory if there exists a positive linear combination of normal vectors from $\texttt{active}_\leq \cup \texttt{active}_<$ that gives $-\mathbf{a}$, as this implies that both $\mathbf{a} \cdot \mathbf{t} < b$ and $\mathbf{a} \cdot \mathbf{t} \geq b$ are active constraints.

Let $\mathbf{n} \cdot \mathbf{t} < b$ be an active strict constraint. Let $\mathbf{a}_i = (a_{i,1}, \dots, a_{i,k})$ be the ith active constraint at $\mathbf{t}^*$. We want to check if there exists a positive linear combination of these vectors $\alpha_1 \cdot \mathbf{a}_1 + \ldots + \alpha_m \cdot \mathbf{a}_m$, $\alpha_i \geq 0$ equal to $-\mathbf{n}$. Let A be the matrix whose ith row is $\mathbf{a}_i$. We construct a linear program over $(\alpha_1, \dots, \alpha_m)$:

$$\begin{array}{ll} \text{minimize} & \alpha_1 + \ldots + \alpha_m \\ \text{subject to constraints} & A^T \cdot \begin{bmatrix} \alpha_1 \\ \vdots \\ \alpha_m \end{bmatrix} = \mathbf{n} \\ & \alpha_1, \dots, \alpha_m \geq 0 \end{array}$$

If this program has a feasible solution then the constraint $\mathbf{n} \cdot \mathbf{t} < b$ is contradicted by the other active constraints, meaning that this sequence of jumps does not have a valid execution. This is checked for each constraint in $\texttt{active}_<$.

A.2 Ensuring Reachability

While the execution found by the cycle optimisation algorithm satisfies all the guards and location invariants, the states involved are not necessarily reachable from the starting state of the automaton. To restrict the result to only reachable cycles, additional computation is required. Through the use of zones and the discrete-successor operation, we can construct the *simulation graph* - a graph whose nodes are locations paired with a zone representing a set of reachable states of the automaton and whose edges correspond to jumps of the automaton [11]. The simulation graph is constructed by performing a depth-first search beginning from a zone containing the starting state and its time successors. By taking all the nodes of the simulation graph associated with a given location we get a union of convex zones representing all the reachable valuations within that location.

Given a sequence of jumps $e_1, e_2, \dots, e_n$ to be optimised, let q be the location from which the jump e_1 is taken from, and let $\texttt{Reach}(q) = \{Z_1, \dots, Z_k\}$ be the set of reachable zones at q obtained from the simulation graph. To restrict to reachable cycles, the cycle optimisation algorithm needs to be executed for each zone Z_i in $\texttt{Reach}(q)$. For each execution, the constraints defining Z_i are added to the linear fractional constraint in the same way as the location invariant at q. This ensures that the states in q that the feasible cycles visit are contained in Z_i and are therefore reachable. By definition, if one of the states of the cycle is reachable, the entire cycle is reachable. The algorithm is run for each zone in $\texttt{Reach}(q)$, with the cycle giving the best ratio returned as the result.

B Parameter Tuning

For comparison between the different methods, the total number of solutions evaluated was kept fixed at 3000. Several experiments were carried out to determine what combination of population size and number of iterations gave each algorithm the best average performance, measured in terms of the normalised average ratio the method returned over 10 trials with each set of parameters. The best performing parameters were then used for the main

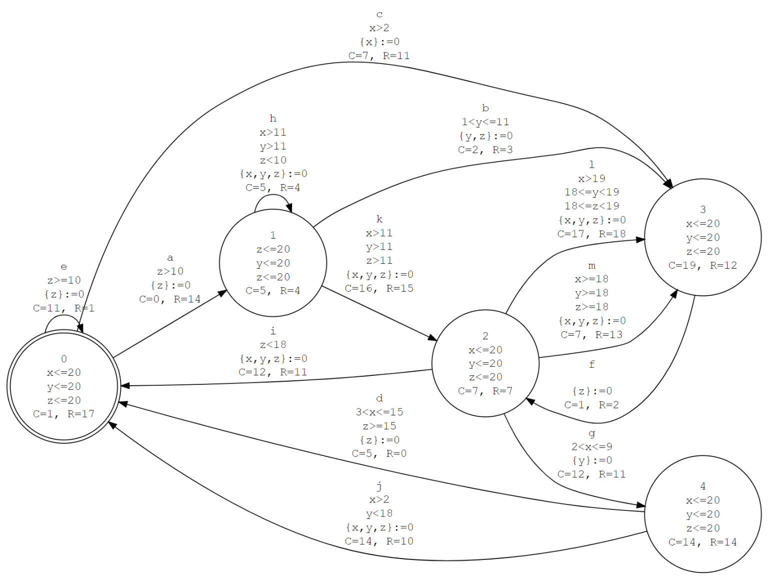

Figure 7 A Generated DPTA with $C = 3, L = 5, J = 5$. Guards and invariants are denoted by inequalities on the clock variables x, y, and z. $X := 0$ denotes a clock reset. C and R denote costs and rewards respectively.

experiments. The ant colony optimisation algorithms perform best with population of 30 ants over 100 iterations. For probability calculations the weighting for pheromone was $\alpha = 1$ and the weighting based on jump and location ratios was $\beta = 1.5$. The evaporation rate used is 0.05 and the decay rate in the local pheromone updates is 0.1. Elitist updates were used with the top 10 ants contributing in the global pheromone update.

C Test Case Generation

Test cases were procedurally generated, ensuring that each automaton met the assumptions of being strongly-reward diverging, deadlock free and bounded. The procedure for generating an automaton takes three inputs: the number of clocks C, the number of locations L, and the minimum number of jumps J. One test case was generated for each assignment of these variables with $1 \leq C \leq 5$, $L \in \{5, 10, 15, 20, 25\}$ and J taking values between L and 45 inclusive at increments of 5. Generation of an automaton begins with a single location with no jumps.

The automaton is built up by either adding a jump from a location to itself, or splitting a location into two. Splitting a location q consists of adding a location q' with a jump from q to q'. Either all of q's inbound jumps are changed to have their destinations be q' or all of q's outbound jumps are changed to have their source be q'. These processes are repeated until the required location and jump counts are reached. All locations have invariants restricting all clocks to be less than or equal to 20. Costs and rewards of locations and jumps take random integer values between 0 and 20, with the exception of jumps added as part of cycles, which have a minimum reward of 1 to ensure reward divergence. Guards and resets are also determined randomly.

Zone based reachability analysis is used to check that all locations are reachable and all jumps can be taken from at least one state. Jump guards are randomly weakened until this is satisfied. Finally, the automaton is checked for deadlocks, again using zone based analysis.

When a deadlock is detected an additional random jump is added that can eventually be taken from all states within that deadlock zone. This process repeats until the automaton is deadlock free. An example of one of the smaller automatons generated via this method is shown in Figure 7.

D Optimal Ratio Calculation

To evaluate the effectiveness of the heuristic methods, it is desirable to know the actual optimal ratio for each generated timed automaton. As a feasible algorithm for calculating this ratio exactly does not currently exist, we instead find the optimal ratio up to a certain cycle length N. To do this we perform a depth first search to generate each discrete cycle of lengths from 1 to N and pass these into the `CycleOpt` function. If the branching factor of the timed automaton (the maximum number of jumps coming out of a single state) is b, then this procedure has worst case time complexity $O(2^N \cdot b^N)$. To balance the amount of time required and the degree of accuracy a maximum cycle length of 8 was used to process each of the 175 test cases.