AALBORG UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE

MASTER THESIS

Priced Timed Automata and Monte Carlo Tree Search

Authors:
Jeppe Høiriis Mortensen
Adriana Mijacika

Supervisors:
Kim Guldstrand Larsen
Peter Gjøl Jensen

AALBORG UNIVERSITY

June 12, 2020

AALBORG UNIVERSITY
STUDENT REPORT

Department of Computer Science
Aalborg University
Selma Lagerlöfs Vej 300
DK-9220 Aalborg Ø
http://www.aau.dk

Title:
Priced Timed Automata and
Monte Carlo Tree Search

Project Period:
Spring Semester 2020

Project Group:
mi103f20

Participant(s):
Jeppe Høiriis Mortensen
Adriana Mijacika

Supervisor(s):
Kim Guldstrand Larsen
Peter Gjøl Jensen

Copies: 1

Page Numbers: 42

Date of Completion:
June 12, 2020

Abstract:

Scheduling problems are one of the problems that we can model with (Priced) Timed Automata and solve by reducing it into the cost-optimal reachability problem. However, finding the cost-optimal solution in such problems can be exhaustive in terms of memory and computational costs, due to the huge state-space exploration. We propose an algorithm for finding the (cost)-optimal solutions, UCT-PTA, an adaptation of the Monte Carlo Tree Search variation UCT, to PTAs. Specifically in this thesis, we investigate different ways of modelling the state space of PTAs, especially how to handle time. We propose, investigate and experiment with three variations: One which handle time increments in a stochastic fashion, one that explores between all the possible time increments and one that explores in the non-lazy time increments. We find that the former approach, namely Non-Lazy UCT-PTA, gives the best results, although not the cost-optimal. We then explore some additional extensions that turn out to improve the overall performance of the UCT-PTA algorithm.

Contents

1 Introduction

Real-time systems are involved in many aspects of our everyday lives such as transport, communication networks, embedded systems etc. The correctness of these systems is conditioned by real-time constraints as timing of events or availability of resources. One type of models that enables the analysis of such time-dependent behaviour is that of Timed Automata [3]. However, Timed Automata themselves does not provide any other quantitative information than time. If one wants to optimise the performance of a system, which is often the case, it is necessary to take into consideration some additional information. In the case of embedded systems it could be the energy and memory consumption, or a required bandwidth in the context of communication networks. In general, such system's attributes can be considered as a time-dependent cost of certain events in the system or a cost of the time elapsed in a certain state. To model these additional quantities an extension of Timed Automata, Priced Timed Automata (PTA), was formalised [5]. One of the problems PTAs allows us to solve are the scheduling problems by reduction into the cost-optimal reachability problem. Due to often large state-spaces of PTAs, a number of different techniques have been tried to optimize the search. In this thesis we aim to improve the efficiency of the state-space exploration by implementing Monte Carlo Tree Search (MCTS) algorithm into the UPPAAL model checking tool [7] and we test its performance on a large number of Job shop scheduling problems.

In the following sections we first give the definition of the Job Shop Scheduling problem and a way we can model it in UPPAAL. Then we present the Monte Carlo Tree search algorithm, its potential in finding optimal solutions and the version of it called Upper Confidence Bounds for Trees (UCT) which main objective is solving exploitation-exploration dilemma that occurs in large state-spaces like ones of PTA. In the Chapter 5 we describe the UCT-PTA algorithm; Firstly we argue the applicability of the UCT in the context of PTA and necessary adaptations to the algorithm, resulting with the UCT-PTA. Then we discuss some challenges of the UCT-PTA, the way we have solved them and the problem of handling time in the context of PTA. We show why the exploration of PTA's state-space in our first version of UCT-PTA from our prior work [15] , Stohastic Delay UCT-PTA, could be problematic. In this thesis, we then explore other ways of handling time. In the proposed approaches, namely Delay Exploratory and Non-Lazy UCT-PTA, we solve the limitation of the only one-delay as a possible delay transition. Additionally, we introduce several extensions that also improve the UCT-PTA's performance: pruning by "making a step" (described thoroughly in subsection 5.4.1) and by introducing the Partial order reduction [8] to the UCT-PTA. Lastly, we experiment with all of the presented UCT-PTA variations on the presented job shop scheduling problems and some task graph problems and discuss the results.

2 Job Shop Scheduling

The Job shop scheduling problem is an optimization problem that aims to optimally schedule a set of jobs to be processed with given resources. Concretely, we are given a set of n jobs $J_0, \dots, J_n$ with varying processing times that require resources defined as a set of m machines $M_1 \dots M_m$. A duration $d_{ij} \in \mathbb{R}_{\geq 0}$, where $i \in \{1 \dots n\}$ and $j \in \{1 \dots m\}$, denotes the time that job J_i needs to be processed by the machine M_j. Additionally, for each job J_i there is a preferred processing total order $\leq_j$ on machines M_j.

What we want to find is a schedule $\sigma : \{1 \dots n\} \times \{1 \dots m\} \implies \mathbb{R}_{\geq 0}$ such that if $\sigma(i,j) = t$ then job J_i will start being processed at time t on machine M_j. Moreover the schedule σ has to be safe which we ensure by satisfying the following constraints:

1. $\forall_i \forall_{j_1,j_2} : \sigma(i, j_1) < \sigma(i, j_2)$ iff $j_1 < j_2$ saying that the order of the processing machines has to be respected.
2. $\forall_j \forall_{i_1,i_2} : \sigma(i_1, j) < \sigma(i_2, j) \implies \sigma(i_1, j) + d_{i_1,j} < \sigma(i_2, j)$ where $i_1 \neq i_2$ i.e machine M_j cannot process more than one job at the time.

For a safe schedule σ we define its length as $span(\sigma) = \max_{i,j}(\sigma(i,j) + d_{i,j})$. The schedule σ we want to find has to be optimal, that is $span(\sigma) \leq span(\sigma^*)$ for any other safe schedule σ^*.

One example of the Job shop problem is the Newspaper problem where a given number of people have to finish reading a certain number of chapters in a newspapers in a certain order. When we observe this problem in the context of the Job shop scheduling problem, we say that the machines are the sections while reading all of the sections is a job. Furthermore, the machine's limited capacity is defined with the restriction that each chapter can be read by the exactly one person at a time. The aim is to find a schedule for which the total time required for everybody to finish reading the newspaper is minimal. Let us present a concrete example where two persons want to read a newspaper containing four sections.

In Figure 2.1 we observe a case where Jan and Wang are reading a newspaper consisting of four sections: Sport, Economy, Local News and Comics. A section can be read by only one person at a time. Each person reads the sections in the given order requiring the annotated amount of time. Let us now consider two schedules (not the only possible ones) as shown on Figure 2.2.

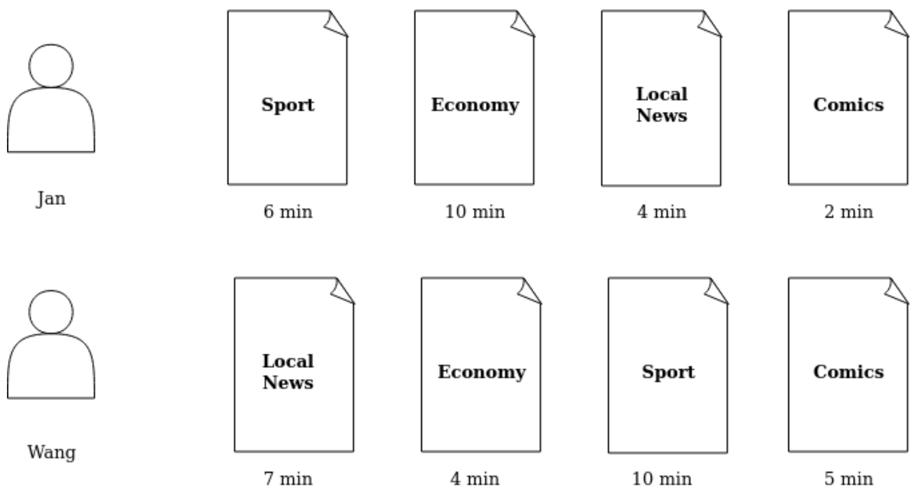
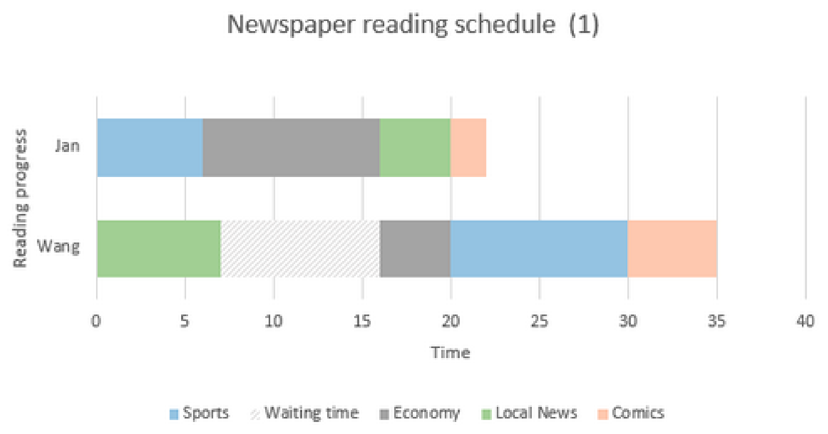


Figure 2.1: The Newspaper problem example with two people. Sections have to be read in the given order and within the time noted underneath each section.



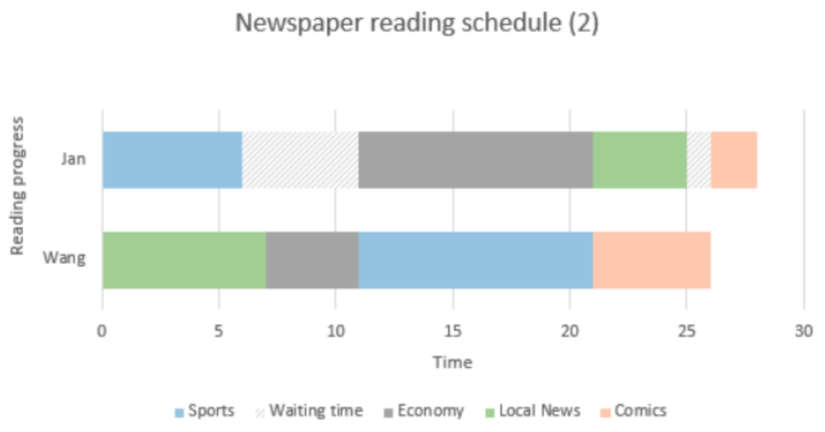


Figure 2.2: Potential optimal schedules for the described instance of the Newspaper problem

In the beginning, both Jan and Wang can start reading as each of them is reading a different section. In the first schedule Jan finishes his first section after six minutes and immediately starts reading the second section: *Economy*. In the seventh second Wang is also done with reading his first section, however he has to wait until Jan finishes reading *Economy* section. Once he does, both can proceed with reading rest of the sections. With this schedule it would take 35 minutes for both readers to finish reading their sections. In the second schedule the difference is that Jan does not start reading the *Economy* section immediately after finishing his first one, but waits for Wang to read it first. With this schedule, as seen on the chart, we achieve more of the time Jan and Wang are reading simultaneously which results in less time for both to finish reading all of the sections - 28 minutes, therefore this is the more optimal schedule of the two.

Scheduling problems often have a defined cost which has to be taken into consideration. The goal in that case, is to find the cost-optimal schedule which minimizes the accumulated cost. To illustrate this we define a waiting cost within the Newspaper scheduling problem: whenever a person has to wait, that cost a certain rate per time unit. For example, we define the cost of Jan's waiting time to be five per minute and the cost of Wang waiting to be two per minute. If we now analyze the schedules from Figure 2.4 they result with accumulated cost as shown on Figure 2.3.

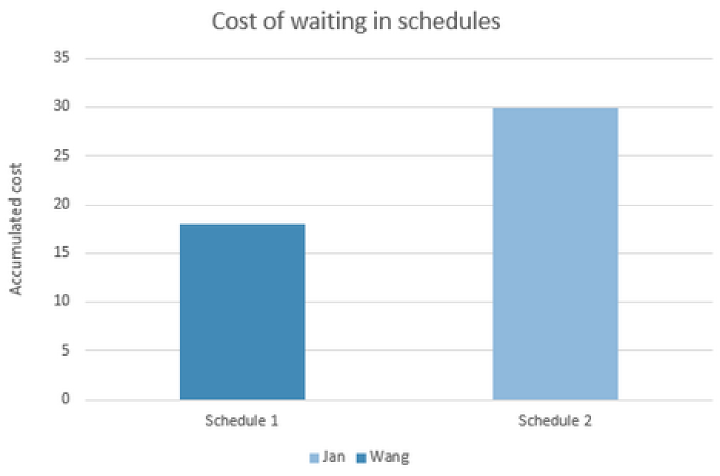


Figure 2.3: Accumulated cost in schedules from Figure 2.2

We see that once the cost is taken into consideration, the first schedule is the optimal one as it results in lower total accumulated cost.

We can imagine how, by adding a new person into the problem, complexity of finding the optimal schedule increases greatly. Therefore a need for an efficient algorithms arises as more and more people are added. In this thesis we use the instances of Newspaper problem with two, four, six, eight, ten and twelve people involved to test our UCT-PTA algorithm. In general, the complexity of the Job shop scheduling has been shown to be a NP-hard problem [16].

This type of the problem can be modelled with a priced timed automata which will allow us to find its cost-optimal schedule. To do so we perform the cost-optimal reachability verification of the priced timed automata which will explore its state-space in order to find the state that satisfies such a condition. In the following section we give a detailed description of the corresponding PTA model.

2.1 The Newspaper Problem in UPPAAL

A priced timed automata model representing the stated Job Shop scheduling problem can be made in a model checking tool like UPPAAL Cora. We create a template as shown in Figure 2.4 which models a set of consecutive jobs (reading sections) to be read by one person. Depending on the number of people defined in the Newspaper problem, we initialize the same number of these templates to model the processes i.e people reading tasks with the defined constraints.

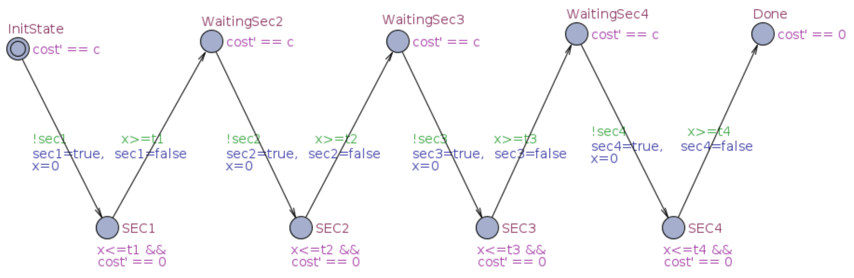


Figure 2.4: Template used in a Job shop scheduling problem: modeling a reading tasks preformed by one person

The template contains the initial location `InitState` representing a state where a person, process of the system, waits for reading of the first section, which is conditioned by its availability (no other person should be reading it). In the model itself this restriction is ensured with the condition `!sec1` being a part of a guard of the following edge; It has to evaluate to true (thus the flag $sec1$ has to be false) for the transition to be enabled. Once it is, the person can change its state by moving onto the next location and start reading the section. Similar "waiting" locations `WaitingSec2, WaitingSec3, WaitingSec4` represent the states in which a person waits for reading the second, third or fourth section respectively. All of them are annotated with the invariant `cost' == c` simulating the cost increase in the system whenever a person has to wait for reading a following section. Therefore, in such a state the cost increases by the rate c, defined differently for each per-

son.
Locations `SEC1, SEC2, SEC3, SEC4` represents the states in which a person is reading a given section. While being in these locations, the cost should not increase thus its rate is defined with `cost' == 0`. As our problem defines the exact reading duration of each section per person, this time restriction of the system's behaviour has to be formulated. We model it by defining `x>=tx` as the following edge's guard condition and with `x<=tx` as an invariant of the reading location - these are in fact lower and upper limit of the sojourn in the reading location. By having the stated guard we ensure that the process can leave the reading location only when the clock x reaches tx i.e. tx time units have to pass, while with the invariant condition we ensure that the process cannot stay in the same location for more than that time. If the problem would have been defined in a way that the reading activity is not deterministic in terms of the time duration but within a time range, these conditions make it easy for us to modify the model accordingly.
When a person finishes reading all four sections, the person goes to the final location `Done`.

With this defined template we can model the instance of Newspaper problem example described in the chapter 2. The resulting model is shown on Figure 2.5.

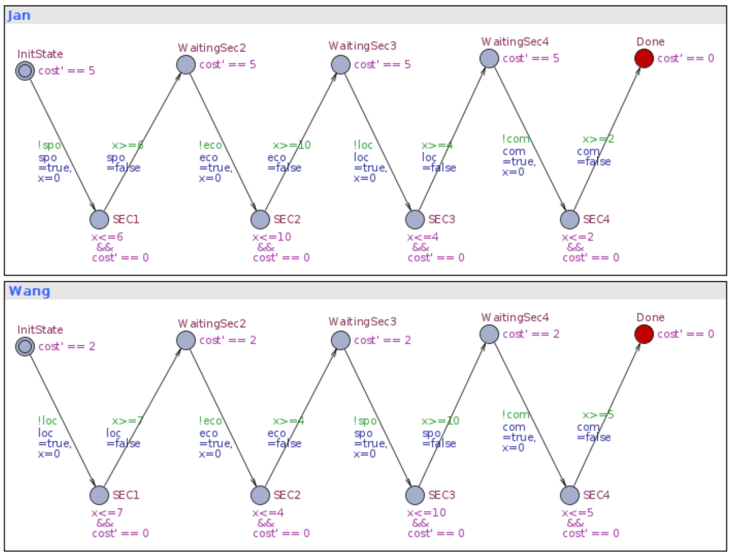


Figure 2.5: Model for the Newspaper problem with two people reading four sections

3 Priced Timed Automata

In this section we shall now introduce a more formal definition of the described model and its semantics.

Let $C = \{x, y...\}$ be a set of clocks - non-negative real valued variables that all grow at the same rate equal to one , measuring the amount of time passed since the last reset. We define the notion of clock valuations as $v : C \to \mathbb{R}_{\geq 0}$. Passage of t time units is annotated with $v + t$ which maps each clock $c \in \mathbb{C}$ to the value $v(c) + t$.
Clocks are used to restrict a behaviour of the model in several ways, which will be explained later in the text, after first presenting the elements of a priced timed automata. A priced timed automata over C is an annotated directed graph with vertices representing the locations and edges representing the transitions between them. Each transition is associated with a guard g, an action Act and a reset set r, while locations are labeled with invariants $I(l)$. A reset set r contains all the clocks that needs to reset after a given transition is taken, which is denoted with $r \mapsto 0$. Guards and invariants are the conjunctions of simple constraints $c \bowtie k$, where c is a clock in C, k is a non-negative integer value and $\bowtie \in \{<, \leq, =, \geq, >\}$. The set of all clock constraints is denoted with $B(C)$ and allow us to restrict a behaviour of the models with respect to the values of clocks. More precisely, guard conditions have to be satisfied to enable a transition, while invariants define conditions that have to be continuously satisfied for a system to stay in a given location. These satisfactions of conditions we denote with $v \models g$ for $g \in B(C)$.

The definition given up until this point, is also the definition of a timed automata. However as we also wants to consider the costs within a system, we use the extended definition with additional annotations of costs and cost rates on the edges and the locations, respectively. The definition of a priced timed automata, as described in [5] is given below.

Definition 1 ***Priced Timed Automata (PTA)***
A Priced Timed Automata over a set of clocks C and actions Act is a tuple (L, l_o, E, I, P), where L is a set of locations, l_0 is the initial location, $E \subseteq L \times B(C) \times Act \times 2^C \times L$ is a set of edges where an edge, being between two locations, contains a guard, an action and a set of clocks to be reset. $I : L \to B(C)$ assigns invariants to the locations and price $P : L \cup E \to \mathbb{N}$ assigns cost rates and costs to locations and edges, respectively. In the case of $(l, g, a, r, l') \in E$, we write $l \xrightarrow{g,a,r} l'$.

In PTA a state s is a pair (l, v) where $l \in L$ denotes a location and v a clock valuation such that invariant $I(l)$ is satisfied. We consider clock valuations of the initial state to be 0, therefore we define $v_0(c) = 0$ for all $c \in C$.

The following semantics of a PTA defined as a priced transition system is as described in [5].

Definition 2 ***Semantics of Priced Timed Automata***
The semantics of a PTA $A = (L, l_0, E, I, P)$ over clocks C and actions Act is given by a priced transition system $T = (S, s_0, \Sigma, \rightarrow)$, where $S = \{(l, v) \in L \times \mathbb{R}^C_{\geq 0} : |v \models I(l)\}$ is the set of states where each state consists of a location l and a clock valuation v such that the invariant of l evaluates to true, $s_0 = (l_0, v_0)$ is the initial state, $\Sigma = Act \cup \mathbb{R}_{\geq 0}$ is the set of labels and $\rightarrow$: $(S \times \Sigma \times \mathbb{R}_{\geq 0} \times S)$ are the transitions labeled either with an action or a delay and with a cost p.

The cost p equals to the cost rate $P(l)$ per time unit of staying in the location l, in the case of a delay, either to the $P(e)$ which is the cost of the edge in a case of action transition . With respect to its label we differ between two types of transitions:

Discrete transitions, defined as

- $(l, v) \xrightarrow{a}_p (l', v')$ iff $\exists e : e = (l, a, g, r, l') \in E$, $v \in g$, $v' = v[r \mapsto 0]$, and $p = P(e)$

Delay transition, defined as

- $(l, v) \xrightarrow{d}_p (l, v + d)$ iff $\forall 0 \leq d' \leq d : v + d' \in I(l)$ and $p = d \cdot P(l)$

where for $d \in \mathbb{R}_{\geq 0}$, $v + d$ denotes an increase of v by delay d for each clock c in $\mathbb{C}$ and $v[r \mapsto 0]$ denotes a reset of clocks in the reset set r and agrees with v over $\mathbb{C} \setminus r$.

An example of PTA from Definition 1 is shown on Figure 3.1.

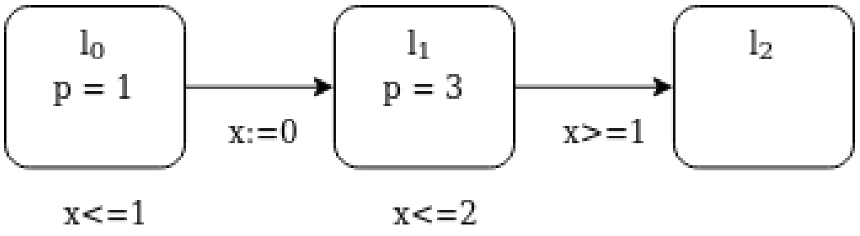


Figure 3.1: An example of Priced Timed Automata

Some of the possible traces of this PTA defined as a transition system $T(A)$ are following:

- $(l_0, x = 0) \rightarrow_0 (l_1, x = 0) \xrightarrow{d=1}_3 (l_1, x = 1) \rightarrow_0 (l_2, x = 1)$
- $(l_0, x = 0) \rightarrow_0 (l_1, x = 0) \xrightarrow{d=2}_3 (l_1, x = 2) \rightarrow_0 (l_2, x = 2)$
- $(l_0, x = 0) \xrightarrow{d=1}_1 (l_0, x = 1) \rightarrow_0 (l_1, x = 0) \xrightarrow{d=1}_3 (l_1, x = 1) \rightarrow_0 (l_2, x = 1)$

4 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) has shown to be a good method for finding an optimal solutions in domains such as planning problems, which can be represented as a tree of sequential decisions. In this section we will provide a brief description of the basic MCTS algorithm, its advantages and disadvantages as well as its popular version called Upper Confidence Bounds for Trees (UCT). The information we present about MCTS are the ones relevant for understanding our UCT-PTA algorithm and the rest of the paper, however for a more interested reader we recommend the thorough Survey on MCTS research provided in [9].

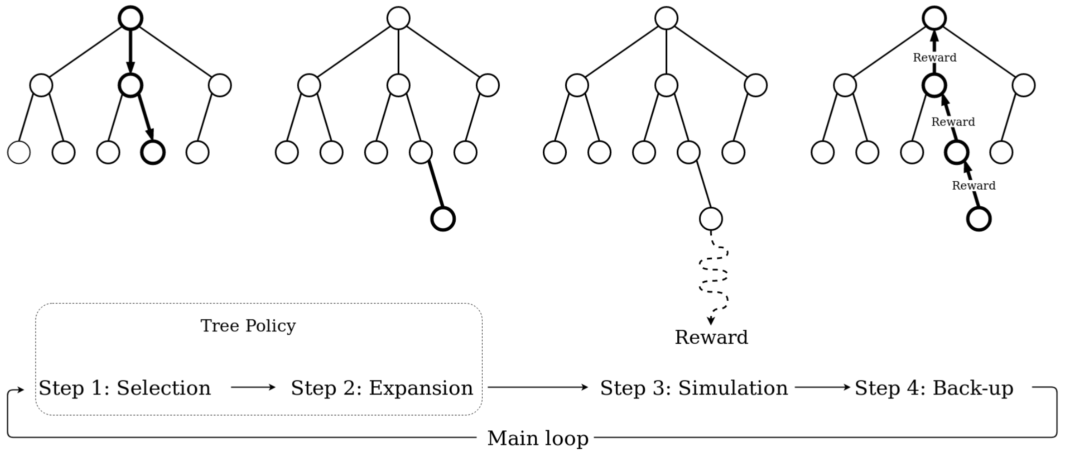


Figure 4.1: The four different steps of the MCTS main loop. A rendition of a similar figure from [9]

4.1 The Basic Algorithm

In this section we will first give the formal definition of the MCTS tree structure and then we explain the steps of the search algorithm, itself.

Definition 3 ***MCTS Tree Structure***
Let $\delta = (S, \rightarrow)$ be a transition system with state set S and transition relation $\rightarrow \subseteq S \times S$. An MCTS tree over δ is a tuple $T = (N, E, S, i, Q, P, Y, Y_U, Y_E)$ where N is a set of nodes, $E \subseteq N \times N$ are the edges, the set of states is $s : N \rightarrow S$ and we assign for each node $n \in N$ a state $s(n) \in S$. Specifically, with s_0 we define the start state of δ and with n_0 we define the root node of T.
Each node then has a defined number of visits $i : N \rightarrow \mathbb{N}$ where given $n \in N : i(n) \in \mathbb{N}$, accumulated award $Q : N \rightarrow \mathbb{R}$ where given a $n \in N : q(n) \in Q$, and a parent node $P : N \rightarrow N$ such that $(p(n), n) \in E$ for all $n \in N$. Also, for each node we define children nodes being

$Y : N \to P(N)$ such that $Y(n) = \{m \mid (n,m) \in E \wedge n = p(m)\}$, for all $n \in N$. Additionally we define subsets of unvisited children as $Y_U \subseteq C$ and explored children as $Y_E \subseteq C$, whereas $Y_U \cap Y_E \neq 0$ and $Y_U \cup Y_E = Y$.

Finally we require the following correspondence between the transition relation $\to$ of δ and the edges E of T:

- $\forall m \in Y(n) : s(n) \to s(m)$
- $\forall t \in S : s(n) \to t \implies \exists m \in N : (n,m) \in E \wedge s(m) = t$

and we define set of children states $Y(S)$ where $\forall m \in Y(n) \wedge s(n) \to s(m) \implies s(m) \in Y(S)$.

MCTS, being a tree search algorithm, builds a tree as one from the Definition 3 incrementally and asymmetrically, in the direction of the nodes leading to the promising scores. Each node in the tree represents a state $s(n)$ of the underlying domain that MCTS is applied to. Therefore, child nodes from a given node are in fact reachable states from the given node's state and choosing a child corresponds to taking an action that would lead to that child state. Additionally, node n consists of a state $s(n)$ with an assigned score $Q(n)$ being the accumulated reward during the search and a visit count $i(n)$ in the form of a non-negative integer, noting the frequency of visits of a given state i.e node. Furthermore, given a state s all the child states of $s(n)$ are contained in the corresponding child nodes $Y(s)$.

Build of the tree is achieved in an iterative fashion consisting of four sub-steps in each iteration, as described at Figure 4.1:

1. Selection:
   The first step consists of selecting a node to expand. In this step a child node is being selected from the root node, recursively down the tree until the unexpanded node is encountered. Unexpanded node is defined as a non-terminal node with previously unvisited children. Once such node is reached the *Expansion* step follows. While descending down the tree, a policy is being used to select the "best" of the child nodes, leading towards promising scores.

2. Expansion:
   From the node chosen by the selection step, one of the unvisited child nodes is randomly chosen and expanded.

3. Simulation:
   From the newly expanded node, a simulation is run. The simulation provides some approximation of the value of the expanded node. This usually happens by continually taking actions according to some simulation policy, usually called a *Default policy*,

until a terminal state has been reached or a condition is met. The simulation should provide a score (reward), for the given expanded node.

4. Backpropagation:
   The score returned from the simulation is backed-up through the tree. It updates all the nodes that were selected during the selection step with the score.

Step 1 and 2 are usually grouped into a so called *Tree policy.* The simulation step is also often called playout or rollout (of the *Default policy).* From the algorithms side, the iteration has no predefined stopping condition, however in practice it usually continues until a given computational budget has been reached or some domain specific conditions have been met. When the search terminates, a child node is chosen from the root node, which corresponds to taking the action that leads to the state the child node represents. The selected node is usually chosen in the same manner as in the selection step of MCTS, by performing the BestChild function. However it can also be chosen according to the other mechanisms, some of them mentioned in the Survey [9]. There, four selection mechanisms are outlined as possible choices for selecting the best action: *Max Child* which chooses the child with highest reward; *Robust Child* chooses the most visited child; *Max-Robust Child* chooses both max reward and number of visits visits and continues the search until one exists and *Secure Child* which chooses the child that maximises a lower confidence bound. In general, if the domain requires it, it is possible to call the *BestChild* multiple times on the final search tree, which will effectively "unroll" the best path found so far. The pseudocode of the algorithm is given in algorithm 1.

Algorithm 1: Basic MCTS Algorithm

```
Function MCTSSearch($s_0$)
    from $s_0$ create root node $n_0$
    while stopping condition not met do
        $n'$ = TreePolicy($n_0$)
        reward = DefaultPolicy($n'$)
        BackUp(reward, $n'$)
    end
    return BestChild($n_0$)
```

4.2 Characteristics of MCTS

MCTS has a number of characteristics which makes it an attractive choice for its application in diverse domains. We list a short description of the most important ones:

Anytime: After each MCTS iteration the complete search tree is up-to-date with all the knowledge gained so far. This means that if one would like to take an action based on the search, it is always possible to stop the iteration and evaluate which child is the (currently known) best choice. However, the performance generally improves with more iterations.

Aheuristic: MCTS, and a lot of its extensions, does not need any information of the underlying domain. This makes MCTS easily usable in any problem that can be structured as a tree search. With that being said, extending MCTS policies with domain-knowledge often leads to improved performance in that domain.

Asymmetric search: MCTS explores the search tree in an asymmetric fashion. It focuses on promising directions of exploration while still looking at less promising directions every now and then.

Easily Extendable: The four different steps of MCTS: selection, expansion, simulation and back-up, are easy to change and extend to alter the behaviour of the algorithm to match a domain, which can improve the performance of the algorithm.

4.3 Upper Confidence Bound for Trees

The basic MCTS algorithm has a exploitation-exploration dilemma during the *Tree policy* step, when considering which child nodes should be selected. In [4] they propose a solution to this dilemma, by addressing it as a multi-armed bandit problem and using UCB1 as the *Tree policy*. UCB1 is a great choice for it, as it is guaranteed to be within a constant factor of the best bound for regret, which makes it an ideal solution to the exploitation-exploration dilemma. Furthermore, it is quite simple and computationally inexpensive, which is a necessity, since the *Tree policy* will be computed at each step during the descension of the search tree.
UCB1 chooses the maximum arm computed by:

$$UCB1 = \overline{X_j} + \sqrt{\frac{2 \ln n}{n_j}} \tag{4.1}$$

In the Algorithm 2 is the pseudo-code implementation of UCT in a slightly changed version of the one presented in [9]. The main takeway is the UCB1 implementation in the BestChild function where $\overline{X_j}$ is the average reward of arm j, n is the total number of plays and n_j is the total number of plays of arm j. In the context of UCT-PTA which will be introduced in the section 5.2, the arm is equivalent to the action being taken. The first term expresses the exploitation weight while the second term expresses the exploration one. The UCB1 implementation, being a part of the *Tree policy* is seen in the BestChild function at line 15, with the MCTS notation integrated. Note that an additional value C_p has been added to the exploration term, as to be able to deactivate and/or control how much exploration the algorithm should do.

Kocis and Szepesvári [4] proved that, when running UCT, as the number of searches grows towards infinity, the chance of selecting a sub-optimal child at the root node converges towards zero at a polynomial rate. This effectively means that UCT converges to the minimax tree if run for enough time.

Algorithm 2: UCT - slightly changed rendition of Algorithm 2 in [9]

```
1  Function UCT-TreePolicy($n$)
2      while $n$ is non-terminal do
3          if $Y_U(n)$ is non-empty then
4              return Expand($n$)
5          else
6              $n$ = BestChild($n$, $C_p$)
7          end
8      end
9      return $n$

10 Function UCT-Expand($n$)
11     choose $s' \in Y_U(s)$ uniformly at random
12     create $n'$ such that $s(n') = s'$, $i(n') = 0$, $q(n') = 0$, $p(n') = n$, $Y_E(n') = \emptyset$ and
        $Y_U(n') = Y(n')$
13     add $n'$ to $Y_E(n)$
14     return $n'$

15 Function UCT-BestChild($n, C_p$)
16     return $\underset{n' \in Y(n)}{\arg\max} \frac{Q(n')}{i(n')} + C_p \sqrt{\frac{2 \ln i(n)}{i(n')}}$

17 Function UCT-DefaultPolicy($s$)
18     while $s$ is non-terminal do
19         choose $s' \in S(s)$ uniformly at random
20         $s = s'$
21     end
22     return reward for $s$

23 Function UCT-BackUp($n, reward$)
24     while $n$ is not null do
25         $i(n) = i(n)+1$
26         $Q(n)$ += $Q(n)+reward$
27         $n = p(n)$
28     end
```

5 UCT-PTA

5.1 Applying MCTS to PTAs

MCTS is an aheuristic algorithm, and as such it generally does not care about the underlying domain. This also applies to using PTAs as the domain upon which an MCTS algorithm can act. However, while the MCTS algorithm may not necessarily differ between domains, it is still very important how the underlying domain is translated into the MCTS search tree. This is true for the correctness of the algorithm, but also for optimizing the performance. In this section we define the representation of PTAs in the MCTS tree structure. We call the resulting tree structure the *UCT-PTA Tree Structure*, since it is a part of the final UCT-PTA algorithm.

Considering the definitions of a state s from both Definition 3 and Definition 2 of the semantics of PTAs we can precisily define UCT-PTA Tree structure.

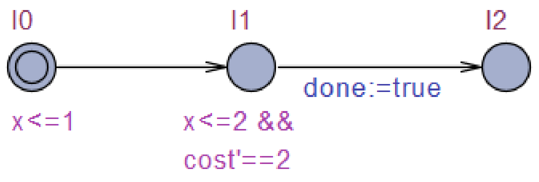
Definition 4 ***UCT-PTA Tree Structure***
UCT-PTA tree T_A for Priced Timed Automata A is an MCTS tree from Definition 3 where the state set is defined as $S = \{(l, v) \mid l \in L \; and \; v : C \rightarrow \mathbb{R}\}$ and the transition relation $\rightarrow$ is restricted to unit delays: $\rightarrow \subseteq S \times (Act \cup \{1\}) \times S$.

Furthermore, to fully enable the functionality of MCTS, we define a set of functions available to the MCTS algorithm. We call these functions the *Environment interface*, as it provides the necessary information from the PTA's environment for MCTS to function.

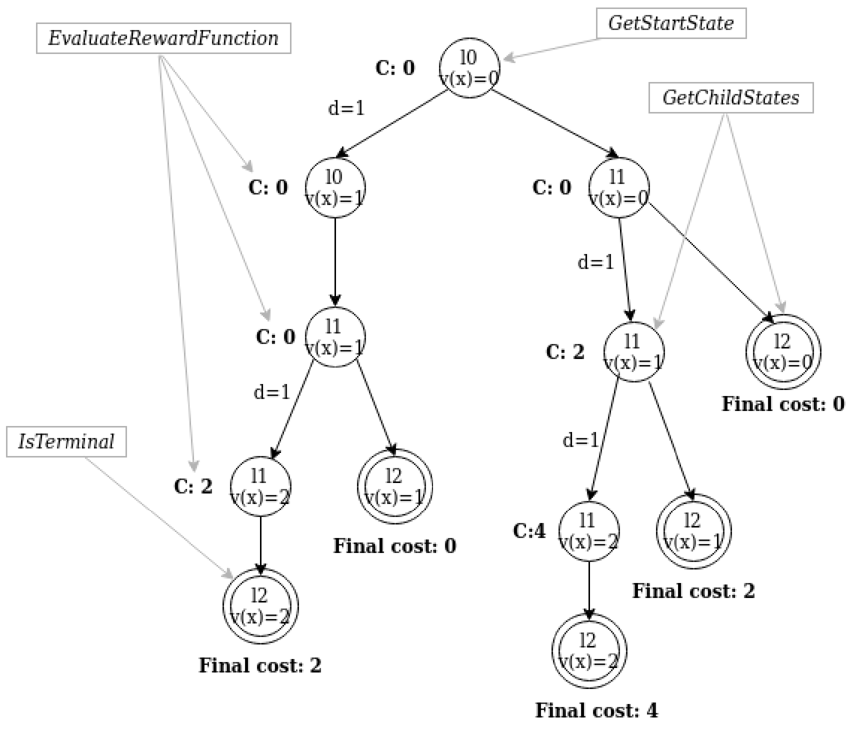
Definition 5 ***Environment Interface***

- *$GetStartState$ returns the initial state s_0.*
- *$IsTerminal$ takes a state s as input, and returns a boolean value that represents wether the state is terminal or not.*
- *$GetValidChildStates$ takes a state s and returns all the possible successor states, described as $Y(s)$ in Definion 4.*
- *$EvaluateRewardFunction$ takes a state s and returns the underlying reward value of that state.*

Changing and/or extending these definitions can drastically affect the tree structure, as will changes to the UCT-PTA algorithm in general. Later, we will explore different variations of UCT-PTA which end up with different tree structures, which again affects the searching in various manners. In Figure 5.1 a small PTA toy example is illustrated along with the fully expanded UCT-PTA search tree. Note again, that UCT-PTA does not use all of the information presented in Figure 5.1b, but rather only the information it retrieves from the *Environment interface*, also presented in the figure where it is relevant. The fully expanded search tree represents the entire state space. However, in practice UCT-PTA will almost always explore only a small fraction of the search tree.



(a) A PTA toy example.



(b) The fully expanded search tree structure of the PTA toy example. Location, clock valuations, cost (annotated with **C**) and terminality are annotated for each node. Delay transitions are annotated (d=1) for the relevant edges.

Figure 5.1: An example of a PTA (5.1a) and the MCTS tree counterpart (5.1b). Note that this search tree construction is specific to the Stochastic Delay UCT-PTA, described in subsection 5.2.2.

5.2 UCT-PTA

In this section we introduce the UCT-PTA algorithm which is our implementation of MCTS for PTAs. We choose the UCT variation of MCTS since it has a lot of valuable properties, as seen in section 4.3, thus it is well known and widely used.

The search-tree structure defined in Definition 4, the *Environment interface* from Definition 5 and the original UCT algorithm in Algorithm 2 combined with the changes in Algorithm 3, explained in this section, makes up the first version of the UCT-PTA algorithm which we call Stohastic Delay UCT-PTA. Later, we will also look into the other variations of UCT-PTA.

5.2.1 General Challenges

The following section is a reiteration of a section in [15], with only minor changes.
As an aheuristic algorithm, UCT should in theory be applicable to any domain that can be represented as a tree-search. However the PTA domain with cost-optimal reachability do have a few properties that make a pure UCT implementation, the one showed in Algorithm 2, impractical if not directly non-functional. This section will describe those problems, and what changes was made to UCT to counteract them.

Infinite Transitions

MCTS and UCT have mostly been developed for game playing [9], and most games do not continue indefinitely. However, it is easy to imagine a (priced) timed automata with infinite transition sequences for instance the one shown in Figure 5.2.
In UCT the problem arises in the *Default policy*, where transitions are unrolled until a terminal state is encountered, potentially running forever. Our solution is to cap the maximum amount of rollouts to a predefined number as shown on line 15 in Algorithm 3. The number of rollouts itself we implemented as one of the tunable parameters. This however, introduces another problem, the need for being able to evaluate the reward/cost of a non-terminal state. Luckily this is easy for PTAs and UPPAAL, since the PTA model contains all the information necessary to evaluate the current cost of any state.

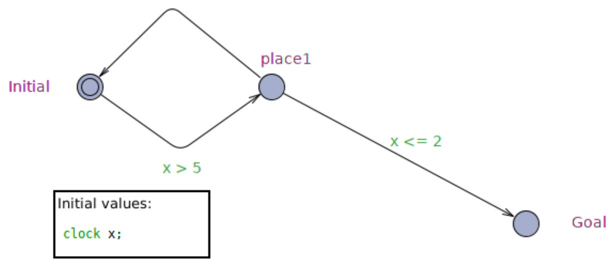


Figure 5.2: A simple example of a TA with infinite transitions. The transitions will keep looping between the *Initial* and *place1* without ever getting to the terminal *Goal* state.

Non-Terminal No-Transition States

UCT is not equipped to deal with states that are not terminal, but have no possible transitions either. In a lot of domains, especially games, a state with no possible transitions is also a terminal state. However, in (priced) timed automatas, a model can easily have non-terminal being at the same time a no-transition state. A classic example of this is a deadlock.

In UCT this problem is encountered in two places, in the *Default policy* and in the *Tree policy*. The former is an easy fix. Here, encountering such a state is added as a reason for terminating the rollout, while afterwards everything happens as usual.
The fix for the latter is more involved. The standard behaviour of the UCT's *Tree policy* is to return the node representing the non-terminal no-transition state and continue the loop as usual. However, the fact that the *Tree policy* choose this node, implies that the current best known transition trace ends in this state, and since the node does not provide any new information for updating the scores, it will continually be chosen by the *Tree policy* until the exploration term kicks in. This might take a lot of search loops, and the node will still be encountered frequently after. The fix is to prune such nodes the first time they are encountered as the output of the *Tree policy*, which is seen at line 8 in Algorithm 3.

Note that it is not possible to encounter such a state in the Job shop scheduling problem, since it consists of a finite amount of jobs that does not depend on each other, takes a finite amount of time to complete and have infinite time available to do so.

Unknown Reward/Cost Range

In [14] Kocsis and Szepesvári found that the ideal C_p value, used in *BestChild* function in the *Tree Policy* (see Algorithm 2 line 15) for rewards in range $[0,1]$ was $C_p = 1/\sqrt{2}$. However, Priced timed automata not only have ranges outside of $[0,1]$, they also vary a lot depending on the underlying model. Furthermore, models can have infinite number of possible rewards , obtained by analyzing the specific PTA itself, thus upper and lower bounds of the reward range are not strictly defined. This poses a significant problem for UCT, where C_p has to be predefined. C_p controls how the balance between exploration and exploitation happens during the search. Having a value not tuned to the reward range will often result in a nonfunctional search. Instead of figuring a C_p value for each model, we decided to normalise the true reward range into $[0,1]$ and use $C_p = 1/\sqrt{2}$. To this we also refer as *bootstrapping*. We normalise a given reward $reward$ with the normalisation function shown here:

$$norm\_reward = \frac{reward - min\_reward}{max\_reward - min\_reward} \tag{5.1}$$

Where min_reward is the smallest reward encountered and max_reward is the largest. To find the reward range, before the main search loop, we started each run with a hundred calls to *Default Policy*, and kept the minimum and maximum reward found, to use for the normalisation. Then in the main loop, any encountered reward outside the range was added as a new range bound. Changing the range over time introduces some undefined behaviour which was not accounted for. However, empirically it did not affect the algorithm throughout the experiments, so no time was spent finding a more elegant solution for now.

Algorithm 3: UCT-PTA: The parts of the UCT that has been changed to accommodate Priced Timed Automata. The other functions has not changed from the UCT as seen in Algorithm 2.

```
Function UCT-PTA-Search($s_0$)
    from $s_0$ create root node $n_0$
    while stopping condition not met do
        $n'$ = TreePolicy($n_0$)
        reward = DefaultPolicy($n'$)
        norm_reward = normalize(reward)
        BackUp(norm_reward, $n'$)
        if $n'$ is non-terminal & $n'$ has no children then
            prune $n'$ from search tree
        end
    end
    return BestChild($n_0$)

Function UCT-PTA-DefaultPolicy($s$, $rollout\_limit$)
    int i = 0;
    while $s$ is non-terminal & $s$ has child states & $i < rollout\_limit$ do
        choose $s' \in S(s)$ uniformly at random
        $s = s'$
        $i++$
    end
    return reward for $s$
```

5.2.2 Handling of Time

Another major problem that arises when accommodating UCT to PTAs is how time will be represented in the tree structure. Time is naturally a crucial part of PTAs, and the representation of it will necessarily have a lot of influence on the exploration of the PTA.

For our initial version of UCT-PTA, Stochastic Delay UCT-PTA, the handling of time is defined as in Definition 4 and introduced in the Section 5.1. We will later look at different variations/extensions of this definition and Definition 5, which results in handling time differently. For now, we investigate potential problems with the time handling of the Stochastic Delay UCT-PTA.

Having a state set as in Definition 4, and a set of child states $Y(S)$ from Definition 3, the only child state that is available with a delay transaction is the one-second delayed state $(l, v) \xrightarrow{d=1} (l, v+1)$. What is important to note here is that a state often has several possible delay values, while with this stochastic approach only a delay of one clock rate is allowed in the current state. The idea here is then, that choosing the delay action then leads to a successor state $(l, v+1)$, which has the next one-delay action allowed that leads to $(l, v+2)$ and so on.
This approach does provide a correct implementation, that leads to a search-tree structure that can represent the whole state space. However, it also introduces some unwanted side-effects.

The main negative consequence of handling time in this fashion, is that it increases the amount of so called *trap states* that will occur in the tree. Ramanujan et al. [17] define a trap state as following:

Definition 6 ***Trap State***
In a two-player game G, the current player p at state s is said to be at risk if there exists a move m from state s such that after executing m, the opponent of p has a k-move winning strategy. The state of the game after executing m is referred to as a level-k search trap for p.

In gameplaying a *trap state* is something to be avoided since it can result in a early loss. However, in UCT-PTA the problem manifest in another fashion. The *trap states* in UCT-PTA are not really traps, rather they are promising states that leads to a good or even optimal trace for a PTA, which exists within an otherwise poor performing subtree. The poor performing subtree will potentially make UCT-PTA shy away from it, making it miss the promising *trap state*. The common ground is the "rarity" of the trap state, and the fact that the algorithm should detect the *trap state* and take it into account when making move *m*, in Ramanujan et al. to avoid it, but in UCT-PTA to exploit it.
Ramanujan et al. continues to show, that UCT has a very hard time finding traps deeper than level-3 search traps, which is a negative property that UCT-PTA unfortunately inherits. This is a general problem for UCT-PTA and may possibly hinder the discovery of optimal traces. However, it is an even larger problem for the Stochastic Delay UCT-PTA.

When delay actions are structured in a stochastic fashion like the one described for Stochastic Delay UCT-PTA, an uneven distribution of considered delay and action transitions is introduced. If a state has some action transitions that are only valid after a certain amount of time has passed, then those action transitions can have a decreasingly low probability of being explored during the *Tree policy* and *Simulation* steps. This occurs because at each node, child nodes are randomly chosen during the *Simulation* step, so to choose a path that leads to a delay of for instance ten clock rates of a state is very unlikely as every new node that represents a single clock increase has siblings being action nodes. A visualisation of this problem is illustrated in Figure 5.3.

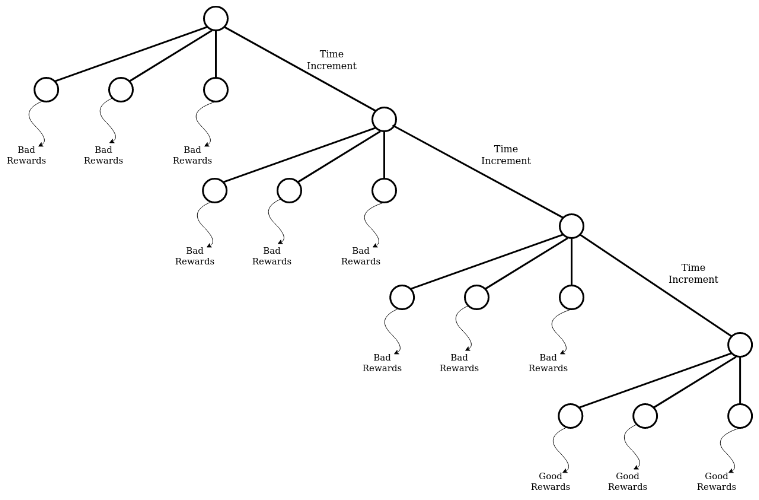


Figure 5.3: An illustration of the exponentially decreasing probability problem during the *Default policy* in the version where one-delay transition is alongside action ones. In this instance, the possibility of reaching the node that leads to good rewards is 0.25 × 0.25 × 0.25 = 0.0156 = 1.56%

Ultimately, this is exactly an introduction of additional *trap states*, since the promising state will be a part of a sub-tree that may overall consist of worse choices. This is especially true for problems where incremented time results in incremented cost. In such problems, the promising *trap state* is almost ensured to exists within a poor performing sub-tree, since the sibling-actions will have worse cost in the sub-tree compared to the sibling-actions of the first delay action not within the sub-tree.
This distribution of delays might in some instances be a good choice for the underlying domain. However, it is more likely to cripple the effectiveness of the *Tree policy* and the *Default policy* by greatly increasing the likelihood, that the promising *trap states* will not be encountered simply due to chance. Note that Stochastic Delay UCT-PTA might not be the only reason for *trap states* to occur, but rather it unnecessarily increases the total amount of such states.

5.3 UCT-PTA Variations

Besides our first implementation, Stochastic Delay UCT-PTA, we present two more variations in this section which we call Delay Exploratory UCT-PTA and Non-Lazy UCT-PTA. The primary concern of both variations is to experiment with different ways of handling time. As we saw in the subsection 5.2.2, the approach in handling time can have significant consequences for the final algorithm. Because of this we decided to focus on this aspect of the algorithm, since it is a more fundamental part of the application of UCT to PTAs, contrary to other promising enhancements which are more auxiliary in nature.

5.3.1 Delay Exploratory UCT-PTA

In the Stochastic Delay UCT-PTA we had an uneven distribution of delays, which in the worst case scenario, lead to an increased amount of promising paths, so-called *trap states*, being inaccessible during the tree search, simply due to low chance of encountering them. This raises the question of how to change the handling of time, which lead to the implementation of the variation we named Delay Exploratory UCT-PTA.

In simplicity, Delay Exploratory UCT-PTA aims to handle all possible delays of a single state, as being equally likely to be chosen during the *Default policy* and being on the same. This is achieved by distinguishing between child states produced by delay transitions $s \xrightarrow{d} s'$ and child states produced by action transitions $s \xrightarrow{a} s'$ and handling them separately both in the *Tree policy* and *Default policy*. In the tree structure this results in nodes having either child nodes produced only by delay transitions or child nodes produced only by action transitions. Furthermore the nodes are arranged in such way, that the tree will alternate between the two node versions. In the nodes with only delay transitions, all possible delays for that state are able to be chosen as the successor node, likewise, nodes with only action transitions has all possible actions from that state to choose from. This means we need functionality to distinguish between node types. We make an extension to Defitinion 3 and 4:

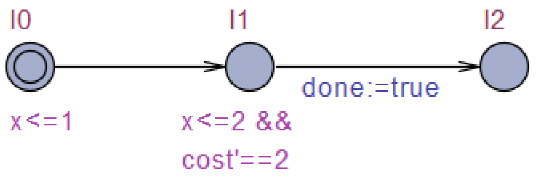
Definition 7 ***Delay Exploratory Tree Structure Extension***
Recall the definition of N from Definition 3. We add a set of boolean flags T to N such that $t : N \to \{0,1\}$, $t \in T$ where $n \in N$ and $(t(n) \implies Y(s) = \{s' \,|\, s \to s', \to : S \times Act \times S\}) \wedge (\neg t(n) \implies Y(s) = \{s' \,|\, s \to s', \to : S \times d \times S\})$.

However, this requires additional information/functionality from the PTA environment. Namely we want to split $GetChildStates$ from Definition 5 into two variations, one for action transitions and one for delay transitions. This extended definition is shown in Definition 8.

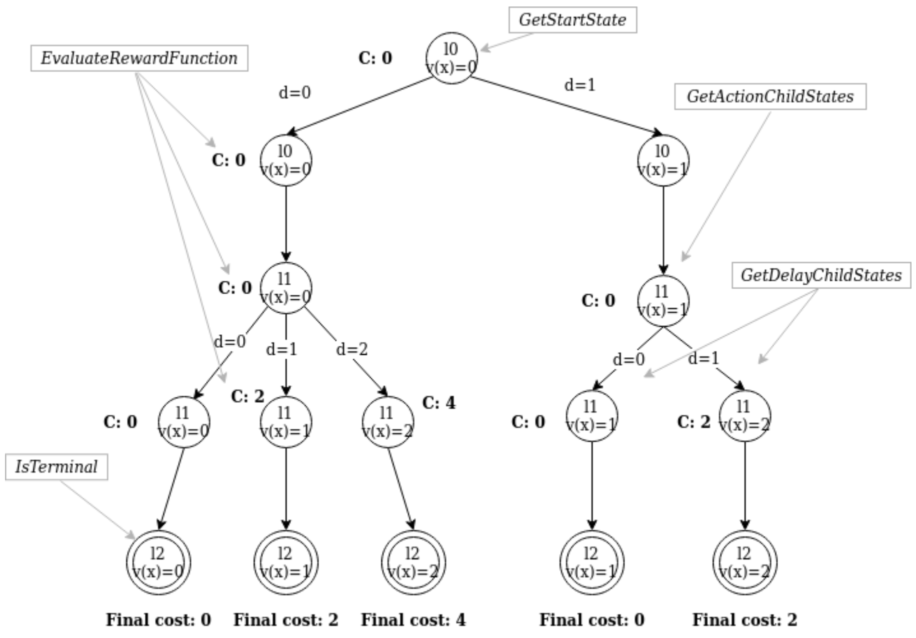
Definition 8 ***Delay Exploratory Environment Interface Extensions***

- *$GetDelayChildStates$ takes a state s and returns all the possible successor states that are produced by delay transitions.*
- *$GetActionChildStates$ takes a state s and returns all the possible successor states that are produced by action transitions.*

With this configuration, choosing a successor state uniformly at random gives an equal opportunity for all successor states of a single state to be chosen. The resulting tree structure is illustrated in Figure 5.4.



(a) A PTA toy example



(b) The tree structure for the Delay Exploratory UCT-PTA.

Figure 5.4: An example of a search tree structure for Delay Exploratory UCT-PTA (5.4b), on the same PTA toy example as in Figure 5.1 (5.4a)

Algorithm 4: Delay Exploratory UCT-PTA: The parts of UCT-PTA that has been changed for Delay Exploratory UCT-PTA.

```
1  Function Delay Exploratory Expand($n$)
2      if $t(n)$ then
3          choose $s'$ uniformly at random from $s(Y_U(n))$
4      else
5          if $s' = s \xrightarrow{Max(d)} s' \in Y_U(n)$ then
6              $s' = s \xrightarrow{Max(d)} s'$
7          else if $s' = \xrightarrow{Min(d)} s' \in Y_U(n)$ then
8              $s' = s \xrightarrow{Min(d)} s'$
9          else
10             choose $s'$ uniformly at random from $Y_U(s)$
11     end
12     create a new node $n' = ()$ such that $s(n') = s'$, $i(n') = 0$, $q(n') = 0$, $p(n') = n$,
         $Y_E(n') = \emptyset$, $Y_U(n') = Y(n')$ and $t(n') = \neg t(n)$
13     return $n'$

14 Function Delay Exploratory Tree Policy($n, ExplorePercentage$)
15     while $n$ is non-terminal do
16         if $size(Y_E(n)) < Size(Y(n)) \times ExplorePercentage$ then
17             return Expand($n$)
18         else
19             $n$ = BestChild($n$, $C_p$)
20         end
21     end
22     return $n$
```

Unfortunately a new problem arises with this way of handling time. In PTA's a state can often have a large number of delays available to choose from and in fact, may theoretically be infinite. In practice though it rarely is. However, it is still possible to encounter large enough numbers, that the branching factor makes the state space explode. It then becomes infeasible to try all possible delays of each state, as the original UCT algorithm demands.

We addressed this issue by limiting the exploration to a preset percentage of the possible delay actions to explore. Furthermore, we ensured to always explore the delay transition with the smallest possible delay and the one with the biggest delay first, and then randomly exploring the delays in between. This way, the state space explosion is limited, however, it comes at the cost of a final trace with possibly worse cost. The limitation on the exploration of in-between delays has been implemented as a percentage parameter that can be tuned, as different variations of models could work well with different exploration percentages. In the models we have used for the experiments, discussed in Chapter 6, 30% exploration limit seemed to be a good choice, after examining some of our prelim-

inary results as in the Appendix A.
Different variations of random distributions were tested for the random exploration namely:

- *Uniformly distributed exploration* - uniform random was a default choice, since it handles all the delays equally, making it a good choice for general PTA exploration.
- *Exponentially distributed exploration* - The smaller delays were given the higher probabilities with exponential reduction towards the higher delays.

The different variations may suit to different problems, and exponential distribution was tested as a distribution to fit the Job shop scheduling more closely, where shorter delays generally lead to better results.
Another approach that could be promising for future research, is proposed by Carpentier et al. in [10]. The problem of a single node having to choose the best node between a computationally infeasible amount of possible child nodes, can be rephrased into the infinitely many armed bandit problem. Carpentier et al. proposes a method that minimize the simple regret of an infinitely many armed bandit. Their proposal requires knowledge of the underlying $\beta - distribution$ for the domain, however, they also propose a bootstrapping technique to find the $\beta - distribution$ from domains where it is not known. This technique is a promising solution to the problem. Firstly, it eliminates the need for experimenting with different percentages and/or random distribution parameters for different PTA models, by automatically fitting to the PTA at hand. Secondly, the bootstrapping phase requires some sampling from the domain, which we already have a bootstrapping phase that does, albeit for the purpose of determining the reward range. It is not far-fetched to think that these bootstrapping techniques easily could be merged together.

5.3.2 Non-Lazy UCT-PTA

By doing preliminary experiments and analyzing the results with the Delay Exploratory UCT-PTA, we had noticed that both scores and the performance of solving the Job shop scheduling problems were not as good as ones obtained from the Stochastic Delay UCT-PTA. We assume the reason for that could be the exploration of insignificant delays that are in between the given lower and upper delay bounds which lead to larger state-space exploration and sometimes could end up with worse scores. Therefore, we try modifying our algorithm in such a way that it behaves as a *Non-lazy Scheduler* [1].
A non-lazy scheduler, as opposed to a lazy one, ensures that if the relevant resource is available, it will be either immediately used by the job i to start a task j or the system will be delayed until a certain point in time where some other job will start using that some resources. That is, a process will always take over an available resource unless the other process can also benefit from it, then the wait i.e delay first process taking a resource is possible. Concretely, if we look at the schedules from Figure 2.4 in the Chapter 2: at sixth second, when Jan stops reading the *Sports* section there will be two possible delays:

either it will be zero, which will be in the case of the first schedule where Jan immediately starts reading the *Economy* section, or it will be delayed by one to a state where Wang starts reading the *Sport* section. In case of this example, the delaying by one will give the optimal schedule therefore result with better score.

For the purpose of defining a lazy schedule we first define, as in [2], a used resource by a job i doing a task j with $\mu^i(j) = m$. Furthermore, we define a Task progress function with $\beta(i,t) = m$ stating which machine is used by a job j at a given time t.
Now we give the definition of Lazy schedule as in [2].

Definition 9 ***Lazy Schedules***
Let S be a schedule, let i be a job and j a task with $\mu^i(j) = m$ that starts at time t. We say that S exhibits laziness at (i,j) if there is a time $r < t$ such that for every $t' \in [r,t)$, $\beta(i,t') = \perp$ and for every $i' \neq i$, $\beta(i',t') \neq m$. A schedule is non-lazy if it exhibits no laziness.

It has been proven by Abdedda et al. [2] that non-lazy schedulers give the optimal schedule for Job shop and Task graph scheduling problems. To implement this the environment interface of Delay Exploratory UCT-PTA has to be altered slightly, this is shown in Definition 10. Note that $GetDelayChildStates$ now only returns two child states, and we know one of these is the optimal child. This drastically reduce the state space representation, and effectively solves our state-space explosion problem we had in Delay Exploratory UCT-PTA.
Besides the changes to the *environment interface*, the algorithm is generally equivalent to that of Delay Exploratory UCT-PTA, except that the tree-policy is reset to that of standard UCT, now that we don't need to only explore a certain percentage of the children.

Definition 10 ***Non-Lazy UCT-PTA Environment Interface Extension***

- *$GetDelayChildStates$ takes a state s and returns the 0 delayed successor state, which is equal to s, and the non-lazy delayed successor state.*

5.4 UCT-PTA Extensions

5.4.1 Tree Pruning with Steps

Another way to extend our UCT-PTA algorithm that can improve the efficiency of the search is by descending through the tree throughout the search. After a certain number of iterations of the main search loop, see algorithm 1, we "make a step", that is, we call the *BestChild* function from algorithm 2 in line 15, on the root node and set the returned best child node as the new root. We proceed moving down the tree until the terminal node is found or the time limit has exceeded.
This is actually the default behaviour of MCTS/UCT, since the algorithms mainly have been used for game playing, in which a player only has a set amount of time/resource

before having to make a move. However, since we focus on exploration of PTAs in a non-live environment, this is not a necessity for the UCT-PTA algorithm, which is why this is introduced as an extension.
With this approach the tree is continually pruned thereby reducing the *Tree policy* exploration in the early stages where it is less necessary, and force it to do exploration further from the original root node, where it will be more beneficial. This comes at the cost of a decreased correctness of the final tree, since the step may possibly be suboptimal.
This gives a new tuneable parameter the *step size*, which denotes the number of iterations to be conducted before the step is made and a new root node is assigned.
In the scope of this project we variate and experiment between different fixed *step sizes*. However, another promising approach, would be to make the step as soon as it becomes clear, that no other child node will become more visited, eg. enough information has been gained to ensure the best move from that node. This way of pruning in the UCT Algorithm is called *Absolute Pruning* and was proposed by Huang et al. in [12]. They show that *Absolute Pruning* preserve the correctness of UCB, that is, preserve the perfect balance of exploration and exploitation as to minimize the cumulative regret, and they also found it to increase the wining rate of *Go* [12]. Preserving the correctness of UCB, means that no suboptimal moves will be made, also no parameters would need to be tuned, both making this version purely beneficial. Lastly, stepping could naturally be modified to be used for realtime applications of PTAs, just like the original MCTS/UCT algorithm intend to.
Implementing *Absolute Pruning*, was not within the scope of this project, however, we believe it would be a great addition in future research.

5.4.2 Partial Order Reduction in UCT-PTA

Another approach with which we try to improve UCT-PTA by reducing the state-space is by introducing the Partial order reduction [8]. As a part of Partial order reduction we finish any process whose task is about to finish within zero clock units, before any other process can start. If there are multiple tasks to finish, the order in which it happens is predefined and it does not matter.
This does not prevent the finding of the best trace in terms of cost, but should reduce the complexity for the UCT-PTA algorithm, by not making it accidently assign jobs to other resources, than the actual optimal one.

6 Experiments

In this chapter we present the results of the experiments on UCT-PTA and compare it to already established methods. Different variations and extensions of UCT-PTA were tested for performance on different Job shop problems, previously described in Chapter 2. Each version of UCT-PTA was ran thousand times on several Job shop models of various complexities, differing in the number of jobs being scheduled. We compare the different UCT-PTA variations against two other algorithms, namely Best-RDFS and Uppaal Cora [6].

- Best-RDFS - a Random Depth First Search that always explores the best state as the next state, in terms of cost. If several states ties in the best cost, a state is chosen at random. It reports the cost of the first trace that ends in a terminal node it encounters.
- Cora - an algorithm that finds and proves the trace with lowest possible cost of a given PTA model. This is a slightly different goal than UCT-PTA, but still interesting to compare against.

The results reported consists of the best and worst cost found, the mean cost found and the standard deviation of the costs found. These numbers should give an insight on the consistency of the performance of the different variations. Alongside the cost, the computational performance of the algorithms are reported in the form of number of states explored and runtime. The implementation of UCT-PTA have not focused on code optimization, and as such the runtime can possibly be improved by a better implementation. Therefore, we suggest to ignore the runtime when comparing different variations of UCT-PTA. However, since the Best-RDFS and Cora handle and count states differently and also have slightly different goals than UCT-PTA, we have kept the runtimes in the result tables, to get a better overall picture of the comparison between these and UCT-PTA.

6.1 Experiments results and discussion

In this section we present and discuss the experiments for the UCT-PTA variations of differently handled delays: Stochastic Delay and Delay Exploratory UCT-PTA alongside with Non-Lazy UCT-PTA. In addition, as previously mentioned, we compare them to the Best-RDFS and Cora.

6.1.1 Job Shop Models Comparison

Our initial experiments were run on the Newspaper instance of the Jobshop problem, described in the section 2.1. Different complexities in terms of alternating numbers of jobs were tested for the algorithms.

Table 6.1: Results of 1000 runs of all UCT-PTA variations on Job shop problems of different complexities in terms of number of jobs. Grey colored rows indicate the results with the best minimum cost found. Cases where in more than 20% of runs the result was not found within the given runtime are noted with * .

Stohastic Delay UCT-PTA						
	Cost				Runtime	States
n-jobs	Min	Max	Mean	Stdev	(seconds)	Explored
2	12	12	12	0	0	1,722
4	65	69	65.06	0.5	3.09	37,868
6	153	157	153.23	0.94	3.32	26,860
8	474	573	476.04	13.97	24.49	159,158
10	954	1,178	1,033.02	21.61	117.71	559,153
12	1,212	1,573	1,466.14	90.67	243.36	961,882

Delay Exploratory UCT-PTA						
	Cost				Runtime	States
n-jobs	Min	Max	Mean	Stdev	(seconds)	Explored
2	12	12	12	0	0	2,014
4	65	114	68.97	2.50	3.27	24,496
6	175	209	192.14	8.05	28.49	120,962
8	470	673	506.24	32.74	121.55	299,728
10	1,003	1,301	1,154.50	79.06	605.77	1,231,493
12	1,189	1,991	1,458.93	159.95	1,646.01	2,545,025

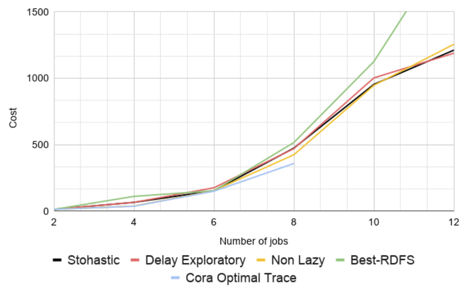
Non-Lazy UCT-PTA						
	Cost				Runtime	States
n-jobs	Min	Max	Mean	Stdev	(seconds)	Explored
2	12	12	12	0	0	1,290
4	37	79	43.55	11.91	11.82	208,414
6	153	178	154.26	3.69	6.94	72,775
8*	424	609	544.31	27.50	138.34	985,882
10	947	1,182	1,036.75	58.59	219.19	1,505,776
12	1,256	1,660	1,416.35	54.20	689.67	4,540,552

Best-RDFS						
	Cost				Runtime	States
n-jobs	Min	Max	Mean	Stdev	(seconds)	Explored
2	12	12	12	0	0	29
4	110	162	135.63	25.38	0	74
6	153	576	402.87	93.09	0	128
8	516	1,280	954.87	128.67	0.01	191
10	1,126	2,481	1,837.27	200.46	0.01	254
12	2,062	4,267	3,001.10	336.70	0.02	328

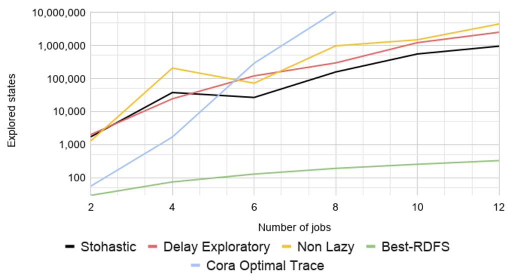
Cora						
	Cost				Runtime	States
n-jobs	Min	Max	Mean	Stdev	(seconds)	Explored
2	12	-	-	-	0.03	55
4	36	-	-	-	0.07	1,702
6	149	-	-	-	92	292,273
8	358	-	-	-	10,768	10,878,116
10	failed	-	-	-	failed	failed
12	failed	-	-	-	failed	failed

Looking at Best-RDFS and Cora relative to all versions of UCT-PTA, a general picture forms. Best-RDFS is unmatched in terms of computation time, but falls significantly behind in cost compared to the UCT-PTA variations, especially when the models start to increase in size. On the other end, Cora naturally has the best performance in terms of cost, since it finds and proves the best cost. However, the runtime and amount of states explored explodes very quickly as the complexity of the models increase. In fact it was unable to run for ten and twelve people Job shop models, terminating due to a lack of memory caused by the fact that Cora is a 32-bit program.
Looking at how the variations of UCT-PTA perform among themselves, we observe that Non-Lazy UCT-PTA and Stochastic Delay UCT-PTA seem to be on par in terms of the cost, with Non-Lazy UCT-PTA being slightly better in terms of the best costs found, but also slightly worse in terms of the worst cost found. However, looking at the states explored

Non-Lazy UCT-PTA is significantly worse, and in some cases even worse than Delay Exploratory UCT-PTA, which is generally the poorest performing variation of UCT-PTA. This runs counter to our theoretical intuition of how the different variations should be performing, were Non-Lazy should be the best performer followed by Delay Exploratory UCT-PTA and lastly Stochastic Delay UCT-PTA.



(a) The best cost found by different algorithms



(b) An average state exploration by different algorithms

Figure 6.1: Visualization of the results presented in Table 6.1

We theorize that the reason for the unreasonable effectiveness of Stochastic Delay UCT-PTA is because of the specific instances of models the algorithms were tested on. Stochastic Delay UCT-PTA and Delay Exploratory UCT-PTA should be disadvantaged compared to Non-Lazy UCT-PTA in problems with states that have a wide range of delay transitions available. However, the specific models tested here, while being normal Job shop scheduling problems, actually had very short running jobs, typically low one-digit numbers, making it a likely explanation for the skewed performance compared to the expected performance.

Increased Time Job Shop Models

To test our hypothesis about the expected vs. actual performance, we modify all of the job shop models to have longer task times. Concretely, all the task durations are multiplied by five compared to their original counterpart. This results in states generally having a broader range of possible delays. To reiterate our hypothesis, as different versions of UCT-PTA have different approaches in handling/exploring delays, we assume this modification will have an impact on their relative performances. In Table 6.2 we present the results of the experiments on these time increased models.

Table 6.2: Comparison of the results after multiplying Job shop problem model's task durations by five. Cases where the result was not found in more than 40% of runs within a given runtime are denoted with **. Cases where the results was not found in 20% of runs were noted with *.

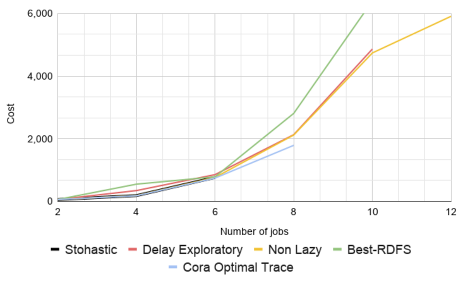
Stohastic Delay UCT-PTA						
	Cost				Runtime	States
n-jobs	Min	Max	Mean	Stdev	(seconds)	Explored
2	60	66	60.05	0.42	3.68	28,297
4**	185	185	185	0	320.29	1,749,626
6**	765	822	781.40	11.54	420.26	2,045,388
8	-	-	-	-	1200	5,088,224
10	-	-	-	-	2400	9,132,099
12	-	-	-	-	2400	8,262,686

Delay Exploratory UCT-PTA						
	Cost				Runtime	States
n-jobs	Min	Max	Mean	Stdev	(seconds)	Explored
2	60	63	60.02	0.23	1.13	21,736
4	345	561	349.16	10.44	72.89	495,461
6	855	1,251	999.52	77.30	265.60	1,220,444
8*	2,130	3,537	2,802.18	174.51	868.94	2,848,072
10*	4,864	6,686	6,064.48	295.70	2,172.37	3,886,699
12	-	-	-	-	2400	-

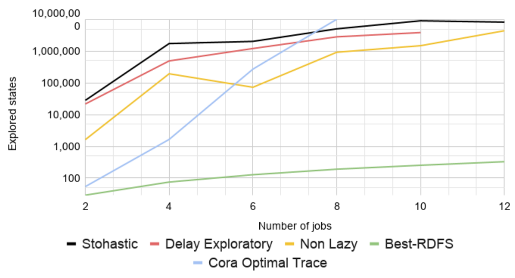
Non-Lazy UCT-PTA						
	Cost				Runtime	States
n-jobs	Min	Max	Mean	Stdev	(seconds)	Explored
2	60	60	60	0	0	1,633
4	185	355	280.61	66.92	11.14	195,008
6	765	806	773.70	12.24	6.97	73,154
8**	2,120	3,020	2716.46	145.05	133.43	930,699
10	4,740	5,965	5162.46	270.44	219.19	1,496,161
12	5,910	8,378	7,074.72	266.27	677.33	4,430,536

Best-RDFS						
	Cost				Runtime	States
n-jobs	Min	Max	Mean	Stdev	(seconds)	Explored
2	60	60	60	0	0	29
4	550	810	674.58	126.84	0	75
6	785	2,880	1,996.50	479.81	0.01	128
8	2,810	6,380	4,737.91	614.50	0.01	191
10	6,400	11,785	9,228.31	953.01	0.01	254
12	9,680	19,475	15,094.53	1,624.46	0.02	329

Cora						
	Cost				Runtime	States
n-jobs	Min	Max	Mean	Stdev	(seconds)	Explored
2	60	-	-	-	0	54
4	180	-	-	-	0.06	1,658
6	745	-	-	-	113	273,741
8	1790	-	-	-	12,801	10,197,507
10	failed	-	-	-	failed	failed
12	failed	-	-	-	failed	failed



(a) The best cost found by different algorithms



(b) Average state exploration by different algorithms

Figure 6.2: Visualization of the results presented in Table 6.2

Looking at Table 6.2, it looks like our hypothesis is correct. Non-Lazy UCT-PTA is now clearly the best performing algorithm in terms of both cost and states explored among the UCT-PTA variations. Furthermore, as expected Delay Exploratory UCT-PTA outperforms Stochastic Delay UCT-PTA, which generally really struggle with the models, not even being able to find a single result for the eight people job shop problem and larger in

the computation time given. This likely stems from the fact, that Stochastic Delay UCT-PTA introduces more *trap states* and generally produces a large tree representation.
Delay Exploratory UCT-PTA is generally better than Stochastic Delay UCT-PTA, but still suffers from the larger ranges of delays. In fact, we initially started with models with ten times increased time for tasks but unfortunately, Stochastic Delay UCT-PTA and Delay Exploratory UCT-PTA were unable to find results for any of the models, so we settled on multiplying by five instead, since we could then get a better comparison between the two.
Best-RDFS and Cora performs somewhat the same relative to the UCT-PTA algorithms, as they did in the original models. Cora especially hardly changes, having almost equivalent runtimes and amount of states explored for the increased time models as it did for the original models.

6.1.2 UCT-PTA extensions

In this section, we choose the best performing variation of the UCT-PTA algorithm, which has shown to be the Non-lazy UCT-PTA, and apply the Partial Order reduction and Stepping extensions to it and show how those modifications affect its performance.

Partial Order Reduction

Partial Order Reduction was applied to Non-Lazy UCT-PTA and compared against Non-Lazy UCT-PTA without Partial Order Reduction. The results can be seen here in Table 6.3:

Table 6.3: The results of Non-Lazy UCT-PTA before and after introducing Partial Order Reduction Extension. Cases where the result was not found in more than 40% of runs within a given runtime are denoted with **. Cases where the results was not found in 20% of runs were noted with *. The rows colored in gray are ones with the best cost found.

Non-Lazy UCT-PTA						
	Cost				Runtime	States
n-jobs	**Min**	**Max**	**Mean**	**Stdev**	**(seconds)**	**Explored**
2	12	12	12	0	0	1,290
4	37	79	43.55	11.91	11.82	208,414
6	153	178	154.26	3.69	6.94	72,775
8**	424	626	540.30	27.35	86.88	549,821
10*	947	1182	1036.75	58.59	219.19	1,505,776
12	1256	1660	1416.35	54.20	689.67	4,540,552

Non-Lazy with POR						
	Cost				Runtime	States
n-jobs	**Min**	**Max**	**Mean**	**Stdev**	**(seconds)**	**Explored**
2	12	12	12	0	0	849
4	37	75	55.55	13.4	1.57	35,412
6	153	174	153.67	3.62	4.12	43,749
8	424	645	485.91	32.70	47.42	346,136
10	849	1,150	1,000.99	60.23	154.42	917,189
12	1,150	1,744	1,462.81	146.98	461.11	2,080,246

The Partial Order Reduction is a straightforward improvement. The states explored is generally reduced by 30-50%, while the cost found stays almost the same. The standard deviation increases a bit for some of the larger models, but it is likely due to the fact that it finds better minimum cost, which increases the range of results.
The four-people problem has been causing some problems for Non-Lazy UCT-PTA and Stochastic Delay UCT-PTA evidenced by the fact that the six-people problem has better performance in terms of states explored for both models in Table 6.1, but also in the

increased time models Table 6.2. However, it seems like the Partial Order Reduction reduced some of the complexity that caused the problems. Non-Lazy UCT-PTA with Partial Order Reduction performs significantly better compared to without on the four people problem. With that being said, it could possibly be beneficial to analyze the four-people problem model more closely, to figure exactly what happens, and why it does not seem to be a problem for Delay Exploratory UCT-PTA.

Stepping

Here we present the results for Non-Lazy UCT-PTA with the stepping implementation. Experiments for different stepping sizes were tested, and shown here side-by-side in Table 6.4. All of the stepping implementations also used Partial Order Reduction, and as such, should be compared against Non-Lazy UCT-PTA with Partial Order Reduction in Table 6.3.

Table 6.4: Results of running Non-Lazy UCT-PTA with different step sizes.

Non-Lazy UCT-PTA with Stepping

Step size: 10						
	Cost				Runtime	States
n-jobs	Min	Max	Mean	Stdev	(seconds)	Explored
2	12	12	12	0	0	165
4	55	124	82.12	18.75	0.03	499
6	157	418	202.63	39.76	0.08	824
8	410	813	599.63	75.28	0.16	1,166
10	890	1,775	1,268.05	151.00	0.29	1,483
12	1,393	2,923	2,040.61	249.89	0.46	1,798

Step size: 50						
	Cost				Runtime	States
n-jobs	Min	Max	Mean	Stdev	(seconds)	Explored
2	12	12	12	0	0.01	418
4	53	79	72.08	4.84	0.09	1,889
6	153	227	166.91	17.33	0.31	3,557
8	370	711	542.09	65.22	0.69	5,378
10	746	1,429	1,048.17	108.78	1.28	6,968
12	1,219	2,211	1,601.43	176.01	2.06	8,611

Step size: 100						
	Cost				Runtime	States
n-jobs	Min	Max	Mean	Stdev	(seconds)	Explored
2	12	12	12	0.02	0	513
4	43	79	69.37	8.0	0.19	3,358
6	153	232	162.91	16.30	0.56	6,644
8	390	677	541.59	64.08	1.30	10,376
10	746	1,323	1004.54	100.28	2.44	13,514
12	1,131	2,023	1507.45	168.02	4	16,863

Step size: 500						
	Cost				Runtime	States
n-jobs	Min	Max	Mean	Stdev	(seconds)	Explored
2	12	12	12	0	0.02	796
4	37	79	45.24	6.69	0.64	10,844
6	153	183	154.46	5.29	2.09	24,862
8	402	741	548.66	86.53	5.74	46,013
10	746	1152	907.34	94.63	11.29	63,033
12	1,113	1,784	1,452.73	166.68	18.96	79,440

Step size: 1,000						
	Cost				Runtime	States
n-jobs	Min	Max	Mean	Stdev	(seconds)	Explored
2	12	12	12	0.03	0	850
4	43	75	58.76	10.60	0.98	17,792
6	153	174	153.88	4.17	2.97	32,573
8	402	691	533.03	62.68	10.45	81,515
10	746	1,126	909.64	78.94	21.73	122,096
12	1,122	1,773	1,324.20	147.15	4 37.07	155,782

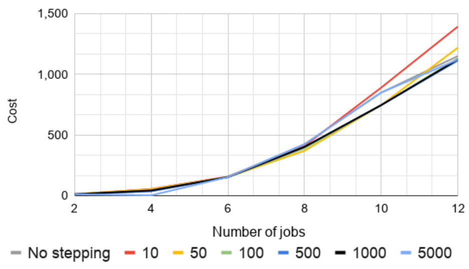
Step size: 5,000						
	Cost				Runtime	States
n-jobs	Min	Max	Mean	Stdev	(seconds)	Explored
2	12	12	12	0	0.3	848
4	37	75	64.18	5.76	1.6	29,985
6	153	174	153.55	3.31	4.36	43,434
8	424	649	502	46.87	27.49	203,781
10	849	1,274	1008.51	62.28	85.34	538,864
12	1,121	1,697	1,233.47	135.27	162.25	664,618

Looking at Table 6.4 it is easy to see, that the stepping implementation for Non-Lazy UCT-PTA provides a significant boost in performance.

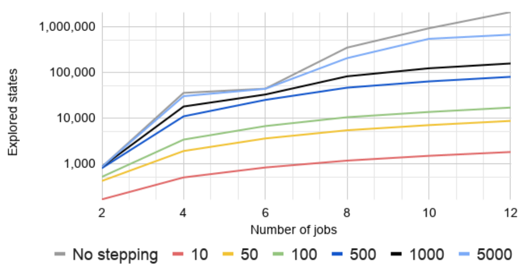
Intuitively one could tend to think, that with stepping enabled, the cost would suffer over an improvement in speed, due to the fact that making a step may hinder the algorithm from reaching promising branches. However, as is evident, this is not the case. In fact several of the experiments show even better cost than Non-Lazy UCT-PTA without stepping. The reason is to do with the fact that [12] presented a pruning technique *absolute pruning*, that was shown to preserve the correctness of UCB. This was also discussed in subsection 5.4.1. Stepping does not preserve correctness of UCB in itself, however, given a large enough stepping size, it will very likely make the same move, as a UCB correct implementation would. Furthermore, the computational resources that would have otherwise been wasted on exploring suboptimal branches, if the move had not been made, can now be allocated to areas of the search-tree where the information gain, will have a much higher pay-off. So in short, the stepping implementation forces the algorithm to avoid exploring in areas where it is unnecessary to do so.

For our experiments, the overall best step size is 50. However, it is slightly ambiguous as step size 500 outperforms 50 in some models. From this we can quickly conclude that a fixed step size is not the best choice as different models require different step sizes to perform best. In fact, it is possible, that even within a single model, different states may require different amount of visits by the *Tree policy* before *BestChild* will be able to choose the best move. As such, *Absolute pruning* seems to be the obvious choice to benefit from this improvement.

Lastly, something very noteworthy is the significant reduction in states explored, which is even easier to see in Figure 6.3b, where the grey line represents the Non-Lazy UCT-PTA without stepping. The best performing step-sizes have reduced the states explored by 1-2 orders of magnitude, compared to Non-Lazy UCT-PTA without stepping, with the exception of the two people problem.



(a) The best found cost by Non-Lazy UCT-PTA with different step sizes noted by different colors.



(b) An average state exploration by Non-Lazy UCT-PTA with different step sizes noted by different colors.

Figure 6.3: Visualization of the results presented in Table 6.4

6.1.3 Final Comparison charts

For the final comparison charts, the best version of Non-Lazy UCT-PTA is compared against Best-RDFS and Cora on the Job shop scheduling problem, as well as on some additional Task Graph Scheduling models.

Job shop scheduling

In Table 6.5 the Job shop scheduling are re-shown, this time for the best version of UCT-PTA, namely Non-Lazy UCT-PTA with stepping and Partial Order Reduction. It is compared against Cora and Best-RDFS. All of the tables have already been shown in previous sections, but are here gathered for a final assessment.

Table 6.5: Job shop scheduling results for the Best-RDFS, Cora and the best performing version of UCT-PTA. All of the algorithms ran 1000 times on each model.

Job shop scheduling

Best-RDFS

	Cost				**Runtime**	**States**
n-jobs	**Min**	**Max**	**Mean**	**Stdev**	**(seconds)**	**Explored**
2	12	12	12	0	0	29
4	110	162	135.63	25.38	0	74
6	153	576	402.87	93.09	0	128
8	516	1,280	954.87	128.67	0.01	191
10	1,126	2,481	1,837.27	200.46	0.01	254
12	2,062	4,267	3,001.10	336.70	0.02	328

Cora

	Cost				**Runtime**	**States**
n-jobs	**Min**	**Max**	**Mean**	**Stdev**	**(seconds)**	**Explored**
2	12	-	-	-	0.03	55
4	36	-	-	-	0.07	1,702
6	149	-	-	-	92	292,273
8	358	-	-	-	10,768	10,878,116
10	failed	-	-	-	failed	failed
12	failed	-	-	-	failed	failed

Non-Lazy UCT-PTA (step size: 50)

	Cost				**Runtime**	**States**
n-jobs	**Min**	**Max**	**Mean**	**Stdev**	**(seconds)**	**Explored**
2	12	12	12	0	0.01	418
4	53	79	72.08	4.84	0.09	1,889
6	153	227	166.91	17.33	0.31	3,557
8	370	711	542.09	65.22	0.69	5,378
10	746	1,429	1,048.17	108.78	1.28	6,968
12	1,219	2,211	1,601.43	176.01	2.06	8,611

The final best version of UCT-PTA compares well against Best-RDFS and Cora. We observe a runtime/state space exploration that is nearing Best-RDFS albeit still an order of magnitude from it. At the same time, the cost found is significantly better than Best-RDFS, especially as the models increase in size, making the trade-off worthwhile.
Compared to Cora, it is evident that Non-Lazy UCT-PTA has a much better scaling in terms of runtime/state space exploration than Cora. Cora failed at ten and twelve people, due to memory constraints of the 32-bit program, which is not entirely of fault to the algorithm, however, already at the eight people model the runtime/state space exploration explodes. This naturally ensures the optimal cost but would quickly become infeasible to compute as the complexity further increases, even if the program was to be

re-implemented to 64-bit.
Cost wise, Non-Lazy UCT-PTA leaves a bit to be desired when comparing against Cora. The best costs found are within 10% difference to the optimal for the models we know the optimal trace for (for four people, step size 500 found a minimum cost of 37), but the algorithm is volatile and the mean and especially worst costs, are often far from the minimum. The exact reasons for this, requires a thorough investigation of the models, along with the decision making of the algorithm to properly assess. However, a likely hindrance to the algorithm are the promising states that lies in otherwise unpromising sub-trees, the so-called *trap states*, which we already concluded was a big obstacle for the Stochastic Delay UCT-PTA variation in subsubsection 6.1.1. Research into how to better detect/reduce these states will likely be very beneficial to the algorithm.
Likewise, we saw a notable performance improvement when reducing the state space representation between Delay Exploratory UCT-PTA and Non-Lazy UCT-PTA. Further research into this area may very likely prove beneficial to the algorithm.

Task-Graph Models

In this section we test the Non-Lazy UCT-PTA algorithm on the Task-Graph Scheduling problem [13] and compare against Best-RDFS. We obtained the task graph UPPAAL models from Ejsing et al. in [11] who in turn made their task graph uppaal models from problems obtained from a standard task graph set of Kasahara et al. [13]. The models comes in two variations: preemptive tasks and non-preemptive tasks. We had to modify the models to make them work with UCT-PTA, but we unfortunately introduced a breaking bug to the preemptive versions, that we were unable to fix. Consequently we only report results for the non-preemptive tasks. The models comes in sizes of 50-1000 tasks and 2-11, 16 machines.
We experimented on all of the non-preemptive tasks. We used Non-Lazy UCT-PTA with Partial Order Reduction and stepping, with step sizes of 50 and rollout limit 50. They ran for 10 minutes and the reported results are the first terminal node found, along with the amount of time it took to find it. The 10 minutes proved to not be enough time to find any results for models larger than 300 tasks. All of these were removed from the result table, to be able to more easily gain an overview of the results for the models, that results were found for. The results are shown in Table 6.6
Cora was unable to find results for any of the models and has been left out of the table.

Table 6.6: This table shows the performance of Non-Lazy UCT-PTA, relative to Best-RDFS, for the non-preemptive task-graph scheduling problems. The 5 **best**, 5 **mean** and 5 **worst** models, in terms of Non-Lazy UCT-PTA performance relative to Best-RDFS, are shown. **Tasks** denote number of tasks, **Machines** number of machines and **Version** denotes a unique task graph problem, although different versions may have same number of tasks and machine. The cost is presented as percentage deviation of Non-Lazy UCT-PTA relative to Best-RDFS, negative scores meaning Non-Lazy UCT-PTA performs better than Best-RDFS and positive scores being worse. **Min** is the minimum found by the algorithms and **Max** is the maximum. **Q1**, **Q2** and **Q3** are the best, middle and worst quartile respectively.

Non-Preemptive Task-Graph Scheduling Problems Results

				Non-Lazy UCT-PTA / Best-RDFS					Non-Lazy UCT-PTA		Best-RDFS	
	Tasks	Ver-sion	Mach-ines	Cost					Runtime (seconds)	States explored	Runtime (seconds)	States explored
				Min	Q1	Q2	Q3	Max				
Best	50	2	3	-6.828	-10.055	-12.801	-15.466	-18.519	6.18	10786	0.047	371
	50	2	4	-7.792	-9.746	-11.715	-12.077	-11.236	6.18	10656	0.046	376
	50	0	5	-3,508	-6.825	-10.587	-15.700	-20.27	6.74	11205	0.064	505
	50	0	4	-4.347	-6.395	-10.327	-14.330	-17.441	7.6	11275	0.08	519
	50	0	6	0	-2.655	-6.536	-11.447	-14.93	6.8	11306	0.065	494
Mean	100	1	8	-1.135	-0.653	-0.459	-2.094	-3.061	27.26	22145	0.153	1038
	300	2	8	-1.515	-0.723	-0.762	-1.311	-3.202	509	67259	1.899	2670
	100	1	10	0	-0.804	-0.497	-1.235	-1.123	27.3	22107	0.150	974
	50	0	2	0.763	-0.063	-0.201	-1.337	-3.521	6.9	11644	0.0634	492
	100	2	2	1.034	0.113	-0.795	-0.899	0	38.5	21959	0.127	651
Worst	300	0	9	1.181	1.237	1.156	1.266	1.873	536.08	67260	1.9002	2625
	300	2	10	1.593	1.819	2.124	2.736	2.713	557.3	67259	1.85	2591
	300	2	16	0.796	1.315	2.321	3.466	3.984	584.3	67228	1.85	2547
	300	2	11	0.398	1.267	2.543	3.627	4.72	572.8	67195	1.829	2567
	100	0	16	1.063	1.063	2.250	4.521	5.319	41.08	21987	0.168	904

Table 6.6, gives mixed results for Non-Lazy UCT-PTA, where a general picture appears that is not much in favor of UCT-PTA. In terms of task size, Non-Lazy UCT-PTA is by far best in the problems with fewest tasks eg. the 50 tasks problem. The fact that four out of five mean models are 100 tasks models, and four out of five worst models are 300 tasks models, may indicate that the state spaces of the larger models, simply have not been sufficiently explored in the time given to report better costs. Also, the models that had more than 300 tasks also failed, is another supporting indicator of this. To properly assess this, it would be necessary to run the models for much longer and allow them to report multiple traces, to see if UCT-PTA would find any better results.
Another interesting observation to make, is that the amount of machines increases as the performance relative to Best-RDFS worsens. A plausible explanation would be, that less amount of machines requires more long-term planning to achieve the best cost. More machines mean resources are generally more readily available, and consequently it may be less beneficial to plan ahead. This directly plays into Best-RDFS, which only looks shortly ahead. At the same time Non-Lazy UCT-PTA have an increased state space to keep track of, and may "waste" computation resources relative to Best-RDFS.
Shifting gaze to runtimes/state space exploration, makes the case worse for Non-Lazy

UCT-PTA. The states explored rises more or less linearly with the amount of tasks, however the runtime grows much faster. This may possibly be partly due to unoptimized code, but that likely does not explain the whole reason.
Generally the trade-off between Non-Lazy UCT-PTA compared to Best-RDFS, is better cost by sacrificing speed. However, while it was a promising trade-off in the Job shop scheduling problems, where the cost was significantly better and the speed only marginally slower, the Task graph scheduling problems are more mixed. Here, running Non-Lazy UCT-PTA does not necessarily guarantee a better cost than Best-RDFS and comes at a cost of greatly reduced speed.

7 Conclusion

We proposed an algorithm, UCT-PTA, that is an adaptation of the Monte Carlo Tree Search variant UCT to Priced Timed Automata. Furthermore, we proposed three variations namely, Stochastic Delay UCT-PTA, Delay Exploratory UCT-PTA and Non-Lazy UCT-PTA, that handles time in different fashion. Stochastic Delay UCT-PTA only have one-delay transitions for each state with possible delays. We found that this variation was problematic, since it introduced very low chances for long delay transitions. Delay Exploratory UCT-PTA has, for each state, the entire range of possible delay transitions available. We found this reduced the problem that Stochastic Delay UCT-PTA had. However, it introduced a new problem in the form of an exploding state-space due to the large branching factor. Finally, we introduced Non-Lazy UCT-PTA, which only had the non-lazy delay transitions available, but otherwise functioning as Delay Exploratory UCT-PTA. This greatly reduced the state-space and consequently turned out to be the best performing variation of the three.
We also proposed some additional extensions to the Non-Lazy UCT-PTA algorithm. First we introduce Partial Order Reduction, which forces tasks that finishes in zero time-increments to free up their resources before any other action is taken. We find that it improves performance. Next we introduce stepping, which was actually a part of the original UCT, but was initially discarded due to the fact, that it is mostly used for live game-playing. We introduce it as a pruning technique, and we find it further boosts performance greatly.
Finally we test the best variation of Non-Lazy UCT-PTA on the Job shop scheduling problem and the Task graph scheduling problem, and compare it against Best-RDFS and Cora. We find that on the Job shop scheduling problem, Non-Lazy UCT-PTA gives good results and especially scales greatly in performance with increasing model complexity, compared to the other two algorithms. On the task graph scheduling problem, we find that the results are more mixed. On models with smallest amount of tasks, Non-Lazy UCT-PTA has a significant advantage over Best-RDFS. However as the amount of tasks increases, Best-RDFS overtakes Non-Lazy UCT-PTA. We conclude that Non-Lazy UCT-PTA may perform better, in terms of cost, with increased time to finish searching the models, but that it is generally notable slower than Best-RDFS for a relatively small performance gain.

7.1 Future Research

In this section the techniques we believe to be most promising for future research in this area, will be quickly walked through.

- *Absolute pruning* [12]: Pruning of the search tree by stepping, proved to improve performance greatly. Absolute pruning is not directly a stepping technique, but prunes in a fashion that achieves the same, or better, results as a finely tuned step size. Concretely, absolute pruning prunes nodes, as soon as it becomes evident, that the node will never be more visited than a sibling node.

- *State equivalence check*: When building a MCTS search-tree, it is possible for the tree to contain multiple nodes in different places of the tree that represent the same state. Since states posses the Markov property, multiple states that are equivalent, would always have the same sub-tree. Identifying these equivalent states may save computation resources by not needing to explore the same sub-tree multiple times.
- *Simple regret for infinitely many-armed bandits*: For systems that can not take advantage of non-lazy schedulers, the Delay Exploratory UCT-PTA may be of use with some further development. As mentioned in subsection 5.3.1 Delay Exploratory UCT-PTA suffers from a huge state space. Carpentier and Valko [10] proposes a an algorithm for minimizing the simple regret of infinitely many-armed bandits. If implemented it could help reduce state space in Delay Exploratory UCT-PTA.
- *Trap states*: In [17] Ramanujan et al. finds that UCT has a hard time finding trap states, something that we also concluded was a problem for UCT-PTA. Finding techniques to detect and/or handle trap states, could be beneficial to UCT-PTA, but likely also to the general UCT algorithm.

Bibliography

[1] Yasmina Abdeddaïm and Oded Maler. “Job-Shop Scheduling Using Timed Automata?” In: *Computer Aided Verification.* Ed. by Gérard Berry, Hubert Comon, and Alain Finkel. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 478–492.

[2] Yasmina Abdeddaïm and Oded Maler. “Job-Shop Scheduling Using Timed Automata”. In: *Computer Aided Verification.* Ed. by Gérard Berry, Hubert Comon, and Alain Finkel. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 478–492.

[3] Rajeev Alur and David Dill. “Automata for modeling real-time systems”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 443 LNCS (1990), pp. 322–335. ISSN: 16113349.

[4] “Bandit based Monte-Carlo planning”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics).* Vol. 4212 LNAI. Springer Verlag, 2006, pp. 282–293. ISBN: 354045375X. DOI: `10.1007/11871842_29`.

[5] Gerd Behrmann, Kim G. Larsen, and Jacob I. Rasmussen. “Priced timed automata: Algorithms and applications”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 3657 LNCS (2005), pp. 162–182. ISSN: 03029743. DOI: `10.1007/11561163{\_}8`.

[6] Gerd Behrmann et al. “Guiding and Cost-Optimality in Uppaal”. English. In: *AAAI Spring Symposium Model-Based Validation of Intelligence.* null ; Conference date: 19-05-2010. 2001.

[7] Johan Bengtsson et al. “UPPAAL — a tool suite for automatic verification of real-time systems”. In: *Hybrid Systems III.* Ed. by Rajeev Alur, Thomas A. Henzinger, and Eduardo D. Sontag. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 232–243.

[8] Frederik M. Bønneland et al. “Start Pruning When Time Gets Urgent: Partial Order Reduction for Timed Systems”. In: *Computer Aided Verification.* Ed. by Hana Chockler and Georg Weissenbacher. Springer International Publishing, 2018.

[9] Cameron B Browne et al. “A Survey of Monte Carlo Tree Search Methods”. In: *Computational Intelligence and AI in Games, IEEE Transactions on* 4.1 (2012), pp. 1–43.

[10] Alexandra Carpentier and Michal Valko. *Simple regret for infinitely many armed bandits.* 2015. arXiv: `1505.04627 [cs.LG]`.

[11] Anne Ejsing et al. “Near Optimal Task Graph Scheduling with Priced Timed Automata and Priced Timed Markov Decision Processes”. In: (2020). URL: `https://arxiv.org/abs/2002.10783`.

[12] J. Huang et al. “Pruning in UCT Algorithm”. In: *2010 International Conference on Technologies and Applications of Artificial Intelligence*. 2010, pp. 177–181.

[13] H Kasahara. *Standard Task Graph Set*. 2020. URL: `http://www.kasahara.cs.waseda.ac.jp/schedule/`.

[14] Levente Kocsis, Csaba Szepesvári, and Jan Willemson. “Improved Monte-Carlo Search”. In: *White paper* 1 (2006), p. 22. DOI: `10.1007/11871842_29`. URL: `http://www.sztaki.hu/{~}szcsaba/papers/cg06-ext.pdf`.

[15] Jeppe Høiriis Mortensen and Adriana Mijacika. *Monte Carlo Tree Search for Priced Timed Automata*. 2020. URL: `https://people.cs.aau.dk/~pgj/mi903e19.pdf`.

[16] Gur Mosheiov. “Complexity analysis of job-shop scheduling with deteriorating jobs”. In: *Discrete Applied Mathematics* 117.1 (2002), pp. 195 –209. ISSN: 0166-218X.

[17] Raghuram Ramanujan, Ashish Sabharwal, and Bart Selman. “On Adversarial Search Spaces and Sampling-Based Planning”. In: *Proceedings of the Twentieth International Conference on Automated Planning and Scheduling (ICAPS 2010)* (2010). URL: `http://www.cs.cornell.edu/~raghu/Raghuram_Ramanujan_files/icaps10.pdf`.

A Appendix A

Table A.1: Preliminary results of the experiments on Delay Exploratory UCT-PTA with different percentages of uniformly explored delyas.

15% exploration				
	Cost		Runtime	States
n-jobs	**Min**	**Max**	**(seconds)**	**Explored**
2	12	12	0.1	1,321
4	65	69	2.2	16,924
6	188	205	23.8	96,479
8	490	578	109.1	273,031
10	1,048	1,252	595.6	1,219,090
12	1,438	1,642	1826	2,765,058

30% exploration				
	Cost		Runtime	States
n-jobs	**Min**	**Max**	**(seconds)**	**Explored**
2	12	12	0.1	1,889
4	69	69	3.7	28,955
6	176	205	29.4	124,872
8	490	490	113.8	278,711
10	1,048	1,252	612.7	1,246,271
12	1,255	1,759	1,584.56	2,432,163

50% exploration				
	Cost		Runtime	States
n-jobs	**Min**	**Max**	**(seconds)**	**Explored**
2	12 1759	12	0	3,131
4	69	69	4.2	30,743
6	192	205	41.5	197,455
8	490	589	160.60	420,276
10	1,186	1252	764.6	1,461,614
12	1,267	1634	1829.00	2,890,821

Table A.2: Preliminary results of the experiments on Delay Exploratory UCT-PTA with different percentages of exponentially explored delays.

15% exploration				
	Cost		Runtime	States
n-jobs	**Min**	**Max**	**(seconds)**	**Explored**
2	12	12	0	6,728
4	65	69	9	90,823
6	183	200	29	174,089
8	474	528	106	473,681
10	1,012	1,168	566	1,857,388
12	1,240	1,587	1,072	2,591,869

30% exploration				
	Cost		Runtime	States
n-jobs	**Min**	**Max**	**(seconds)**	**Explored**
2	12	12	0	13,484
4	69	69	10	112,143
6	153	188	35	224,017
8	490	651	193	937,534
10	1,048	1,168	650	2,294,120
12	1,239	1,419	1,149	2,826,117

50% exploration				
	Cost		Runtime	States
n-jobs	**Min**	**Max**	**(seconds)**	**Explored**
2	12	12	0	25,785
4	65	69	11	131,472
6	188	188	52	397,542
8	434	651	330	1,693,029
10	1019	1183	948	3,686,201
12	1239	1561	1435	3,858,951