# Faster Algorithms for Quantitative Analysis of MCs and MDPs with Small Treewidth*

Ali Asadi[1], Krishnendu Chatterjee[2], Amir Kafshdar Goharshady[2], Kiarash Mohammadi[3], and Andreas Pavlogiannis[4]

[1] Sharif University of Technology, Tehran, Iran
aasadi@ce.sharif.edu
[2] IST Austria, Klosterneuburg, Austria
{kchatterjee, goharshady}@ist.ac.at
[3] Ferdowsi University of Mashhad, Mashhad, Iran
mohammadi.kiarash@mail.um.ac.ir
[4] Aarhus University, Aarhus, Denmark
pavlogiannis@cs.au.dk

**Abstract.** Discrete-time Markov Chains (MCs) and Markov Decision Processes (MDPs) are two standard formalisms in system analysis. Their main associated *quantitative* objectives are hitting probabilities, discounted sum, and mean payoff. Although there are many techniques for computing these objectives in general MCs/MDPs, they have not been thoroughly studied in terms of parameterized algorithms, particularly when treewidth is used as the parameter. This is in sharp contrast to *qualitative* objectives for MCs, MDPs and graph games, for which treewidth-based algorithms yield significant complexity improvements.

In this work, we show that treewidth can also be used to obtain faster algorithms for the quantitative problems. For an MC with $n$ states and $m$ transitions, we show that each of the classical quantitative objectives can be computed in $O((n+m)\cdot t^2)$ time, given a tree decomposition of the MC that has width $t$. Our results also imply a bound of $O(\kappa\cdot(n+m)\cdot t^2)$ for each objective on MDPs, where $\kappa$ is the number of strategy-iteration refinements required for the given input and objective. Finally, we make an experimental evaluation of our new algorithms on low-treewidth MCs and MDPs obtained from the DaCapo benchmark suite. Our experiments show that on MCs and MDPs with small treewidth, our algorithms outperform existing well-established methods by one or more orders of magnitude.

**Keywords:** Markov Chains · Markov Decision Processes · Parameterized Algorithms

*The research was partly supported by Austrian Science Fund (FWF) Grant No. NFN S11407-N23 (RiSE/SHiNE), Vienna Science and Technology Fund (WWTF) Project ICT15-003, the Facebook PhD Fellowship Program, and DOC Fellowship No. 24956 of the Austrian Academy of Sciences (ÖAW).

# 1 Introduction

*MCs.* Perhaps the most standard formalism for modeling randomness in discrete-time systems is that of discrete-time Markov Chains (MCs). MCs have immense applications in verification, and are used to express randomness both in the system and in the environment [12]. Besides the theoretical appeal, the analysis of MCs is also a core component in several model checkers [21,32].

*MDPs.* When the system exhibits both stochastic and non-deterministic behavior, the standard model of MCs is lifted to Markov Decision Processes (MDPs). For example, MDPs are used to model stochastic controllers, where the non-determinism models freedom of the controller and randomness models the behavior of the system. MDPs are also a topic of active study in verification [41,15].

*Quantitative Analysis.* Three of the most standard analysis objectives for MCs are the following: The *hitting probabilities* objective takes as input a set of target vertices $\mathfrak{T}$ of the MC, and asks to compute for each vertex $u$, the probability that a random walk from $u$ eventually hits $\mathfrak{T}$. The *discounted sum* objective takes as input a discount factor $\lambda \in (0, 1)$ and a reward function $R$ that assigns a reward to each edge of the MC. The task is to compute for each vertex $u$ the expected reward value of a random walk starting from $u$, where the value of the walk is the sum of the rewards along its edges, discounted by the factor $\lambda$ at each step*. Finally, the *mean payoff* objective is similar to discounted sum, except that the value of a walk is the long-run average of the rewards along its edges. In MDPs, the analyses ask for a strategy that maximizes the respective quantity.

*Analysis Algorithms.* Given the importance of quantitative objectives for MCs and MDPs, there have been various techniques for solving them efficiently. For MCs, the hitting probabilities and discounted sum objectives reduce to solving a system of linear equations [34]. For MDPs, all three objectives reduce to solving a linear program [34]. Besides the LP formulation, two popular approaches for solving quantitative objectives on MDPs are value iteration [3] and strategy iteration [30]. Value iteration is the most commonly used method in verification and operates by computing optimal policies for successive finite horizons. However, this process leads only to approximations of the optimal values, and for some objectives no stopping criterion for the optimal strategy is known [2]. In cases where such criteria are

*The undiscounted sum objective is obtained by letting $\lambda = 1$ and our algorithms for discounted sum can be slightly modified to handle this case, too.

known (e.g. [37]), the number of iterations necessary before the numbers can be rounded to provide an optimal solution can be extremely high [11]. Nevertheless, value iteration has proved to be very successful in practice and is included in many probabilistic model checkers, such as [32,21]. On the other hand, strategy iteration lies on the observation that given a fixed strategy, the MDP reduces to an MC, and hence one can compute the value of each vertex using existing techniques on MCs. Then, the strategy can be refined to a new strategy that improves the value of each vertex. The running time of strategy iteration can be written as $O(\kappa \cdot f)$, where $\kappa$ is the number of strategy refinements and $f$ is the time for evaluating the strategy. Although $\kappa$ can be exponentially large [22], it behaves as a small constant in practice, which makes strategy iteration work well in practice [31].

*Treewidth. Treewidth* is a well-studied graph parameter. There are many classes of graphs which arise in practice and have constant treewidth. A prime example is that Control Flow Graphs (CFGs) of `goto`-free programs in many programming languages have constant treewidth [42,28,18]. Treewidth has important algorithmic implications, as many graph problems admit (more) efficient solutions on graphs of low treewidth [16,42,25,26]. In program analysis, treewidth has been exploited to develop improvements for register allocation [42], algebraic-path analysis [14], data-flow analysis [17,9], data-dependence analysis [8], and model checking [35,24].

*Our Contributions.* The contributions of this work are as follows:

1. *Theoretical Contributions.* Our most general theoretical result is a linear-time algorithm for solving systems of linear equations whose primal graph has low treewidth. Given a linear system $S$ of $m$ equations over $n$ unknowns, and a tree decomposition of the primal graph of $S$ that has width $t$, our algorithm solves $S$ in time $O((n+m) \cdot t^2)$. Given an MC $M$ of treewidth $t$ and a corresponding tree decomposition, our algorithm directly implies similar running times for the hitting probabilities and discounted sum objectives for $M$. In addition, we develop an algorithm that solves the mean-payoff objective for $M$ in time $O((n+m) \cdot t^2)$. Our results on MCs also imply upper-bounds for the running time of strategy iteration on low-treewidth MDPs. Given an MDP $P$ with treewidth $t$ and a quantitative objective, our results imply that $P$ can be solved in time $O(\kappa \cdot (n+m) \cdot t^2)$, where $\kappa$ is the number of iterations until strategy iteration stabilizes for the respective input and objective.
2. *Practical Contributions.* We develop two practical algorithms for solving the hitting probabilities and discounted sum objectives on low-treewidth MCs. Although these algorithms

have the same worst-case complexity of $O((n+m)\cdot t^2)$ as our general solution, they avoid its most practically time-consuming step, i.e. applying the Gram-Schmidt process, and replace it with simple changes to the MC. We report on an implementation of these algorithms and their performance in solving MCs and MDPs with low treewidth. We perform an extensive comparison of our implementation and previous methods as follows:

(a) *Comparison with classical approaches*: We compare our algorithms for MCs against a heavily-optimized Gaussian elimination. For MDPs, we additionally compare with classical value-iteration and strategy-iteration.

(b) *Comparison with out-of-the-box tools*: We compare our implementation with standard industrial optimizers and probabilistic model checkers, including Matlab [33], lpsolve [4], Gurobi [27], PRISM [32] and Storm [21].

Our results show a consistent advantage of our new algorithms over all baseline methods, when the input models have small treewidth. Our algorithms outperform both the existing classical approaches for solving MCs/MDPs, and the highly-refined standard solvers.

*Closest Related Works.* The existing works closest to this paper are [13,25]. The work of [13] considers the maximal end-component decomposition and the almost-sure reachability set computation in low-treewidth MDPs. Note that these are both *qualitative* objectives, and thus very different from the *quantitative* objectives we consider here, which cannot be solved by [13]. Specifically, the main problem solved by [13] is almost-sure reachability, i.e. reachability with probability 1, which is a very special qualitative case of computing hitting probabilities. The work of [25] develops an algorithm for solving linear systems of low treewidth. Considering the computational complexity when applied to MCs/MDPs of treewidth $t$, the algorithms we develop in this work are a factor $t$ faster compared to [25]. On the practical side, the algorithms in [25] have more complicated intermediate steps, which we expect will lead to large constant factors in the runtime of their implementations. This being said, it is highly nontrivial to provide a practically efficient implementation of [25] and we are not aware of any such implementation.

## 2 Preliminaries

### 2.1 Markov Chains and Markov Decision Processes

*Discrete Probability Distributions.* Given a finite set $X$, a probability distribution over $X$ is a function $d : X \to [0,1]$ such that $\sum_{x\in X} d(x) = 1$. We denote the set of all probability distributions over $X$ by $\mathcal{D}(X)$.

*Markov Chains (MCs).* A *Markov chain* $C = (V, E, \delta)$ consists of a finite directed graph $(V, E)$ and a probabilistic transition function $\delta : V \to \mathcal{D}(V)$, such that for any pair $u, v$ of vertices, we have $\delta(u)(v) > 0$ only if $(u, v) \in E$. In an MC $C$, we start a random walk from a vertex $v_0 \in V$ and at each step, being in a vertex $v$, we probabilistically choose one of the successors of $v$ and go there. The probability with which a successor $w$ is chosen is given by $\delta(v)(w)$. Let $O \subseteq V^\omega$ be a measurable set of infinite paths on $V$, we use the notation $Pr_{v_0}(O)$ to denote the probability that our infinite random walk starting from $v_0$ is a member of $O$.

*Markov Decision Processes (MDPs).* A *Markov decision process* is a tuple $P = (V, E, V_1, V_P, \delta)$ which consists of a finite directed graph $(V, E)$, a partitioning of $V$ into two sets $V_1$ and $V_P$, and a probabilistic transition function $\delta : V_P \to \mathcal{D}(V)$, such that for any $(u, v) \in V_P \times V$, we have $\delta(u)(v) > 0$ only if $(u, v) \in E$. We assume that all vertices of an MDP have at least one outgoing edge. Intuitively, an MDP is a one-player game in which we have two types of vertices: those controlled by Player 1, i.e. $V_1$, and those that behave probabilistically, i.e. $V_P$.

*Strategies.* In an MDP $P$, a *strategy* is a function $\sigma : V_1 \to V$, such that for every $v \in V_1$ we have $(v, \sigma(v)) \in E$. Informally, a strategy is a recipe for Player 1 that tells her which successor to choose based on the current state[†]. Given an MDP $P$ with a strategy $\sigma$, we start a random walk from a vertex $v_0 \in V$ and at each step, being in a vertex $v$, choose the successor as follows: (i) if $v \in V_1$, then we go to $\sigma(v)$, and (ii) if $v \in V_P$ we act as in the case of MCs, i.e. we go to each successor $w$ with probability $\delta(v)(w)$. As before, given a measurable set $O \subseteq V^\omega$ of infinite paths on $V$, we define $Pr^\sigma_{v_0}(O)$ as the probability that our infinite random walk becomes a member of $O$. Note that an MDP with a fixed strategy $\sigma$ is basically an MC, in which for every $v \in V_1$ we have $\delta(v)(\sigma(v)) = 1$.

*Hitting Probabilities* [34]. Let $C = (V, E, \delta)$ be an MC and $\mathfrak{T} \subseteq V$ a designated set of *target* vertices. We define $\mathit{Hit}(\mathfrak{T}) \subseteq V^\omega$ as the set of all infinite sequences of vertices that intersect $\mathfrak{T}$. The *Hitting probability* $\mathit{HitPr}(u, \mathfrak{T})$ is defined as $Pr_u(\mathit{Hit}(\mathfrak{T}))$. In other words, $\mathit{HitPr}(u, \mathfrak{T})$ is the probability of eventually reaching $\mathfrak{T}$, assuming that we start our random walk at $u$. In case of MDPs, we assume that the player aims to maximize the hitting probability by choosing the best possible strategy. Therefore, we define $\mathit{HitPr}(u, \mathfrak{T})$ as $\max_\sigma Pr^\sigma_u(\mathit{Hit}(\mathfrak{T}))$.

*Discounted Sums of Rewards* [36]. Let $C = (V, E, \delta)$ be an MC and $R : E \to \mathbb{R}$ a *reward function* that assigns a real value to each edge. Also, let $\lambda \in (0, 1)$ be a *discount factor*.

[†]We only consider pure memoryless strategies because they are sufficient for our use-cases, i.e. there always exists an optimal strategy that is pure and memoryless [31].

Given an infinite path $\pi = v_0, v_1, \ldots$ over $(V, E)$, we define the total reward $R(\pi)$ of $\pi$ as

$$\sum_{i=0}^{\infty} \lambda^i \cdot R(v_i, v_{i+1}) = R(v_0, v_1) + \lambda \cdot R(v_1, v_2) + \lambda^2 \cdot R(v_2, v_3) + \ldots.$$

Let $u \in V$ be a vertex, we define *ExpDisSum*$(u)$ as the expected value of the reward of our random walk if we begin it at $u$, i.e. *ExpDisSum*$(u) := \mathbb{E}_u[R(\pi)]$. As in the previous case, when considering MDPs, we assume that the player aims to maximize the discounted sum, hence given an MDP $P = (V, E, V_1, V_P, \delta)$, a reward function $R$ and a discount factor $\lambda$, we define *ExpDisSum*$(u) := \max_\sigma \mathbb{E}^\sigma_u[R(\pi)]$.

*Mean Payoff* [36,31]. Let $C$ be an MC and $R$ a reward function. Given an infinite path $\pi = v_0, v_1, \ldots$ over $C$, we define the $n$-step average reward of $\pi$ as

$$R(\pi[0..n]) := \frac{1}{n} \sum_{i=1}^{n} R(v_{i-1}, v_i).$$

Given a start vertex $u \in V$, the expected *long-time average* or *mean payoff* value from $u$ is defined as *ExpMP*$(u) := \lim_{n\to\infty} \mathbb{E}_u[R(\pi[0..n])]$. In other words, *ExpMP*$(u)$ captures the expected reward per step in a random walk starting at $u$. For an MDP $P$, we define *ExpMP*$(u) := \max_\sigma \lim_{n\to\infty} \mathbb{E}^\sigma_u[R(\pi[0..n])]$. The limits in the former definitions are guaranteed to exist [36,31].

*Problems.* We consider the following classical problems for both MCs and MDPs:

- Given a target set $\mathfrak{T}$ compute *HitPr*$(u, \mathfrak{T})$ for every vertex $u$.
- Given a reward function $R$ and a discount factor $\lambda$ compute *ExpDisSum*$(u)$ for every vertex $u$.
- Given a reward function $R$, compute *ExpMP*$(u)$ for every vertex $u$.

*Solving MCs* [34]. A classical approach to the above problems for MCs is to reduce them to solving systems of linear equations. In case of hitting probabilities, we define one variable $x_u$ for each vertex $u$, whose value in the solution to the system would be equal to *HitPr*$(u, \mathfrak{T})$. The system is constructed as follows:

- We add the equation $x_\mathfrak{t} = 1$ for every $\mathfrak{t} \in \mathfrak{T}$, and
- For every vertex $u \notin \mathfrak{T}$ with successors $u_1, \ldots, u_k$, we add the equation $x_u = \sum_{i=1}^{k} \delta(u)(u_i) \cdot x_{u_i}$.

If every vertex can reach a target, then it is well-known that the resulting system has a unique solution in which the value assigned to each $x_u$ is equal to *HitPr*$(u, \mathfrak{T})$. A similar

approach can be used in the case of discounted sums. We define one variable $y_u$ per vertex $u$ and if the successors of $u$ are $u_1, \ldots, u_k$, then we add the equation $y_u = \sum_{i=1}^{k} \delta(u)(u_i) \cdot (R(u, u_i) + \lambda \cdot y_{u_i})$.

*Primal Graphs.* Let $S$ be a system of linear equations with $m$ equations and $n$ unknowns (variables). The primal graph $G(S)$ of $S$ is an undirected graph with $n$ vertices, each corresponding to one unknown in $S$, in which there is an edge between two unknowns $x$ and $y$ iff there exists an equation in $S$ that contains both $x$ and $y$ with non-zero coefficients.

*Solving MDPs.* There are two classical approaches to solving the above problems for MDPs. One is to reduce the problem to Linear Programming (LP) in a manner similar to the reduction from MC to linear systems [23]. The other approach is to use dynamic programming [3]. We consider a widely-used variety of dynamic programming, called *strategy iteration* or *policy iteration* [30].

*Strategy Iteration (SI)* [3]. In SI we start with an arbitrary initial strategy $\sigma_0$ and attempt to find a better strategy in each step. Formally, assume that our strategy after $i$ iterations is $\sigma_i$. Then, we compute $\mathit{val}_i(u) = \mathit{HitPr}^{\sigma_i}(u, \mathfrak{T})$ for every vertex $u$. This is equivalent to computing hitting probabilities in the MC that is obtained by considering our MDP together with the strategy $\sigma_i$. We use the values $\mathit{val}_i(u)$ to obtain a better strategy $\sigma_{i+1}$ as follows: for every vertex $v \in V_1$ with successors $v_1, v_2, \ldots, v_k$, we set $\sigma_{i+1}(v) = \arg\max_{v_j} \mathit{val}_i(v_j)$. In case of discounted sum, we let $\mathit{val}_i(u) = \mathit{ExpDisSum}^{\sigma_i}(u)$ and $\sigma_{i+1}(v) = \arg\max_{v_j} R(v, v_j) + \lambda \cdot \mathit{val}_i(v_j)$. We repeat these steps until we reach a point where our strategy converges. It is well-known that strategy iteration always converges to the optimal strategy $\sigma_\kappa$, and at that point the values $\mathit{val}_\kappa$ will be the desired hitting probabilities/discounted sums [30,23]. Given that SI solves the classic problems above on MDPs by several calls to a procedure for solving the same problems on MCs, our runtime improvements for MCs are naturally extended to MDPs. So, in the sequel we focus on MCs.

### 2.2 Tree Decompositions and Treewidth

Treewidth is a widely-used graph parameter. Intuitively, the treewidth of a graph is a measure of its tree-likeness. Only trees and forests have treewidth 1.

*Tree Decompositions* [39]. Given a directed or undirected graph $G = (V, E)$, a *tree decomposition* of $G$ is a tree $(T, E_T)$ such that:

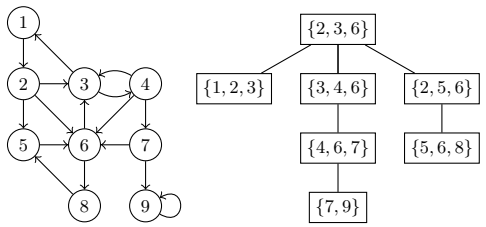


**Fig. 1.** A graph $G$ (left) and a tree decomposition of $G$ with width 2 (right).

- Each vertex $b \in T$ of the tree is associated with a subset $V_b \subseteq V$ of vertices of the graph. For clarity, we reserve the word "vertex" for vertices of $G$ and use the word "bag" to refer to vertices of $T$. Also, we define $E_b := \{(u, v) \in E \mid u, v \in V_b\}$.
- Each vertex appears in at least one bag, i.e. $\bigcup_{b \in T} V_b = V$.
- Each edge appears in at least one bag, i.e. $\bigcup_{b \in T} E_b = E$.
- Each vertex appears in a connected subtree of $T$. In other words, for all $b, b', b'' \in T$, if $b''$ is in the unique path between $b$ and $b'$, then $V_b \cap V_{b'} \subseteq V_{b''}$.

*Treewidth* [39]. The *width* of a tree decomposition is the size of its largest bag minus one, i.e. $w(T) = \max_{b \in T} |V_b| - 1$. A tree decomposition of $G$ is called *optimal* if it has the smallest possible width. The *treewidth* $tw(G)$ of $G$ is defined as the width of its optimal tree decomposition(s).

*Computing Treewidth and Tree Decompositions.* The problem of computing the treewidth of a graph is solvable in linear time when parameterized by the treewidth itself [7]. The algorithm in [7] also finds an optimal tree decomposition in linear time. Moreover, [42] proves control-flow graphs of structured programs have constant treewidth and provides a linear-time algorithm for producing the tree decomposition by a single parse of the program. In the sequel, we focus on linear-time algorithms for MCs and MDPs with constant treewidth. As is standard for treewidth-based approaches, we assume that an optimal tree decomposition is given as part of the input. This does not affect the complexity of our approach, as we can use [7,42] or tools such as [10] to obtain the tree decomposition in linear time.

## 3 Algorithms for MCs with Constant Treewidth

We now consider quantitative problems on MCs. As mentioned before, our improvements carry over to MDPs using SI. We build on classical state-elimination algorithms such as those

used in [19,29]. The main novelty of our approach is that we use the tree decompositions to obtain a suitable *order* for eliminating vertices. This specific ordering significantly reduces the runtime complexity of classical state-elimination algorithms from cubic to linear. Aside from the ordering, which is the main basis for our algorithmic improvements, the rest of this section is mostly well-known transformations on MCs. However, a new subtlety arises: while in general MCs there are several variants of elimination rules, in small-treewidth MCs we must also make sure the elimination step does not increase the treewidth.

### 3.1 A Simple Algorithm for Computing Hitting Probabilities

We begin by looking into the problem of computing hitting probabilities for general MCs without exploiting the treewidth. Without loss of generality, we can assume that our target set contains a single vertex. Otherwise, we add a new vertex $\mathfrak{t}$ and add edges with probability 1 from every target vertex to $\mathfrak{t}$. This will keep the hitting probabilities intact. Consider our MC $C = (V, E, \delta)$ and our target vertex $\mathfrak{t} \in V$. If there is only one vertex in the MC then there is not much to solve. We just return that $\mathit{HitPr}(\mathfrak{t}, \mathfrak{t}) = 1$. Otherwise, we take an arbitrary vertex $u \neq \mathfrak{t}$ and try to remove it from the MC to obtain a smaller MC that can in turn be solved using the same method. We should do this in a manner that does not change $\mathit{HitPr}(v, \mathfrak{t})$ for any vertex $v \neq u$. Figure 2 shows how to remove a vertex $u$ from $C$ in order to obtain a smaller MC $\overline{C} = (V \setminus \{u\}, \overline{E}, \overline{\delta})$[‡]. Basically, we remove $u$ and all of its edges, and instead add new edges from every predecessor $u'$ to every successor $u''$. We also update the transition function $\delta$ by setting $\overline{\delta}(u')(u'') = \delta(u')(u'') + \delta(u')(u) \cdot \delta(u)(u'')$. It is easy to verify that for every $v \neq u$, we have $\overline{\mathit{HitPr}}(v, \mathfrak{t}) = \mathit{HitPr}(v, \mathfrak{t})$. Hence, we can compute hitting probabilities for every vertex $v \neq u$ in $\overline{C}$ instead of $C$. Finally, if $u_1, u_2, \ldots, u_k$ are the successors of $u$ in $C$, we know that $\mathit{HitPr}(u, \mathfrak{t}) = \sum_{i=1}^{k} \delta(u)(u_i) \cdot \mathit{HitPr}(u_i, \mathfrak{t}) = \sum_{i=1}^{k} \delta(u)(u_i) \cdot \overline{\mathit{HitPr}}(u_i, \mathfrak{t})$. Hence, we can easily compute the hitting probability for $u$ using this formula. A pseudocode of this approach is available in Appendix A.

A special case arises when there is a self-loop transition from $u$ to $u$. If $\delta(u)(u) = 1$, i.e. $u$ is an absorbing trap, then we can simply remove $u$, noting that $\mathit{HitPr}(u, \mathfrak{t}) = 0$. On the other hand if $0 < \delta(u)(u) < 1$, then we should distribute $\delta(u)(u)$ proportionately among the other successors of $u$ because staying for a finite number of steps in the same vertex $u$ does not change the hitting property of a path, and the probability of staying at $u$ forever is 0.

[‡]We always use $\overline{C}$ to denote an MC that is obtained from $C$ by removing one vertex. We apply this rule across our notation, e.g. $\overline{\delta}$ is the respective transition function.


**Fig. 2.** Removing a vertex $u$. The vertex $u'$ is a predecessor of $u$ and $u''$ is one of its successors. The left side shows the changes when there is no edge from $u'$ to $u''$ and the right side shows the other case, where $(u', u'') \in E$. Edge labels are $\delta$ values.

Removing each vertex can take at most $O(n^2)$ time, given that it has $O(n)$ predecessors and successors. We should remove $n-1$ vertices, leading to a total runtime of $O(n^3)$, which is worse than the reduction to system of linear equations and then applying Gaussian elimination. However, the runtime can be significantly improved if we remove vertices in an order that guarantees every vertex has a low degree upon removal.

### 3.2 Computing Hitting Probabilities in Constant Treewidth

The main idea behind our algorithm is simple: we take the algorithm from the previous section and use tree decompositions to obtain an ordering for the removal of vertices. Given that we can choose any bag in $T$ as the root, without loss of generality, we assume that the target vertex $\mathfrak{t}$ is in the root bag[§]. We base our approach on the following lemmas:

**Lemma 1.** *Let $l \in T$ be a* leaf *bag of the tree decomposition $(T, E_T)$ of our MC $C$, and let $\bar{l}$ be the parent of $l$. If $V_l \subseteq V_{\bar{l}}$, then $(T \setminus \{l\}, E_T \setminus \{(\bar{l}, l)\})$ is also a valid tree decomposition for $C$.*

*Proof.* We just need to check that all the required properties of a tree decomposition hold after removal of $l$. Given that $V_l \subseteq V_{\bar{l}}$, any vertex that appears in $l$ is also in $\bar{l}$ and hence removal of $l$ does not cause any vertex to be unrepresented in the tree decomposition. The same applies to edges. Moreover, removing a *leaf* bag cannot disconnect the previously-connected set of bags containing a vertex.

**Lemma 2.** *Let $l \in T$ be a bag of the tree decomposition $(T, E_T)$ and assume that the vertex $u \in V$ only appears in $V_l$, i.e. it does not appear in the vertex set of any other bag. Then, the vertex $u$ has at most $|V_l|$ predecessors/successors in $C$.*

[§]If $|\mathfrak{T}| \geq 2$, we use the same technique as in the previous section to have only one target $\mathfrak{t}$. To keep the tree decomposition valid, we add $\mathfrak{t}$ to every bag.

*Proof.* If $u'$ is a predecessor/successor of $u$, then there is an edge between them. By definition, a tree decomposition should cover every edge. Hence, there should be a bag $b$ such that $u, u' \in V_b$. By assumption, $u$ only appears in $V_l$. Hence, every predecessor/successor $u'$ must also appear in $V_l$.

*The Algorithm.* The above lemmas provide a convenient order for removing vertices. At each step, we choose an arbitrary *leaf* bag $l$. If there is a vertex $u$ that appears *only* in $V_l$, then we eliminate $u$ as in Figure 2. In this case, Lemma 2 guarantees that $u$ has $O(t)$ predecessors and successors. Otherwise, $V_l \subseteq V_{\bar{l}}$ (recall that each vertex appears in a connected subtree) and we can remove $l$ from our tree decomposition according to Lemma 1. See Appendix A for a pseudocode.

*Example 1.* Consider the graph and tree decomposition in Figure 1 with an arbitrary transition probability function $\delta$ and target vertex $\mathfrak{t} = 6$. On this example, our algorithm would first choose an arbitrary leaf bag, say $\{7, 9\}$ and then realize that 9 has only appeared in this bag. Hence it removes vertex 9 from the MC using the same procedure as in the previous section. In the next iteration, it chooses the bag $\{7\}$ and realizes that the set of vertices in this bag is a subset of vertices that appear in its parent. Hence, it removes this unnecessary bag. The algorithm continues similarly, until only the target vertex 6 remains, at which point the problem is trivial. Figure 3 shows all the steps of our algorithm. Note that because the width of our tree decomposition is 2, at each step when we are removing a vertex $u$, it has at most 3 neighbors (counting itself).

Note that throughout this algorithm the tree decomposition remains valid, because we are only adding edges between vertices that are already in the same leaf bag $l$. Given that we remove at most $O(n)$ bags and $n - 1$ vertices and that removing each vertex takes only $O(t^2)$, the total runtime is $O(n \cdot t^2)$. Hence, we have the following theorem:

**Theorem 1.** *Given an MC with $n$ vertices and treewidth $t$ and an optimal tree decomposition of the MC, our algorithm computes hitting probabilities from every vertex to a designated target set in $O(n \cdot t^2)$.*

### 3.3 Computing Expected Discounted Sums in Constant Treewidth

We use a similar approach for handling the discounted sum problem. The only difference is in how a vertex is removed. Given an MC $C = (V, E, \delta)$, a tree decomposition $(T, E_T)$ of $C$, a reward function $R : E \to \mathbb{R}$ and a discount factor $\lambda \in (0, 1)$, we first add a new vertex

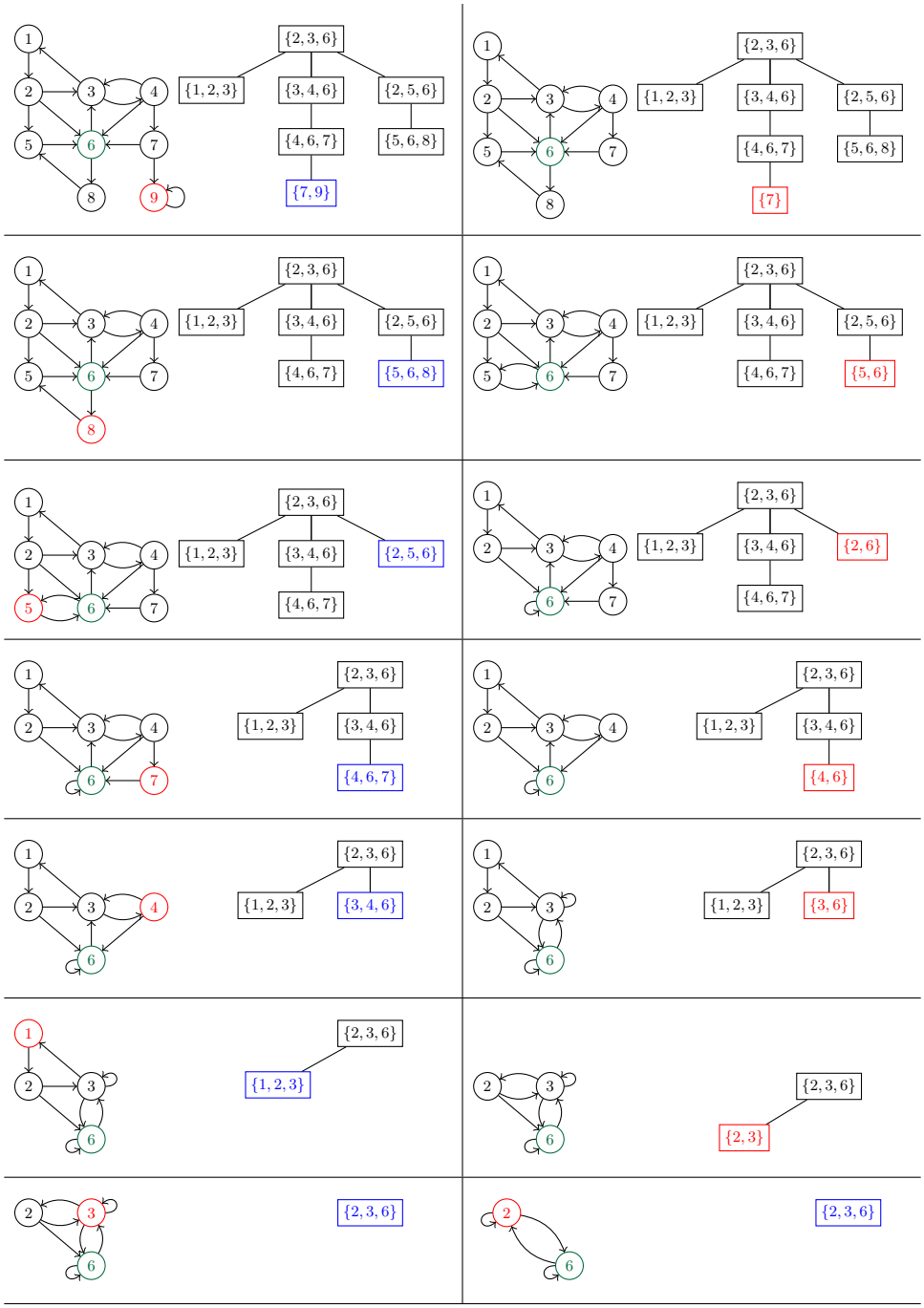


**Fig. 3.** The Steps Taken by Algorithm 2 on the Graph and Tree Decomposition in Figure 1. The target vertex $\mathfrak{t} = 6$ is shown in green. At each step the vertex/bag that is being removed is shown in red. An active bag whose vertices, but not itself, are considered for removal is shown in blue. After removing vertex 2, the graph has only one vertex and the base case of the algorithm is run.

called $\hat{\mathbf{1}}$ to the MC. The vertex $\hat{\mathbf{1}}$ is disjoint from all other vertices and only has a single self-loop with probability 1 and reward $1-\lambda$. In other words, we define $\delta(\hat{\mathbf{1}})(\hat{\mathbf{1}}) = 1$ and $R(\hat{\mathbf{1}}, \hat{\mathbf{1}}) = 1-\lambda$. We also add $\hat{\mathbf{1}}$ to the vertex set of every bag. The reason behind this gadget is that we have $\mathit{ExpDisSum}(\hat{\mathbf{1}}) = (1-\lambda) \cdot (1 + \lambda + \lambda^2 + \ldots) = 1$.

In our algorithm, the requirement that for all $u, v$ we should have $0 \leq \delta(u)(v) \leq 1$ is unnecessary and becomes untenable, too. Therefore, we allow $\delta(u)(v)$ to have any real value, and use the linear system interpretation of $C$ as in Section 2.1, i.e. instead of considering $C$ as an MC, we consider it to be a representation of the linear system $S_C$ defined as follows:

- For every vertex $u \in V$, the system $S_C$ contains one unknown $y_u$, and
- For every vertex $u \in V$, whose successors are $u_1, u_2, \ldots, u_k$, the system $S_C$ contains an equation $\mathfrak{e}_u := y_u = \sum_{i=1}^{k} \delta(u)(u_i) \cdot (R(u, u_i) + \lambda \cdot y_{u_i})$.

As mentioned in Section 2.1, in the solution to $S_C$, the value assigned to the unknown $y_u$ is equal to $\mathit{ExpDisSum}(u)$ in the MC $C$. However, the definition above does not depend on the fact that $C$ is an MC and can also be applied if $\delta$ has arbitrary real values.

Now suppose that we want to remove a vertex $u \neq \hat{\mathbf{1}}$ with successors $u_1, \ldots, u_k$ from $C$. This is equivalent to removing $y_u$ from $S_C$ without changing the values of other unknowns in the solution. Given that we have $y_u = \sum_{i=1}^{k} \delta(u)(u_i) \cdot (R(u, u_i) + \lambda \cdot y_{u_i})$, we can simply replace every occurrence of $y_u$ in other equations with the right-hand-side expression of this equation. If $u' \neq u$ is a predecessor of $u$, then we have $y_{u'} = A + \delta(u')(u) \cdot (R(u', u) + \lambda \cdot y_u)$, where $A$ is an expression that depends on other successors of $u'$. We can rewrite this equation as $y_{u'} = A + \delta(u')(u) \cdot R(u', u) + \sum_{i=1}^{k} \delta(u')(u) \cdot \delta(u)(u_i) \cdot \lambda \cdot (R(u, u_i) + \lambda \cdot y_{u_i})$. This is equivalent to obtaining a new $\overline{C}$ from $C$ by removing the vertex $u$ and adding the following edges from every predecessor $u'$ of $u$:

- An edge $(u', \hat{\mathbf{1}})$, such that $R(u', \hat{\mathbf{1}}) = 0$ and $\delta(u')(1) = \frac{1}{\lambda} \cdot (\delta(u')(u) \cdot R(u', u))$,
- An edge $(u', u_i)$ to every successor $u_i$ of $u$, such that $R(u', u_i) = R(u, u_i)$ and $\delta(u')(u_i) = \delta(u')(u) \cdot \delta(u)(u_i) \cdot \lambda$.

This construction is shown in Figure 4. As shown above, using this construction the value of $y_v$ remains the same in solutions of $S_C$ and $S_{\overline{C}}$. There are a few special cases, e.g. when the graph has parallel edges or self-loops. See Appendix B for details of handling such cases.

As in the previous section, we can solve the problem on the smaller $\overline{C}$ and then use the equation $\mathfrak{e}_u$ to compute the value of $y_u$ in the solution to $S_C$. This algorithm's runtime can be analyzed exactly as before. We have to remove $n$ vertices and each removal takes $O(n^2)$ for a total runtime of $O(n^3)$. To obtain a better algorithm that exploits tree decompositions, we can use the exact same removal order as in the previous section, leading to the same runtime,

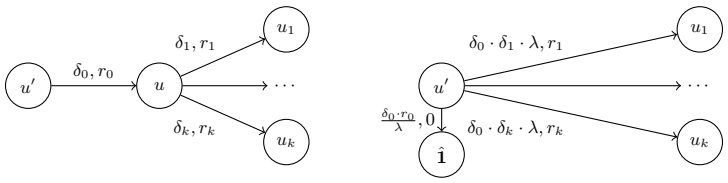


**Fig. 4.** Removing $u$ from $C$ (left) to obtain $\overline{C}$ (right). The vertex $u'$ is a predecessor of $u$ and $u_1, \ldots, u_k$ are its successors. Each edge is labelled with its $\delta$ and $R$ values.

i.e. $O(n \cdot t^2)$. Note that we have added $\hat{\mathbf{1}}$ to the associated vertex set of every bag, so the tree decomposition always remains valid throughout our algorithm. Given this discussion, we have the following theorem:

**Theorem 2.** *Given an MC with $n$ vertices and treewidth $t$ and an optimal tree decomposition of the MC, the algorithm described in this section computes expected discounted sums from every vertex of the MC in $O(n \cdot t^2)$.*

### 3.4 Systems of Equations with Constant-Treewidth Primal Graphs

The ideas used in the previous section can be extended to obtain faster algorithms for solving any linear system whose primal graph has a small treewidth. However, new subtleties arise, given that general linear systems might have no solution or infinitely many solutions. In contrast, the systems $S_C$ discussed in the previous section were guaranteed to have a unique solution. We consider a system $S$ of $m$ linear equations over $n$ real unknowns as input, and assume that its primal graph $G(S)$ has treewidth $t$. Our algorithm for solving $S$ is similar to our previous algorithms, and is actually what most students are taught in junior high school. We take an arbitrary unknown $x$ and choose an arbitrary equation $\mathfrak{e}$ in which $x$ appears with a non-zero coefficient. We then rewrite $\mathfrak{e}$ as $x = R_x$, where $R_x$ is a linear expression based on other unknowns. Finally, we replace every occurrence of $x$ in other equations with $R_x$ and solve the resulting smaller system $\overline{S}$. If $\overline{S}$ has no solutions or inifinitely many solutions, then so does $S$. Otherwise, we evaluate $R_x$ in the solution of $\overline{S}$ to get the solution value for $x$. Using this algorithm, we have to remove $O(n)$ unknowns. When removing $x$, we might have to replace an expression of size $O(n)$, i.e. $R_x$, in $O(m)$ potential other equations where $x$ has appeared. Hence, the overall runtime is $O(n^2 \cdot m)$.

Given a tree decomposition $(T, E_T)$ of the primal graph $G(S)$, we choose the unknows in the usual order, i.e. we always choose an unknown $x$ that appears only in a leaf bag. If $x$ does

not appear in any equations, then we can simply remove it and then $S$ is satisfiable iff $\overline{S}$ is satisfiable. Moreover, if $S$ is satisfiable, then it has infinitely many solutions, given that $x$ is not restricted. Otherwise, there is an equation $\mathfrak{e}$ in which $x$ appears with non-zero coefficient, and hence we can rewrite this equation as $x = R_x$. Note that $x$ has $O(t)$ neighbors in $G(S)$, given that it only appears in a leaf bag and all of its neighbors should also appear in the same bag, hence the length of $R_x$ is $O(t)$, too. The problem is that $x$ might have appeared in any of the other $O(m)$ equations. Hence, replacing it with $R_x$ in every equation will lead to a runtime of $O(m \cdot t)$. We repeat this for every unknown, so our total runtime is $O(n \cdot m \cdot t)$, which is not linear.

The crucial observation is that while $x$ might have appeared in as many as $m$ equations, not all of them are linearly independent. Let $\mathfrak{E}_x$ be the set of equations containing $x$ and $l$ be the leaf bag in which $x$ appears and assume that $V_l = \{x, y_1, \ldots, y_{k-1}\}$. Then the only unknowns that can appear together with $x$ in an equation are $y_1, \ldots, y_{k-1}$. In other words, all equations in $\mathfrak{E}_x$ are over $V_l$. Hence, we can apply the Gram-Schmidt process on $\mathfrak{E}_x$ to remove the unnecessary equations and only keep at most $k$ equations that form an orthogonal basis (or alternatively realize that the system is unsatisfiable). Given that we are operating in dimension $k = O(t)$, this will take $O(t^2 \cdot |\mathfrak{E}_x|)$ time. See Appendix A for a pseudocode. As in previous algorithms, our approach always keeps the tree decomposition valid. Moreover, as argued above, its runtime is $O((n+m) \cdot t^2)$, which is linear in the size of the system. Hence, we have the following theorem:

**Theorem 3.** *Given a system of $m$ linear equations over $n$ unknowns, its primal graph, and a tree decomposition of the primal graph with width $t$, our algorithm solves the system in time $O((n+m) \cdot t^2)$.*

The algorithm can easily be extended to find a basis for the solution set. Moreover, it can also be combined with the algorithms in the previous sections to solve the mean-payoff objective, leading to the following theorem:

**Theorem 4 (Proof and Details in Appendix C).** *Given an MC with $n$ vertices and treewidth $t$ and an optimal tree decomposition, expected mean payoffs from every vertex can be computed in $O(n \cdot t^2)$.*

## 4 Experimental Results

In this section, we report on a C/C++ implementation of our algorithms and provide a performance comparison with previous approaches in the literature.

*Compared Approaches.* We consider the hitting probability and discounted sum problems for MCs and MDPs. In the case of MCs, we directly use our algorithms from Section 3.2 and Section 3.3. For MDPs, we use strategy iteration, where we use the above algorithms for the strategy evaluation step in each iteration. We compare our approach with the following alternatives:

- *Classical Approaches.* In case of MCs, we compare against an implementation of Gaussian elimination (Gauss). For MDPs, we consider our own implementation of value iteration (VI) and strategy iteration (SI).
- *Numerical and Industrial Optimizers.* We use Matlab [33] and Gurobi [27] to solve systems of linear equalities corresponding to MCs. For MDPs, we use Matlab [33], Gurobi [27] and lpsolve [4] to handle the corresponding LPs.
- *Probabilistic Model Checkers.* The well-known model checkers Storm [21] and Prism [32] have standard procedures for computing hitting probabilities, but not for discounted sums. We therefore compare our runtimes on hitting probability instances with their runtimes.

Despite the fact that treewidth has been extensively studied in verification and model checking [35,24], including for the analysis of MDPs [13], to the best of our knowledge there are no benchmark suites consisting of low-treewidth MCs/MDPs. Previous works such as [13] do not provide any experimental results.

*Motivation for Benchmarks.* The main motivation to study MCs/MDPs with small treewidth is that they occur naturally in static program analysis, where a key algorithmic problem is reachability on the CFGs, e.g. data-flow analyses in frameworks such as IFDS are reduced to reachability [38]. Moreover, probability annotations of the CFG are useful in many contexts such as (i) in probabilistic programs where the branches are probabilistic; or (ii) when branch-profiling information is available that assigns probabilities to branch execution [40]. If we consider CFGs where all branches are deterministic or probabilistic, then we have MCs; and if there are also non-deterministic branches, then we have MDPs. In both cases, the reachability analysis in CFGs with probability annotation corresponds to the computation of hitting probabilities. Therefore, hitting probabilities can be used to answer questions like "given the branch profiles, compute the probability that a given pointer is null in some instruction". Additionally, [20] shows how discounted-sum objectives are relevant in the analysis of systems, e.g. with discounted-sum reachability we can model that a later bug is better than an earlier one. It is well-established that structured programs have small treewidth, both theoretically [42] and experimentally [28,18]. Thus, quantitative analysis of MCs/MDPs with

small treewdith is a relevant problem in program analysis, and we consider benchmarks from this domain.

*Benchmarks.* Given the points above, we used CFGs of the 40 Java programs from the DaCapo suite [5] as our benchmarks. They have between 33 and 103918 vertices and transitions. To obtain MDPs, we randomly (with probability 1/2) turned each vertex into a Player 1 or a probabilistic one. We assigned random probabilities to each outgoing edge of a probabilistic vertex. To obtain MCs, we did the same, except that all vertices are probabilistic. For the hitting probabilities problem, we chose one random vertex from each connected component of the control flow graphs as a target. In case of discounted sum, we uniformly chose a discount factor between 0 and 1 for each instance, and also assigned random integral rewards between $-1000$ to 1000 to each edge. Finally, we used JTDec [10] to compute tree decompositions. In each case the width of the obtained decomposition was no more than 9. Note that the time and memory used for computing a tree decomposition are negligible, given that it is obtained by a single pass over the program. See Appendix D for details of benchmarks.

*Results.* The runtimes for computing the values of hitting probabilities and discounted sums are shown in Figures 5–8. The benchmarks are on the $x$-axes and ordered by their size. Note that the $y$-axes are in *logarithmic scale*. For example, Figure 5 shows results for computing hitting probabilities in MCs, where Prism is the slowest tool by far, while our approach comfortably beats every other method. The gap is more apparent in MDPs (Figures 7–8). Overall, we see that our new algorithms consistently outperform both classical approaches like VI and SI, and highly optimized solvers and model checkers like Gurobi, Prism and Storm, by one or more orders of magnitude. Hence, the theoretical improvements are also realized in practice. See Appendix D for raw numbers. It is also noteworthy that the outputs of the different approaches agreed with each other within an error range of $10^{-5}$. Finally, note that our approach is only applicable to instances with small treewidth, such as CFGs of structured programs. For MCs/MDPs with arbitrary treewidth, the problem of computing an optimal tree decomposition is NP-hard [7].

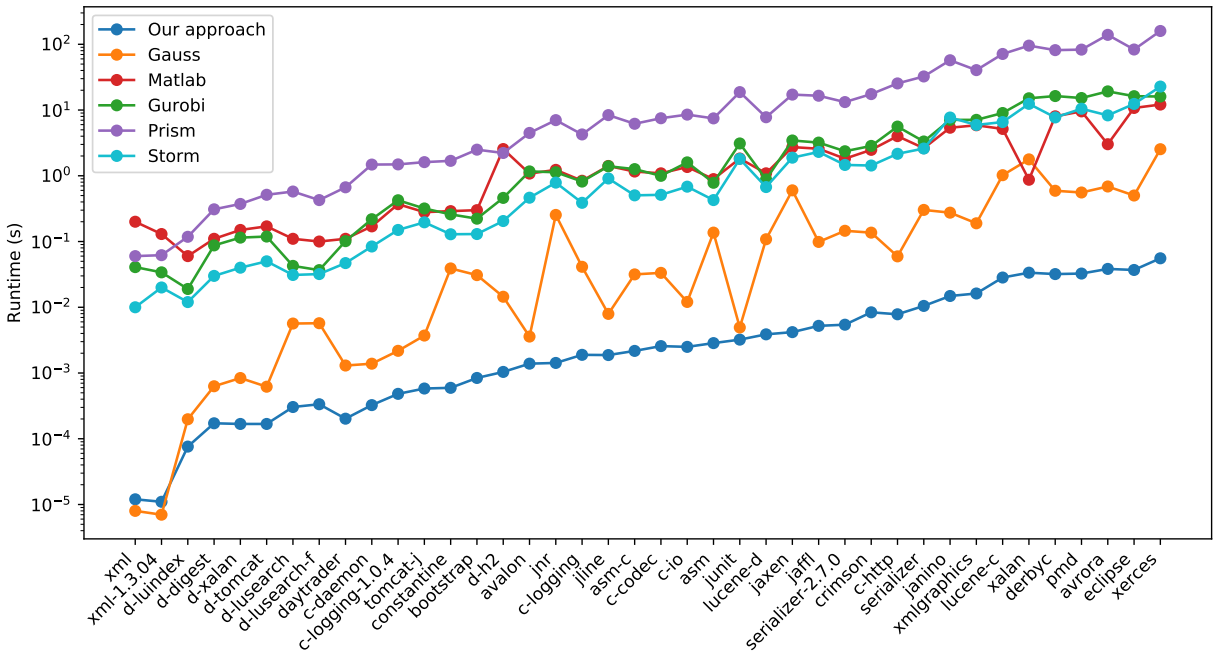


**Fig. 5.** Experimental Results for Computing Hitting Probabilities in MCs.

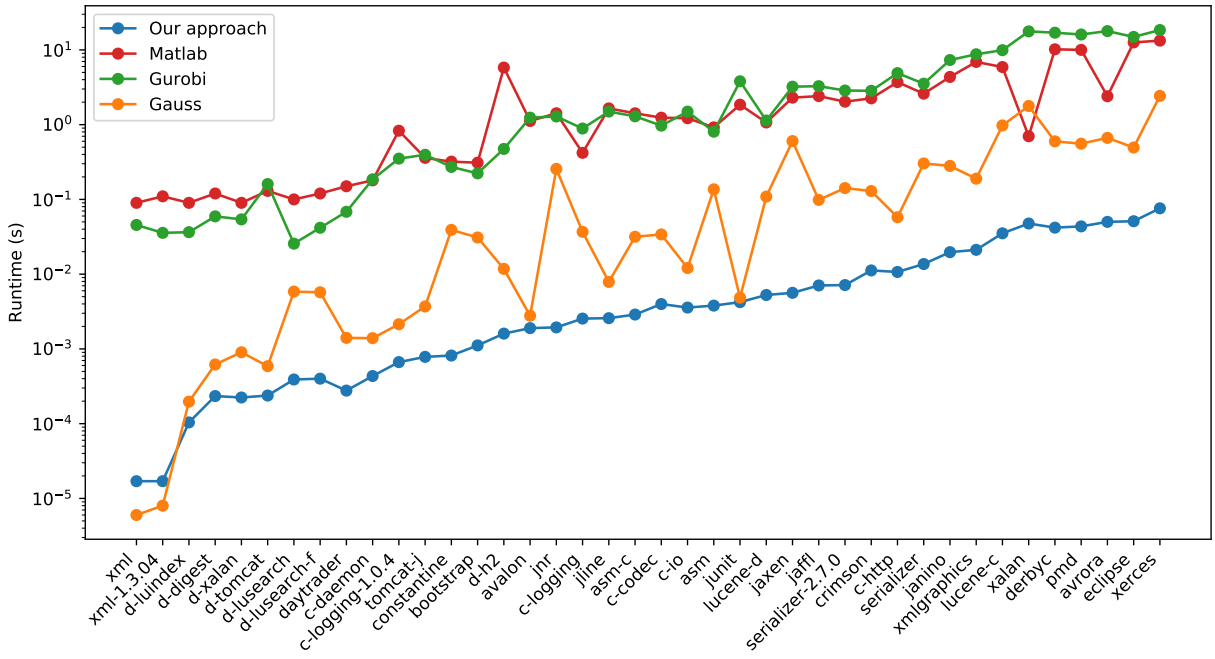


**Fig. 6.** Experimental Results for Computing Expected Discounted Sums in MCs.

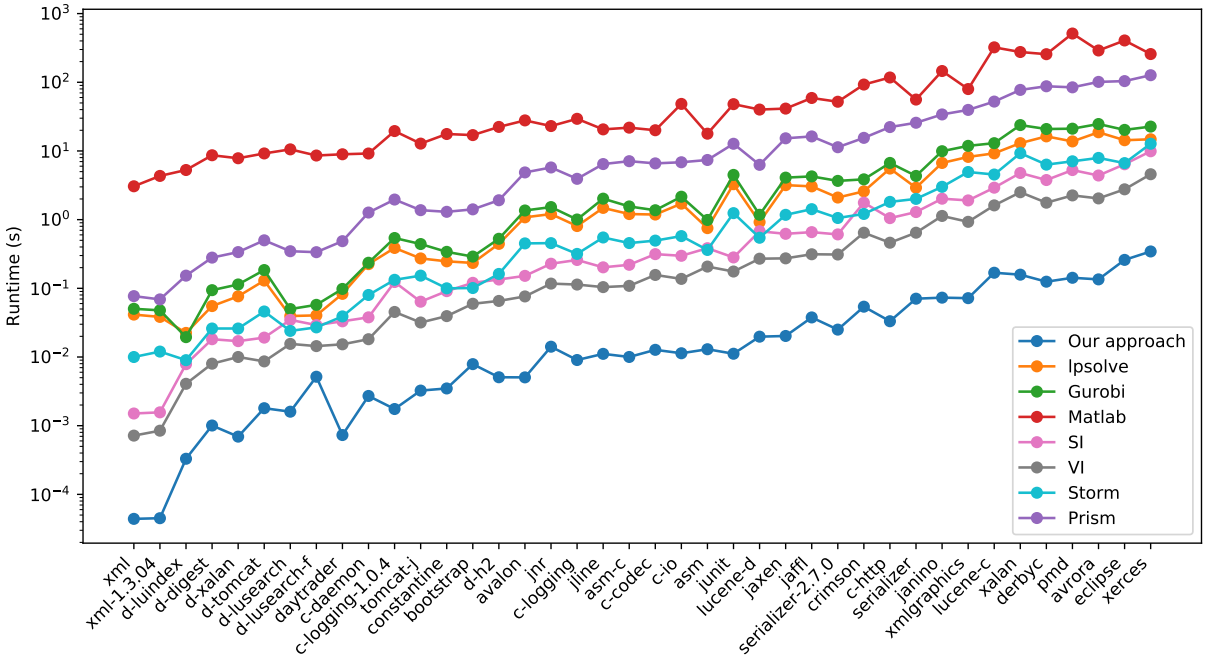


**Fig. 7.** Experimental Results for Computing Hitting Probabilities in MDPs.

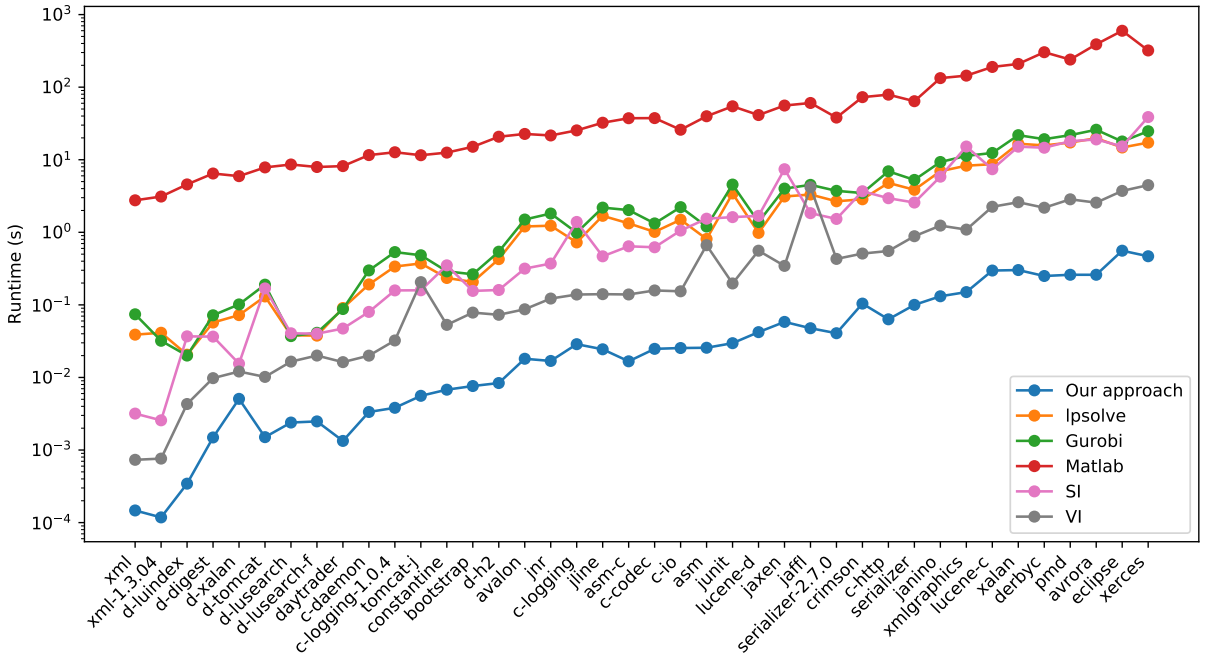


**Fig. 8.** Experimental Results for Computing Expected Discounted Sums in MDPs.

# References


1. Asadi, A., et al.: Faster algorithms for quantitative analysis of Markov chains and Markov decision processes with small treewidth. arXiv preprint:2004.08828 (2020)
2. Ashok, P., Chatterjee, K., Daca, P., Křetínský, J., Meggendorfer, T.: Value iteration for long-run average reward in Markov decision processes. In: CAV (2017)
3. Bellman, R.: A Markovian decision process. Journal of Mathematics and Mechanics (1957)
4. Berkelaar, M., Eikland, K., Notebaert, P.: lpsolve Linear Programming system
5. Blackburn, S.M., et al.: The DaCapo benchmarks: Java benchmarking development and analysis. In: OOPSLA (2006)
6. Bodlaender, H.L.: A tourist guide through treewidth. Acta cybernetica **11**(1-2) (1994)
7. Bodlaender, H.L.: A linear-time algorithm for finding tree-decompositions of small treewidth. SIAM Journal on Computing **25**(6) (1996)
8. Chatterjee, K., Choudhary, B., Pavlogiannis, A.: Optimal dyck reachability for data-dependence and alias analysis. In: POPL (2017)
9. Chatterjee, K., Goharshady, A.K., Ibsen-Jensen, R., Pavlogiannis, A.: Optimal and perfectly parallel algorithms for on-demand data-flow analysis. In: ESOP (2020)
10. Chatterjee, K., Goharshady, A.K., Pavlogiannis, A.: JTDec: A tool for tree decompositions in soot. In: ATVA (2017)
11. Chatterjee, K., Henzinger, T.A.: Value iteration. In: Model Checking (2008)
12. Chatterjee, K., Henzinger, T.A., Jobstmann, B., Singh, R.: Measuring and synthesizing systems in probabilistic environments. In: CAV (2010)
13. Chatterjee, K., Łącki, J.: Faster algorithms for Markov decision processes with low treewidth. In: CAV (2013)
14. Chatterjee, K., et al.: Algorithms for algebraic path properties in concurrent systems of constant treewidth components. TOPLAS **40**(3) (2018)
15. Chatterjee, K., et al.: Symbolic algorithms for graphs and Markov decision processes with fairness objectives. In: CAV (2018)
16. Chatterjee, K., et al.: Efficient parameterized algorithms for data packing. In: POPL (2019)
17. Chatterjee, K., et al.: Faster algorithms for dynamic algebraic queries in basic RSMs with constant treewidth. TOPLAS (2019)
18. Chatterjee, K., et al.: The treewidth of smart contracts. In: SAC (2019)
19. Daws, C.: Symbolic and parametric model checking of discrete-time Markov chains. In: ICTAC (2004)
20. De Alfaro, L., Henzinger, T.A., Majumdar, R.: Discounting the future in systems theory. In: ICALP (2003)
21. Dehnert, C., et al.: A storm is coming: A modern probabilistic model checker. In: CAV (2017)
22. Fearnley, J.: Exponential lower bounds for policy iteration. In: ICALP (2010)
23. Feinberg, E.A.: Handbook of Markov decision processes. Springer (2012)
24. Ferrara, A., Pan, G., Vardi, M.Y.: Treewidth in verification: Local vs. global. In: LPAR (2005)
25. Fomin, F.V., et al.: Fully polynomial-time parameterized computations for graphs and matrices of low treewidth. TALG **14**(3) (2018)
26. Goharshady, A.K., Mohammadi, F.: An efficient algorithm for computing network reliability in small treewidth. Reliability Engineering & System Safety **193** (2020)
27. Gurobi Optimization, L.: Gurobi optimizer (2019), `http://www.gurobi.com`
28. Gustedt, J., et al.: The treewidth of Java programs. In: ALENEX (2002)
29. Hahn, E.M., Hermanns, H., Wachter, B., Zhang, L.: Param: A model checker for parametric markov models. In: CAV (2010)

30. Howard, R.A.: Dynamic programming and Markov processes. (1960)
31. Křetínský, J., Meggendorfer, T.: Efficient strategy iteration for mean payoff in Markov decision processes. In: ATVA (2017)
32. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: CAV (2011)
33. MATLAB: The MathWorks Inc. (2018)
34. Norris, J.R.: Markov chains. Cambridge University Press (1998)
35. Obdržálek, J.: Fast mu-calculus model checking when tree-width is bounded. In: CAV (2003)
36. Puterman, M.L.: Markov Decision Processes. Wiley (2014)
37. Quatmann, T., Katoen, J.P.: Sound value iteration. In: CAV (2018)
38. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: POPL (1995)
39. Robertson, N., Seymour, P.D.: Graph minors. iii. planar tree-width. Journal of Combinatorial Theory, Series B **36**(1) (1984)
40. Smith, J.E.: A study of branch prediction strategies. In: ISCA (1998)
41. Tappler, M., Aichernig, B.K., Bacci, G., Eichlseder, M., Larsen, K.G.: $L^*$-based learning of Markov decision processes. In: FM (2019)
42. Thorup, M.: All structured programs have small tree width and good register allocation. Information and Computation **142**(2) (1998)

# A Pseudocodes

**Function** `ComputeHitProbs`$(C = (V, E, \delta), \mathfrak{t})$**:**
  **if** $V = \{\mathfrak{t}\}$ **then**
    $\mathit{HitPr}(\mathfrak{t}, \mathfrak{t}) \leftarrow 1$
  **else**
    Choose an arbitrary $u \in V \setminus \{\mathfrak{t}\}$
    **if** $\delta(u)(u) = 1$ **then**
      $\mathit{HitPr}(u, \mathfrak{t}) \leftarrow 0$
      `ComputeHitProbs` $((V \setminus \{u\}, E, \delta), \mathfrak{t})$
    **else**
      $f \leftarrow \frac{1}{1-\delta(u)(u)}$
      $\delta(u)(u) \leftarrow 0$
      $E \leftarrow E \setminus \{(u, u)\}$
      **foreach** $u'' \in V : (u, u'') \in E$ **do**
        $\delta(u)(u'') \leftarrow \delta(u)(u'') \cdot f$
      **foreach** $u' \in V : (u', u) \in E$ **do**
        **foreach** $u'' \in V : (u, u'') \in E$ **do**
          $\delta(u')(u'') \leftarrow \delta(u')(u'') + \delta(u')(u) \cdot \delta(u, u'')$
          $E \leftarrow E \cup \{(u', u'')\}$
      `ComputeHitProbs` $((V \setminus \{u\}, E, \delta), \mathfrak{t})$
      $\mathit{HitPr}(u, \mathfrak{t}) \leftarrow 0$
      **foreach** $u'' \in V : (u, u'') \in E$ **do**
        $\mathit{HitPr}(u, \mathfrak{t}) \leftarrow \mathit{HitPr}(u, \mathfrak{t}) + \delta(u, u'') \cdot \mathit{HitPr}(u'', \mathfrak{t})$

**Algorithm 1:** A Simple Algorithm for Computing Hitting Probabilities.

```
1  Function ComputeHitProbs(C = (V, E, δ), t, (T, E_T)):
2      if V = {t} then
3          HitPr(t, t) ← 1
4      else
5          repeat
6              Choose an arbitrary leaf bag l ∈ T
7              l̄ ← parent of l
8              if V_l ⊆ V_l̄ then
9                  T ← T \ {l}
10                 E_T ← E_T \ {(l̄, l)}
11             else
12                 Choose an arbitrary u ∈ V_l \ V_l̄
13                 V_l ← V_l \ {u}
14                 break
15         if δ(u)(u) = 1 then
16             HitPr(u, t) ← 0
17             ComputeHitProbs ((V \ {u}, E, δ), t)
18         else
19             f ← 1/(1−δ(u)(u))
20             δ(u)(u) ← 0
21             E ← E \ {(u, u)}
22             foreach u'' ∈ V_l : (u, u'') ∈ E do
23                 δ(u)(u'') ← δ(u)(u'') · f
24             foreach u' ∈ V_l : (u', u) ∈ E do
25                 foreach u'' ∈ V_l : (u, u'') ∈ E do
26                     δ(u')(u'') ← δ(u')(u'') + δ(u')(u) · δ(u, u'')
27                     E ← E ∪ {(u', u'')}
28             ComputeHitProbs ((V \ {u}, E, δ), t)
29             HitPr(u, t) ← 0
30             foreach u'' ∈ V_l : (u, u'') ∈ E do
31                 HitPr(u, t) ← HitPr(u, t) + δ(u, u'') · HitPr(u'', t)
```

**Algorithm 2:** Computing Hitting Probabilities using a Tree Decomposition.

```
1  Function SolveLinearSystem(S, G = (V, E), (T, E_T)):
2      if V = {∅} then
3          solution ← ∅
4          return solution
5      else
6          repeat
7              Choose an arbitrary leaf bag l ∈ T
8              l̄ ← parent of l
9              if V_l ⊆ V_l̄ then
10                 T ← T \ {l}
11                 E_T ← E_T \ {(l̄, l)}
12             else
13                 Choose an arbitrary x ∈ V_l \ V_l̄
14                 V_l ← V_l \ {x}
15                 break
16         foreach y_1, y_2 ∈ V_l : y_1 ≠ y_2 do
17             E ← E ∪ {(y_1, y_2)}
18         𝔈 ← equations in S that contain x with non-zero coefficient
19         S ← S \ 𝔈
20         if Gramm-Schmidt(𝔈) = Unsatisfiable then
21             return Unsatisfiable
22         𝔈 ← Gramm-Schmidt(𝔈)
23         if 𝔈 = ∅ then
24             if SolveLinearSystem(S, G \ {x}, (T, E_T)) = Unsatisfiable then
25                 return Unsatisfiable
26             else
27                 return Underdetermined
28         else
29             Choose an arbitrary 𝔢 ∈ 𝔈 and write it as x = R_x
30             𝔈 ← 𝔈 \ {𝔢}
31             foreach 𝔢′ ∈ 𝔈 do
32                 𝔢′ ← 𝔢′[R_x/x] //replace every occurrence of x with R_x
33             S ← S ∪ 𝔈
34             if SolveLinearSystem(S, G \ {x}, (T, E_T)) ∈ {Unsatisfiable, Underdetermined} then
35                 return SolveLinearSystem(S, G \ {x}, (T, E_T))
36             else
37                 solution ← SolveLinearSystem(S, G \ {x}, (T, E_T))
38                 solution ← solution[x ↦ [R_x]_solution]
39                 return solution
```

**Algorithm 3:** Solving a system $S$ of linear equations, given its primal graph $G = (V, E)$ and exploiting a tree decomposition $(T, E_T)$ of $G$. Note that $G$ is undirected. Lines 16–17 ensure that $G$ always remains a supergraph of the primal graph of $S$ and that $(T, E_T)$ always remains a valid tree decomposition of $G$.

## B Special Cases in Computing Expected Discounted Sums

There are two special cases that can cause our construction in Section 3.3 to fail. However, we can avoid both of these cases using simple transformations in the graph before applying this construction. We now describe how we handle each of them:

- *Parallel Edges.* If two edges with the same direction are created between the same pair $(u, v)$ of vertices, then we replace them with a single edge. If the $\delta$ values of initial edges were $\delta_1, \delta_2$ and their $R$ values were $r_1, r_2$, we set $\delta(u)(v) = \delta_1 + \delta_2$ and $R(u, v) = \frac{\delta_1 \cdot r_1 + \delta_2 \cdot r_2}{\delta_1 + \delta_2}$. It is straigthforward to verify that this transformation is sound, i.e. it does not change the solution of the corresponding system.
- *Self-loops.* If a self-loop $(u, u)$ appears in our graph, this is equivalent to having an equation $\mathfrak{e}_u := y_u = R$ in the linear system, in which $R$ is a linear expression that contains a non-zero multiple of $y_u$. In this case, we simplify this equation to $y_u = R'$ by moving the summand containing $y_u$ to the left hand side and multiplying both sides by a suitable factor. We then update the outgoing edges of $u$ in our graph to model the new system. Note that this update does not add any new edges to the graph, except possibly the edge $(u, \hat{\mathbf{1}})$ for handling leftover constant factors.

## C Computing Expected Mean Payoffs in Constant Treewidth

*Strongly Connected Components.* Given an MC $C = (V, E, \delta)$, a *Strongly Connected Component* (SCC) is a maximal subset $A \subseteq V$, such that for every pair of vertices $u, v \in A$, there is a path from $u$ to $v$ in $C$. An SCC $B$ is called a *Bottom Strongly Connected Component* (BSCC) if no other SCC is reachable from $B$. It is well-known that every vertex belongs to a unique SCC and that there is a linear-time algorithm that computes the SCCs and BSCCs of any given MC. An MC is called *ergodic* if its vertex set consists of only a single BSCC.

*Limiting Distribution* [34]. Given an ergodic MC $C = (B, E, \delta)$ with a single BSCC $B$ and an arbitrary vertex $u \in B$, we define the *limiting distribution* $\delta_{\lim}$ over $B$ as follows: $\delta_{\lim}(v) := \lim_{n\to\infty} \mathbb{E}_u \left[\frac{1}{n} \cdot |\{i \mid 0 \leq i < n \wedge \pi_i = v\}|\right]$, where $\pi$ is a random walk beginning at $u$. Informally, $\delta_{\lim}(v)$ is the fraction of time that we are expected to spend in vertex $v$, when we start a random walk in $C$. Note that due to ergodicity, the starting vertex of the random walk does not matter. We can similarly define a limiting distribution $\delta^E_{\lim}$ over the edges of $C$ by letting $\delta^E_{\lim}(u, v) := \delta_{\lim}(u) \cdot \delta(u)(v)$.

From the definition above, it is easy to see that the mean payoff value $\mathit{ExpMP}(u)$ is the same for every vertex $u \in B$ of the ergodic MC. More specifically, we have $\mathit{ExpMP}(u) =$

$\sum_{(v_1,v_2)\in E} R(v_1,v_2)\cdot \delta^E_{\lim}(v_1,v_2)$. Therefore, computing the *ExpMP* values is reduced to computing the limiting distribution.

Now consider a general MC $C=(V,E,\delta)$ and a vertex $u\in V$. If $u$ is in a BSCC $B$, then any path starting from $u$ will never leave $B$. Therefore, $\mathit{ExpMP}_V(u)=\mathit{ExpMP}_B(u)$. On the other hand, if $u$ is in a non-bottom SCC $A$, then the random walk beginning from $u$ will eventually reach a BSCC almost-surely (with probability 1). Let $B_1,B_2,\ldots$ be the BSCCs of $C$ and $b_i\in B_i$. Hence, given that we can ignore a finite prefix when computing mean payoffs, the expected mean payoff from $u$ is

$$\mathit{ExpMP}(u)=\sum_i \mathit{HitPr}(u,B_i)\cdot\mathit{ExpMP}(b_i)=\sum_i \mathit{HitPr}(u,b_i)\cdot\mathit{ExpMP}(b_i).$$

Every vertex in $B_i$ has the same expected mean payoff and will be reached from every other vertex in $B_i$ with probability 1, i.e. hitting probabilities between pairs of vertices in the same BSCC $B_i$ are always 1, hence the choice of $b_i$ is arbitrary.

We use the two observations above to compute expected mean payoffs in a given MC $C$. Algorithm 4 summarizes our approach. Hence, the problem is reduced to computing $\delta_{\lim}$ (Line 5) and hitting probabilities (Lines 11–12). We now explain how we handle each of these two subproblems.

```
1  Function ComputeExpMP(C = (V, E, δ)):
2      B_1, B_2, ... ← BSCCs of C
3      Choose an arbitrary b_i from each B_i
4      foreach B_i do
5          Compute δ_lim for (B_i, E ∩ (B_i × B_i), δ)
6          foreach (v_1, v_2) ∈ E ∩ (B_i × B_i) do
7              δ^E_lim(v_1, v_2) ← δ_lim(v_1) · δ(v_1)(v_2)
8          x ← Σ_{(v_1,v_2)∈E∩(B_i·B_i)} R(v_1, v_2) · δ^E_lim(v_1, v_2)
9          foreach u ∈ B_i do
10             ExpMP(u) ← x
11     foreach u ∈ V \ ⋃ B_i do
12         ExpMP(u) ← Σ_i HitPr(u, b_i) · ExpMP(b_i)
```

**Algorithm 4:** Computing expected mean payoffs in a given MC $C$.

*Computing Limiting Distribution of an Ergodic MC.* Let $C=(B,E,\delta)$ be an ergodic MC. We define the linear system $S_C$ as follows:

– We add a variable $x_u$ for each vertex $u\in B$.

– For each vertex $u \in B$ with *predecessors* $u_1, u_2, \ldots, u_k$, we add a constraint $x_u = \sum_{i=1}^{k} x_{u_i} \cdot \delta(u_i)(u)$.
– We add the constraint $\sum_{u \in B} x_u = 1$.

It is well-known that $S_C$ has a unique solution in which the value of each $x_u$ is equal to $\delta_{\lim}(u)$ [34]. Unfortunately, the last constraint includes all of the variables in the system and hence the primal graph of our system does not have constant treewidth. However, this is a minor restriction. We can consider the system $S'_C$ obtained by ignoring the last constraint. This system is homogeneous and its primal graph is the isomorphic to $(V, E)$ and has treewidth $t$. Hence, we can use the algorithm of Section 3.4 to find an arbitrary solution to $S'_C$. We can then scale all the values in our solution to satisfy the constraint $\sum_{u \in B} x_u = 1$, hence obtaining the unique solution of $S_C$. Therefore, Line 5 of Algorithm 4 takes $O(|B_i| \cdot t^2)$ time according to Theorem 3.

*Computing Expected Mean Payoff for non-BSCC vertices.* We can compute all the values of $\mathit{ExpMP}(u)$ for $u \in V \setminus \bigcup B_i$ (Lines 11–12) with *a single* call to our algorithm for hitting probabilities (Algorithm 2, Section 3.2). Note that Algorithm 2 does not rely on the premise that the function $\delta$ can only have values between 0 and 1. Hence, we can set all the $b_i$'s as targets, but when merging them to a single target $\mathfrak{t}$, we set $\delta(b_i)(\mathfrak{t}) = \mathit{ExpMP}(b_i)$, which was computed in Line 10. This ensures that the value computed for $\mathit{ExpMP}(u)$ is exactly the RHS of Line 11 in Algorithm 4. Using this trick, the runtime of Lines 10–11 of our algorithm is $O(n \cdot t^2)$ as per Theorem 1.

**Theorem 4.** *Given an MC with $n$ vertices and treewidth $t$ and an optimal tree decomposition, Algorithm 4 computes expected mean payoffs from every vertex in $O(n \cdot t^2)$.*

*Remark.* In SI over MDPs with mean payoff objectives, one also needs to compute additional values, called *potentials* or *biases* [31,36]. However, this computation is classically reduced to solving a system of linear equations whose primal graph is the MDP. Hence, the algorithm of Section 3.4 can be applied, and our improvements for computing mean payoff in MCs extend to MDPs.

## D Details of Experimental Results

*Experimental Setting.* The results were obtained on Ubuntu 18.04 with an Intel Core i5-7200U processor (2.5 GHz, 4 MB cache) using 8 GB of RAM.

*Details about Benchmarks.* Table 1 provides an overview of the DaCapo benchmarks used in our experimental results.

| **Benchmark** | $\|f\|$ | $\|V\|$ | $\|E\|$ | $t$ | **Benchmark** | $\|f\|$ | $\|V\|$ | $\|E\|$ | $t$ |
|---|---|---|---|---|---|---|---|---|---|
| asm-3.1 | 105 | 3044 | 3262 | 4 | daytrader | 12 | 339 | 332 | 3 |
| asm-commons-3.1 | 168 | 2404 | 2473 | 9 | derbyclient | 2097 | 37865 | 37997 | 9 |
| avalon-framework-4.2.0 | 153 | 1899 | 1849 | 4 | eclipse | 1974 | 45657 | 47039 | 8 |
| avrora-cvs-20091224 | 2539 | 43685 | 43521 | 9 | jaffl | 455 | 6099 | 6126 | 9 |
| bootstrap | 29 | 936 | 967 | 5 | janino-2.5.15 | 942 | 16861 | 17021 | 8 |
| commons-codec | 146 | 2728 | 2973 | 5 | jaxen-1.1.1 | 425 | 5490 | 5375 | 5 |
| commons-daemon | 28 | 453 | 437 | 4 | jline-0.9.95 | 209 | 2427 | 2387 | 5 |
| commons-httpclient | 693 | 9765 | 9772 | 5 | jnr-posix | 165 | 2040 | 1902 | 4 |
| commons-io-1.3.1 | 216 | 3216 | 3175 | 5 | junit-3.8.1 | 453 | 4356 | 4067 | 5 |
| commons-logging | 106 | 2231 | 2303 | 4 | lucene-core-2.4 | 1216 | 24906 | 25795 | 6 |
| commons-logging-1.0.4 | 53 | 689 | 677 | 3 | lucene-demos-2.4 | 120 | 4063 | 4413 | 7 |
| constantine | 34 | 776 | 758 | 4 | pmd-4.2.5 | 2131 | 37822 | 38672 | 7 |
| crimson-1.1.3 | 378 | 8572 | 9328 | 8 | serializer | 465 | 11038 | 11751 | 6 |
| dacapo-digest | 8 | 201 | 208 | 3 | serializer-2.7.0 | 330 | 6174 | 6447 | 9 |
| dacapo-h2 | 57 | 1293 | 1311 | 9 | tomcat-juli | 45 | 738 | 740 | 5 |
| dacapo-luindex | 3 | 84 | 87 | 4 | xalan-2.6.0 | 2088 | 35765 | 36946 | 8 |
| dacapo-lusearch | 5 | 282 | 300 | 4 | xerces_2_5_0 | 2129 | 50279 | 53639 | 9 |
| dacapo-lusearch-fix | 5 | 282 | 300 | 4 | xml-apis | 5 | 19 | 14 | 1 |
| dacapo-tomcat | 18 | 250 | 244 | 3 | xml-apis-1.3.04 | 5 | 19 | 14 | 1 |
| dacapo-xalan | 10 | 219 | 216 | 3 | xmlgraphics-1.3.1 | 1014 | 17677 | 17890 | 9 |

**Table 1.** Details of our benchmarks. In each case, $|f|$ is the number of functions in the benchmark, $|V|$ is the total number of vertices and $|E|$ is the total number of edges. Moreover, $t$ is the width of the tree decomposition constructed by JTDec [10]. Note that this is an upper-bound on the treewidth, given that JTDec is not an exact tool.

*Raw Numbers.* Tables 2–5 provide runtimes of each of the approaches mentioned in Section 4 over every benchmark.

| Benchmark | Runtime in seconds | | | | | |
|---|---|---|---|---|---|---|
| | Ours | Gauss | Matlab | Gurobi | Prism | Storm |
| xml-apis | 0.00001 | 0.00001 | 0.20000 | 0.04082 | 0.06000 | 0.01000 |
| xml-apis-1.3.04 | 0.00001 | 0.00001 | 0.13000 | 0.03398 | 0.06200 | 0.02000 |
| dacapo-luindex | 0.00008 | 0.00020 | 0.06000 | 0.01897 | 0.11800 | 0.01200 |
| dacapo-digest | 0.00017 | 0.00063 | 0.11000 | 0.08734 | 0.30900 | 0.03000 |
| dacapo-xalan | 0.00017 | 0.00084 | 0.15000 | 0.11467 | 0.37300 | 0.04000 |
| dacapo-tomcat | 0.00017 | 0.00062 | 0.17000 | 0.11905 | 0.51400 | 0.05000 |
| dacapo-lusearch | 0.00030 | 0.00566 | 0.11000 | 0.04260 | 0.57600 | 0.03100 |
| dacapo-lusearch-fix | 0.00034 | 0.00572 | 0.10000 | 0.03674 | 0.42700 | 0.03200 |
| daytrader | 0.00020 | 0.00130 | 0.11000 | 0.10136 | 0.66800 | 0.04700 |
| commons-daemon | 0.00033 | 0.00139 | 0.17000 | 0.21845 | 1.48100 | 0.08400 |
| commons-logging-1.0.4 | 0.00048 | 0.00218 | 0.37000 | 0.42445 | 1.49000 | 0.15000 |
| tomcat-juli | 0.00058 | 0.00371 | 0.28000 | 0.31819 | 1.61300 | 0.19600 |
| constantine | 0.00060 | 0.03897 | 0.29000 | 0.25880 | 1.68700 | 0.12900 |
| bootstrap | 0.00084 | 0.03084 | 0.30000 | 0.22259 | 2.49600 | 0.13000 |
| dacapo-h2 | 0.00104 | 0.01446 | 2.55000 | 0.46082 | 2.22300 | 0.20500 |
| avalon-framework-4.2.0 | 0.00139 | 0.00359 | 1.08000 | 1.16564 | 4.48500 | 0.46400 |
| jnr-posix | 0.00142 | 0.25385 | 1.23000 | 1.13165 | 7.02200 | 0.78700 |
| commons-logging | 0.00189 | 0.04127 | 0.84000 | 0.81669 | 4.25500 | 0.38800 |
| jline-0.9.95-SNAPSHOT | 0.00188 | 0.00793 | 1.41000 | 1.39634 | 8.37200 | 0.91000 |
| asm-commons-3.1 | 0.00217 | 0.03162 | 1.16000 | 1.26295 | 6.18000 | 0.50400 |
| commons-codec | 0.00257 | 0.03337 | 1.09000 | 0.99834 | 7.50300 | 0.51300 |
| commons-io-1.3.1 | 0.00250 | 0.01205 | 1.36000 | 1.59909 | 8.51900 | 0.68300 |
| asm-3.1 | 0.00285 | 0.13666 | 0.89000 | 0.78284 | 7.46700 | 0.42700 |
| junit-3.8.1 | 0.00322 | 0.00493 | 1.83000 | 3.10246 | 18.74200 | 1.80400 |
| lucene-demos-2.4 | 0.00388 | 0.10829 | 1.09000 | 0.88066 | 7.77000 | 0.67400 |
| jaxen-1.1.1 | 0.00419 | 0.60061 | 2.73000 | 3.44220 | 17.16700 | 1.88600 |
| jaffl | 0.00522 | 0.09861 | 2.58000 | 3.19677 | 16.49900 | 2.30600 |
| serializer-2.7.0 | 0.00543 | 0.14564 | 1.82000 | 2.36092 | 13.21600 | 1.46200 |
| crimson-1.1.3 | 0.00838 | 0.13650 | 2.48000 | 2.84511 | 17.47500 | 1.43200 |
| commons-httpclient | 0.00782 | 0.05948 | 4.01000 | 5.60121 | 25.39700 | 2.15900 |
| serializer | 0.01048 | 0.30260 | 2.63000 | 3.32304 | 32.41300 | 2.59700 |
| janino-2.5.15 | 0.01485 | 0.27481 | 5.38000 | 7.09017 | 57.19600 | 7.69100 |
| xmlgraphics-commons-1.3.1 | 0.01621 | 0.18939 | 5.87000 | 7.08701 | 40.47300 | 5.93600 |
| lucene-core-2.4 | 0.02834 | 1.02118 | 5.16000 | 9.00895 | 71.27600 | 6.56200 |
| xalan-2.6.0 | 0.03358 | 1.77685 | 0.87000 | 15.09464 | 95.42400 | 12.41900 |
| derbyclient | 0.03198 | 0.59163 | 8.02000 | 16.32040 | 81.51000 | 7.72300 |
| pmd-4.2.5 | 0.03257 | 0.55702 | 9.54000 | 15.22455 | 82.97800 | 10.46600 |
| avrora-cvs-20091224 | 0.03832 | 0.68720 | 3.02000 | 19.21516 | 139.14800 | 8.29700 |
| eclipse | 0.03693 | 0.50101 | 10.76000 | 16.24855 | 83.30500 | 12.38800 |
| xerces_2_5_0 | 0.05558 | 2.53732 | 12.11000 | 16.05042 | 159.37600 | 22.75000 |

**Table 2.** Detailed Experimental Results for **Hitting Probabilities in MCs**. All runtimes are reported in seconds. Note that Prism and Storm round the times to the nearest millisecond, while Matlab rounds to the nearest centisecond.

| Benchmark | Runtime in seconds | | | |
|---|---|---|---|---|
| | Ours | Gauss | Matlab | Gurobi |
| xml-apis | 0.00002 | 0.00001 | 0.09000 | 0.04552 |
| xml-apis-1.3.04 | 0.00002 | 0.00001 | 0.11000 | 0.03570 |
| dacapo-luindex | 0.00010 | 0.00020 | 0.09000 | 0.03646 |
| dacapo-digest | 0.00023 | 0.00062 | 0.12000 | 0.05931 |
| dacapo-xalan | 0.00022 | 0.00090 | 0.09000 | 0.05408 |
| dacapo-tomcat | 0.00024 | 0.00059 | 0.13000 | 0.16045 |
| dacapo-lusearch | 0.00039 | 0.00584 | 0.10000 | 0.02560 |
| dacapo-lusearch-fix | 0.00040 | 0.00572 | 0.12000 | 0.04174 |
| daytrader | 0.00028 | 0.00141 | 0.15000 | 0.06834 |
| commons-daemon | 0.00043 | 0.00139 | 0.18000 | 0.18583 |
| commons-logging-1.0.4 | 0.00067 | 0.00214 | 0.83000 | 0.35145 |
| tomcat-juli | 0.00078 | 0.00370 | 0.36000 | 0.39547 |
| constantine | 0.00082 | 0.03912 | 0.32000 | 0.27186 |
| bootstrap | 0.00111 | 0.03096 | 0.31000 | 0.22324 |
| dacapo-h2 | 0.00160 | 0.01178 | 5.81000 | 0.47357 |
| avalon-framework-4.2.0 | 0.00190 | 0.00278 | 1.12000 | 1.24184 |
| jnr-posix | 0.00194 | 0.25623 | 1.42000 | 1.28773 |
| commons-logging | 0.00255 | 0.03683 | 0.42000 | 0.88823 |
| jline-0.9.95-SNAPSHOT | 0.00258 | 0.00788 | 1.65000 | 1.49695 |
| asm-commons-3.1 | 0.00288 | 0.03166 | 1.42000 | 1.29334 |
| commons-codec | 0.00398 | 0.03401 | 1.24000 | 0.97073 |
| commons-io-1.3.1 | 0.00358 | 0.01210 | 1.22000 | 1.49369 |
| asm-3.1 | 0.00380 | 0.13651 | 0.92000 | 0.80580 |
| junit-3.8.1 | 0.00423 | 0.00485 | 1.85000 | 3.80143 |
| lucene-demos-2.4 | 0.00526 | 0.10898 | 1.07000 | 1.13659 |
| jaxen-1.1.1 | 0.00563 | 0.60335 | 2.30000 | 3.22225 |
| jaffl | 0.00706 | 0.09858 | 2.41000 | 3.27322 |
| serializer-2.7.0 | 0.00715 | 0.14182 | 2.03000 | 2.86329 |
| crimson-1.1.3 | 0.01122 | 0.12941 | 2.25000 | 2.83951 |
| commons-httpclient | 0.01070 | 0.05777 | 3.71000 | 4.89463 |
| serializer | 0.01367 | 0.30280 | 2.60000 | 3.53548 |
| janino-2.5.15 | 0.01970 | 0.28066 | 4.35000 | 7.31999 |
| xmlgraphics-commons-1.3.1 | 0.02116 | 0.18956 | 6.91000 | 8.74569 |
| lucene-core-2.4 | 0.03520 | 0.97595 | 5.90000 | 9.92471 |
| xalan-2.6.0 | 0.04760 | 1.77619 | 0.70000 | 17.70531 |
| derbyclient | 0.04193 | 0.59825 | 10.19000 | 16.99502 |
| pmd-4.2.5 | 0.04358 | 0.55667 | 10.01000 | 16.07630 |
| avrora-cvs-20091224 | 0.05008 | 0.66573 | 2.40000 | 17.86095 |
| eclipse | 0.05109 | 0.49472 | 12.60000 | 14.92674 |
| xerces_2_5_0 | 0.07604 | 2.42570 | 13.31000 | 18.45862 |

**Table 3.** Detailed Experimental Results for **Expected Discounted Sums in MCs**. All runtimes are reported in seconds. Note that Prism and Storm round the times to the nearest millisecond, while Matlab rounds to the nearest centisecond.

| Benchmark | Runtime in seconds | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | Ours | lpsolve | Gurobi | Matlab | SI | VI | Prism | Storm |
| xml-apis | 0.00004 | 0.04145 | 0.05033 | 3.06000 | 0.00151 | 0.00072 | 0.07700 | 0.01000 |
| xml-apis-1.3.04 | 0.00005 | 0.03836 | 0.04782 | 4.33000 | 0.00157 | 0.00085 | 0.06900 | 0.01200 |
| dacapo-luindex | 0.00033 | 0.02243 | 0.01947 | 5.28000 | 0.00787 | 0.00407 | 0.15300 | 0.00900 |
| dacapo-digest | 0.00101 | 0.05517 | 0.09404 | 8.66000 | 0.01816 | 0.00800 | 0.28000 | 0.02600 |
| dacapo-xalan | 0.00069 | 0.07645 | 0.11382 | 7.83000 | 0.01703 | 0.01001 | 0.33700 | 0.02600 |
| dacapo-tomcat | 0.00180 | 0.12966 | 0.18451 | 9.20000 | 0.01916 | 0.00863 | 0.50000 | 0.04600 |
| dacapo-lusearch | 0.00160 | 0.03950 | 0.04988 | 10.55000 | 0.03473 | 0.01553 | 0.34700 | 0.02400 |
| dacapo-lusearch-fix | 0.00516 | 0.04036 | 0.05746 | 8.58000 | 0.02918 | 0.01441 | 0.33600 | 0.02700 |
| daytrader | 0.00073 | 0.08304 | 0.09799 | 8.95000 | 0.03308 | 0.01531 | 0.48600 | 0.03900 |
| commons-daemon | 0.00271 | 0.22508 | 0.23568 | 9.18000 | 0.03780 | 0.01816 | 1.26900 | 0.08000 |
| commons-logging-1.0.4 | 0.00175 | 0.38711 | 0.54011 | 19.48000 | 0.12354 | 0.04529 | 1.96100 | 0.13300 |
| tomcat-juli | 0.00325 | 0.27254 | 0.44136 | 12.79000 | 0.06386 | 0.03181 | 1.37400 | 0.15300 |
| constantine | 0.00349 | 0.24723 | 0.33957 | 17.65000 | 0.09099 | 0.03940 | 1.30000 | 0.10000 |
| bootstrap | 0.00788 | 0.23418 | 0.28990 | 17.02000 | 0.12021 | 0.05957 | 1.41200 | 0.10100 |
| dacapo-h2 | 0.00507 | 0.44370 | 0.52646 | 22.39000 | 0.13434 | 0.06553 | 1.91600 | 0.16200 |
| avalon-framework-4.2.0 | 0.00505 | 1.08039 | 1.35913 | 27.85000 | 0.15185 | 0.07622 | 4.86800 | 0.45100 |
| jnr-posix | 0.01413 | 1.20555 | 1.52360 | 23.05000 | 0.22817 | 0.11717 | 5.74900 | 0.45500 |
| commons-logging | 0.00904 | 0.80701 | 1.00470 | 29.37000 | 0.25868 | 0.11347 | 3.92900 | 0.31700 |
| jline-0.9.95-SNAPSHOT | 0.01112 | 1.48252 | 2.02085 | 20.62000 | 0.20213 | 0.10417 | 6.44600 | 0.55000 |
| asm-commons-3.1 | 0.01002 | 1.20580 | 1.55601 | 21.84000 | 0.22011 | 0.10860 | 7.09800 | 0.45600 |
| commons-codec | 0.01272 | 1.18706 | 1.37049 | 20.03000 | 0.31467 | 0.15661 | 6.59400 | 0.49500 |
| commons-io-1.3.1 | 0.01134 | 1.70944 | 2.15621 | 48.41000 | 0.29763 | 0.13720 | 6.84900 | 0.57600 |
| asm-3.1 | 0.01298 | 0.75230 | 0.98912 | 17.84000 | 0.38381 | 0.20731 | 7.38900 | 0.36200 |
| junit-3.8.1 | 0.01116 | 3.28112 | 4.48130 | 48.02000 | 0.28184 | 0.17565 | 12.72300 | 1.24400 |
| lucene-demos-2.4 | 0.01978 | 0.92039 | 1.17712 | 40.06000 | 0.67695 | 0.26995 | 6.27900 | 0.54500 |
| jaxen-1.1.1 | 0.02025 | 3.17878 | 4.08971 | 41.38000 | 0.62182 | 0.27318 | 15.28600 | 1.17500 |
| jaffl | 0.03770 | 3.05149 | 4.26184 | 59.08000 | 0.65796 | 0.31335 | 16.27800 | 1.42100 |
| serializer-2.7.0 | 0.02506 | 2.09810 | 3.66582 | 52.06000 | 0.60972 | 0.31064 | 11.33300 | 1.05700 |
| crimson-1.1.3 | 0.05404 | 2.58277 | 3.86147 | 92.63000 | 1.76648 | 0.64496 | 15.51500 | 1.21000 |
| commons-httpclient | 0.03319 | 5.54198 | 6.69283 | 117.38000 | 1.05233 | 0.46138 | 22.27200 | 1.81400 |
| serializer | 0.07065 | 2.91949 | 4.32873 | 56.01000 | 1.28958 | 0.64523 | 25.73500 | 2.00500 |
| janino-2.5.15 | 0.07314 | 6.68790 | 9.91680 | 145.83000 | 2.01809 | 1.13186 | 34.01700 | 3.00600 |
| xmlgraphics-commons-1.3.1 | 0.07193 | 8.19395 | 11.86806 | 79.65000 | 1.90528 | 0.92933 | 39.59300 | 4.93900 |
| lucene-core-2.4 | 0.16863 | 9.23440 | 12.98207 | 322.52000 | 2.92001 | 1.61256 | 52.34900 | 4.53700 |
| xalan-2.6.0 | 0.15804 | 13.00583 | 23.80962 | 275.80000 | 4.77874 | 2.50689 | 77.43000 | 9.28700 |
| derbyclient | 0.12489 | 16.30398 | 20.86866 | 256.40000 | 3.77297 | 1.76492 | 87.46500 | 6.32500 |
| pmd-4.2.5 | 0.14238 | 13.77016 | 21.09903 | 512.47000 | 5.27168 | 2.25309 | 84.40400 | 7.07000 |
| avrora-cvs-20091224 | 0.13498 | 18.70873 | 24.66722 | 290.39000 | 4.39129 | 2.03788 | 101.28400 | 7.92000 |
| eclipse | 0.25944 | 14.32922 | 20.26574 | 406.15000 | 6.40507 | 2.76358 | 104.00300 | 6.65400 |
| xerces_2_5_0 | 0.34422 | 14.79255 | 22.62586 | 257.56000 | 9.89841 | 4.58320 | 126.73300 | 12.73800 |

**Table 4.** Detailed Experimental Results for **Hitting Probabilities in MDPs**. All runtimes are reported in seconds. Note that Prism and Storm round the times to the nearest millisecond, while Matlab rounds to the nearest centisecond.

| Benchmark | Runtime in seconds | | | | | |
|---|---|---|---|---|---|---|
| | Ours | lpsolve | Gurobi | Matlab | SI | VI |
| xml-apis | 0.00015 | 0.03889 | 0.07418 | 2.76000 | 0.00318 | 0.00073 |
| xml-apis-1.3.04 | 0.00012 | 0.04129 | 0.03201 | 3.11000 | 0.00257 | 0.00076 |
| dacapo-luindex | 0.00034 | 0.02059 | 0.02004 | 4.57000 | 0.03685 | 0.00431 |
| dacapo-digest | 0.00149 | 0.05713 | 0.07190 | 6.46000 | 0.03654 | 0.00978 |
| dacapo-xalan | 0.00507 | 0.07229 | 0.10148 | 5.94000 | 0.01549 | 0.01208 |
| dacapo-tomcat | 0.00151 | 0.13067 | 0.18988 | 7.84000 | 0.16863 | 0.01018 |
| dacapo-lusearch | 0.00239 | 0.03813 | 0.03714 | 8.60000 | 0.04062 | 0.01653 |
| dacapo-lusearch-fix | 0.00248 | 0.03770 | 0.04122 | 7.94000 | 0.04010 | 0.02004 |
| daytrader | 0.00134 | 0.09054 | 0.08769 | 8.17000 | 0.04721 | 0.01626 |
| commons-daemon | 0.00335 | 0.19132 | 0.29852 | 11.59000 | 0.08002 | 0.01998 |
| commons-logging-1.0.4 | 0.00382 | 0.33759 | 0.53407 | 12.68000 | 0.15830 | 0.03233 |
| tomcat-juli | 0.00558 | 0.37249 | 0.48284 | 11.54000 | 0.15890 | 0.20590 |
| constantine | 0.00678 | 0.23486 | 0.28992 | 12.54000 | 0.35064 | 0.05305 |
| bootstrap | 0.00762 | 0.20695 | 0.26406 | 15.08000 | 0.15577 | 0.07828 |
| dacapo-h2 | 0.00838 | 0.42674 | 0.54066 | 20.75000 | 0.16036 | 0.07301 |
| avalon-framework-4.2.0 | 0.01810 | 1.20308 | 1.50038 | 22.67000 | 0.31613 | 0.08697 |
| jnr-posix | 0.01688 | 1.23360 | 1.81516 | 21.60000 | 0.37101 | 0.12226 |
| commons-logging | 0.02875 | 0.72560 | 0.98885 | 25.38000 | 1.38934 | 0.13886 |
| jline-0.9.95-SNAPSHOT | 0.02444 | 1.68731 | 2.18862 | 32.36000 | 0.46697 | 0.14053 |
| asm-commons-3.1 | 0.01669 | 1.32881 | 2.02500 | 37.33000 | 0.64163 | 0.13915 |
| commons-codec | 0.02478 | 1.01284 | 1.32599 | 37.47000 | 0.62215 | 0.15808 |
| commons-io-1.3.1 | 0.02544 | 1.49117 | 2.22190 | 25.90000 | 1.05737 | 0.15389 |
| asm-3.1 | 0.02566 | 0.80930 | 1.20244 | 39.74000 | 1.53952 | 0.66412 |
| junit-3.8.1 | 0.02967 | 3.43031 | 4.54870 | 54.49000 | 1.61903 | 0.19736 |
| lucene-demos-2.4 | 0.04212 | 0.98169 | 1.38443 | 41.28000 | 1.68492 | 0.55803 |
| jaxen-1.1.1 | 0.05828 | 3.11529 | 4.00512 | 55.82000 | 7.40472 | 0.34462 |
| jaffl | 0.04769 | 3.31774 | 4.50510 | 60.56000 | 1.83675 | 4.18679 |
| serializer-2.7.0 | 0.04071 | 2.67593 | 3.72690 | 38.10000 | 1.52886 | 0.43002 |
| crimson-1.1.3 | 0.10414 | 2.85201 | 3.47488 | 72.86000 | 3.67908 | 0.50956 |
| commons-httpclient | 0.06327 | 4.79292 | 6.92428 | 78.85000 | 2.95078 | 0.55254 |
| serializer | 0.10003 | 3.85245 | 5.27190 | 64.05000 | 2.57026 | 0.88358 |
| janino-2.5.15 | 0.13146 | 6.94878 | 9.30685 | 133.24000 | 5.84305 | 1.23715 |
| xmlgraphics-commons-1.3.1 | 0.15004 | 8.27969 | 11.30344 | 144.15000 | 15.20058 | 1.08951 |
| lucene-core-2.4 | 0.29733 | 8.68221 | 12.39745 | 189.91000 | 7.40384 | 2.24896 |
| xalan-2.6.0 | 0.30192 | 16.62097 | 21.80253 | 208.95000 | 15.14658 | 2.59791 |
| derbyclient | 0.25004 | 15.75678 | 19.19841 | 303.18000 | 14.63486 | 2.18177 |
| pmd-4.2.5 | 0.25993 | 17.26150 | 21.83183 | 240.07000 | 17.99462 | 2.84897 |
| avrora-cvs-20091224 | 0.25995 | 19.79896 | 25.89820 | 389.31000 | 19.05126 | 2.56352 |
| eclipse | 0.55813 | 14.74519 | 17.84516 | 598.65000 | 15.26801 | 3.71952 |
| xerces_2_5_0 | 0.46794 | 17.21913 | 24.70113 | 320.46000 | 38.72352 | 4.46858 |

**Table 5.** Detailed Experimental Results for **Expected Discounted Sums in MDPs**. All runtimes are reported in seconds. Note that Prism and Storm round the times to the nearest millisecond, while Matlab rounds to the nearest centisecond.

*Remark.* As mentioned before, our inputs contain tree decompositions of the MCs/MDPs. Note that the time used to compute the tree decompositions is negligible, given that constant-width tree decompositions of CFGs are computed by a single pass of the program parse tree [42,10].