Faster Algorithms for Markov Decision Processes with Low Treewidth

Krishnendu Chatterjee[1] and Jakub Łącki[2]

[1] IST Austria (Institute of Science and Technology Austria)
[2] Institute of Informatics, University of Warsaw, Poland

Abstract. We consider two core algorithmic problems for probabilistic verification: the maximal end-component decomposition and the almost-sure reachability set computation for Markov decision processes (MDPs). For MDPs with treewidth k, we present two improved static algorithms for both the problems that run in time $O(n \cdot k^{2.38} \cdot 2^k)$ and $O(m \cdot \log n \cdot k)$, respectively, where n is the number of states and m is the number of edges, significantly improving the previous known $O(n \cdot k \cdot \sqrt{n \cdot k})$ bound for low treewidth. We also present decremental algorithms for both problems for MDPs with constant treewidth that run in amortized logarithmic time, which is a huge improvement over the previously known algorithms that require amortized linear time.

1 Introduction

In this work we will present efficient static and decremental algorithms for two core graph algorithmic problems in probabilistic verification when the graph has low treewidth. We start with the basic description of the model, the problem, and its importance.

Markov Decision Processes with Parity Objectives. The standard model of systems in probabilistic verification that exhibit both probabilistic and non-deterministic behavior are *Markov decision processes (MDPs)* [20]. MDPs have been used for control problems for stochastic systems [18], where nondeterminism represents the freedom of the controller to choose a control action, and the probabilistic component of the behavior describes the system response to control actions; as well as in many other applications [13,2,19]. A *specification* describes the set of good behaviors of the system. In the verification and control of stochastic systems the specification is typically an ω-regular set of paths. The class of ω-regular languages extends classical regular languages to infinite strings, and provides a robust specification language to express all commonly used specifications, such as safety, liveness, fairness, etc. [28]. A canonical way to define such ω-regular specifications are *parity* objectives. Hence MDPs with parity objectives provide the mathematical framework to study problems such as the verification and control of stochastic systems.

The Analysis Problems. There are two types of analysis for MDPs with parity objectives. The *qualitative analysis* problem given an MDP with a parity

objective, asks for the computation of the set of states from where the parity objective can be ensured with probability 1 (almost-sure winning). The more general *quantitative analysis* asks for the computation of the maximal probability at each state with which the controller can satisfy the parity objective.

Significance of Qualitative Analysis. The qualitative analysis of MDPs is an important problem in verification. In several applications the controller must ensure that the correct behavior arises with probability 1. For example, in analysis of randomized embedded schedulers, the relevant questions is whether every thread progresses with probability 1 [15]. Moreover, even in applications where it is sufficient to satisfy the specification with probability $p < 1$, the correct choice of p is a challenging problem, due to the simplifications introduced during modeling; for example, for randomized distributed algorithms it is common to require correctness with probability 1 (see, e.g., [25,22,27]). Furthermore, in contrast to quantitative analysis, qualitative analysis is robust to numerical perturbations and precise transition probabilities, and consequently the algorithms for qualitative analysis are discrete and combinatorial. Finally, the best known algorithms for quantitative analysis of MDPs with parity objectives first perform the qualitative analysis, and then a quantitative analysis on the result of the qualitative analysis [13,14,10].

Core Algorithmic Problems. The qualitative analysis of MDPs with parity objectives relies on two graph algorithmic problems: (1) the maximal end-component decomposition; and (2) the almost-sure reachability set computation. An end-component C in an MDP is a set of states that is strongly connected and closed (no probabilistic transition from C leaves C), and a maximal end-component is an end-component which is maximal with respect to inclusion ordering. The maximal end-component (MEC) problem generalizes the scc (maximal strongly connected component) decomposition problem for directed graphs, and recurrent classes for Markov chains. The almost-sure reachability set for a set U of target vertices is the set of states such that it can be ensured that the set U is reached with probability 1 (in other words, it is the qualitative analysis for reachability objectives). The qualitative analysis problem for MDPs with parity objectives with d-priorities can be solved with $\log d$ calls to the MEC decomposition problem and one call to the almost-sure reachability problem [6]. Thus the MEC decomposition and the almost-sure reachability set computation are the core algorithmic problems required for the qualitative analysis of MDPs with parity objectives. In addition to qualitative analysis of MDPs with parity objectives, several algorithms for quantitative analysis of MDPs with quantitative objectives such as lim sup and lim inf objectives [8], combination of mean-payoff and parity objectives [9], and multi-objective mean-payoff objectives [5], rely crucially on the MEC decomposition problem.

Dynamic Algorithms. In the design and analysis of probabilistic systems it is natural that the systems under verification are developed incrementally by adding choices or removing choices for player 1, whereas the probabilistic choices which represent choice of nature or uncertainty remain unchanged. Hence there is a clear motivation to obtain dynamic algorithms for MEC decomposition and

almost-sure reachability set for MDPs that achieve a better running time than recomputation from scratch when player-1 edges are inserted or deleted.

Previous Results. The current best known algorithms for both the MEC decomposition and the almost-sure reachability set computation require $O(m \cdot \min(\sqrt{m}, n^{2/3}))$ time [6,7], where n is the number of states and m is the number of transitions (edges). Using a well-known fact that graphs of treewidth k have $O(n \cdot k)$ edges, one can obtain $O(n \cdot k \cdot \sqrt{n \cdot k})$ algorithms for MEC decomposition and almost-sure reachability set computation (they follow directly from the general $O(m \cdot \sqrt{m})$-time algorithm). The best known incremental and decremental algorithms for both problems require amortized linear time ($O(n)$ time) [6].

Our Contributions. In this work we consider MDPs with low treewidth. The concept of treewidth and tree decomposition of graphs was introduced in [26]. On one hand treewidth is a very relevant graph theoretic notion that measures how a graph can be decomposed into a tree, on the other hand, most systems developed in practice have low treewidth. For example, it has been shown that the control flow graphs of goto free Pascal programs have treewidth at most 3, and that the control flow graphs of goto free C programs have treewidth at most 6 [29]. It was also shown in [29] that tree decompositions, which are very costly to compute in general, can be generated in linear time with small constants for these control flow graphs. Our main results are efficient static and decremental algorithms for the MEC decomposition and the almost-sure reachability set computation for MDPs with low treewidth. Several benchmarks in PRISM are probabilistic programs written in programming languages mentioned above and consequently have small treewidth, and our results are relevant for such MDPs. The details of our contribution are as follows:

1. We present two improved static algorithms both for the MEC decomposition and the almost-sure reachability set computation for MDPs with treewidth k that run in time $O(n \cdot k^{2.38} \cdot 2^k)$ and $O(m \cdot \log n \cdot k)$, respectively, where n is the number of states and m is the number of edges (also note that for treewidth k we have $m = O(n \cdot k)$). For MDPs with low treewidth, our new linear-time algorithms are significant improvements over the previous known $O(n \cdot k \cdot \sqrt{n \cdot k})$ algorithms for both the problems.
2. We present decremental algorithms for the MEC decomposition and the almost-sure reachability set computation for MDPs with treewidth k that require $O(k \cdot \log n)$ amortized time, which is a huge improvement for constant treewidth over the previous algorithms that require $O(n)$ amortized time.

Our key technical contribution is as follows: for MDPs we establish a separation property for the almost-sure reachability set that allows us to use tree decomposition to obtain the $O(n \cdot k^{2.38} \cdot 2^k)$-time static algorithm. A similar intuition also works for the MEC decomposition problem. We then view the MEC decomposition and the almost-sure reachability set computation problems as decremental graph problems, and use dynamic graph algorithmic techniques to obtain the $O(m \cdot \log n \cdot k)$-time static algorithms and the decremental algorithms. Note that when edges are inserted, the treewidth of the graph may increase and the tree

decomposition can change. Thus, incremental algorithms with polylogarithmic amortized cost remain an interesting open question (even for scc decomposition). Proofs omitted for space available in [11].

Related Works. The notion of treewidth is studied in context of many graph theoretic algorithms, see [4] for an excellent survey. In verification, the problem of low and medium treewidth has been considered for efficient algorithms for parity games: a polynomial time algorithm for parity games with constant treewidth was presented in [24]; a recent improved result for constant treewidth was presented in [17]; and the algorithmic problem of parity games with medium treewidth was considered in [16]. Though the games problem has been studied with the treewidth restriction, to the best of our knowledge, improved algorithms for MDPs have not been considered with the treewidth restriction.

2   Preliminaries

In this section we first present the basic graph theoretic definitions of the MEC decomposition and the almost-sure reachability set computation, and then define the notions of treewidth.

2.1   MEC Decomposition and Almost-Sure Reachability

Markov decision processes (MDPs). A *Markov decision process (MDP)* $G = ((V, E), (V_1, V_P), \delta)$ consists of a finite directed *MDP graph* (V, E), a partition (V_1, V_P) of the *finite* set V of vertices, and a probabilistic transition function $\delta \colon V_P \to \mathcal{D}(V)$, where $\mathcal{D}(V)$ denotes the set of probability distributions over the vertex set V, such that for all vertices $u \in V_P$ and $v \in V$ we have $uv \in E$ iff $\delta(u)(v) > 0$. An edge $uv \in E$ is a *player-1* edge if $u \in V_1$. For the algorithmic problems we will consider, the probabilistic transition function will not be relevant and we will consider the MDP graph along with the partition.

Maximal end-component decomposition. For the maximal end-component decomposition, the input is a directed graph $G = (V, E)$ and a partition (V_1, V_P) of its vertex set (i.e., the MDP graph and the partition). An end-component U is a set of vertices such that the subgraph induced by U is strongly connected and for each edge $uv \in E$, if $u \in U \cap V_P$ then $v \in U$. If U_1 and U_2 are two end-components and $U_1 \cap U_2 \neq \emptyset$, then $U_1 \cup U_2$ is also an end-component. The maximal end-component (MEC) decomposition consists of all the maximal end-components of V and all vertices of V that do not belong to any MEC.

Almost-sure reachability. For almost-sure reachability, the input is an MDP and a target set $U \subseteq V$ of vertices, and the goal is to compute the set A of vertices, such that player 1 can ensure that the set U is reached with probability 1. We first note that given the target set U, we can add a new vertex s as the new target vertex, and transform the set U such that all out-edges from vertices in U end up in s, and the vertex s has only a self-loop. Thus we will consider the

case when the target set is a single vertex s. We first reduce the computation of the almost-sure reachability set for a target vertex s to the following problem. The input is a directed graph $G = (V, E)$, a partition (V_1, V_P) of its vertex set (the MDP graph and the partition), and a target vertex $s \in V$. The goal is to compute a maximal (w.r.t inclusion) subset $Q \subseteq V$, such that the following two conditions are satisfied:

- for every $q \in Q$, there exists a path from q to s consisting only of vertices in Q (global condition), and
- for every $uv \in E$, if $u \in Q \cap V_P$, then $v \in Q$ (local condition).

First observe that if $Q_1 \subseteq V$ and $Q_2 \subseteq V$ both satisfy the global and the local conditions, then so does $Q_1 \cup Q_2$. It follows that there is a unique maximum set $A^* \subseteq V$ that satisfies both the global and the local conditions. The resulting set A^* is the almost-sure reachability set (in the following also called an *ASR set*). Let A be the almost-sure reachability set and A^* be the largest set that satisfies the two conditions (the global and the local conditions).

Lemma 1. *We have $A = A^*$.*

Since $A = A^*$ we consider the graph theoretic problem of computation of A^* (i.e., the largest set satisfying the global and the local conditions).

Notations. Let G be a directed graph. We denote its vertex and edge set by $V(G)$ and $E(G)$, respectively. By $G[S]$ we denote the subgraph of G induced on vertices belonging to S, whereas by $G \setminus S$ we denote the subgraph of G induced on $V(G) \setminus S$. A *separator* is a subset $S \subseteq V(G)$, such that $G \setminus S$ has more connected components than G (when all edges are treated as undirected).

2.2   Tree Decomposition of Graphs

We begin by introducing some definitions depicted in Fig. 1.

Definition 1. *Let $G = (V, E)$ be an undirected graph. A* tree decomposition *of G is a pair (B, T), where B is a family $B_1, \ldots, B_n$ of subsets of V (called* bags*) and T is a tree, whose nodes are sets B_i. The decomposition satisfies the following properties:*

1. *$\bigcup B_i = V$ (bags cover vertices).*
2. *For every $uv \in E$ there exists B_j, such that $u, v \in B_j$ (bags cover edges).*
3. *For every $v \in V$ the sets B_i containing v form a connected subtree of T.*

Definition 2. *The* width *of a tree decomposition (B, T) is equal to $\max_{B_i \in B} |B_i| - 1$. The* treewidth *of an undirected graph is the minimal possible width of its tree decomposition.*

The concept of treewidth grasps the *sparseness* of a graph. Treewidth of a tree is equal to 1, while cliques on n vertices have treewitdth $n - 1$. Note that the definitions are given for undirected graphs, but they can also be applied to directed graphs. In such case, we treat all edges as undirected.
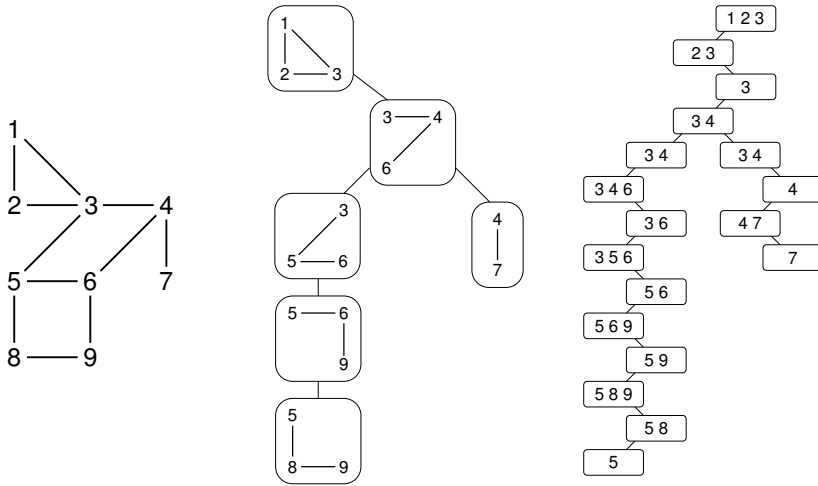
Fig. 1. A sample graph (left), its tree decomposition (center, edges covered by each bag have been marked for illustration) and a nice tree decomposition (right)

Definition 3. *A tree decomposition* (B, T) *is called* nice *if T is a rooted tree and each of its nodes B_i belongs to one of the following four types:*

1. *leaf — B_i is a leaf of T and $|B_i| = 1$.*
2. *introduce — B_i has a single child B_j and $B_i = B_j \cup \{v\}$.*
3. *forget — B_i has a single child B_j and $B_i = B_j \setminus \{v\}$.*
4. *join — B_i has two children B_j and B_k, and $B_i = B_j = B_k$.*

Theorem 1 ([3]). *Let G be a graph of treewidth k. Assuming that k is a constant, the tree decomposition of G of width k can be computed in $O(n)$ time.*

Lemma 2 (see e.g. [21]). *A tree decomposition can be transformed, in linear time, into a nice tree decomposition of the same width, consisting of $O(n)$ nodes.*

We also use the following well-known fact, which can be derived from the definition. Informally, consider a vertex t_B of a tree decomposition T of a graph G. Assume that it contains a bag $B \subset V(G)$. Denote the connected components of $T \setminus \{t_B\}$ by $T_1, \ldots, T_k$. Then, the trees T_i correspond to connected components of $G \setminus B$, namely bags from each T_i cover vertices from one connected component.

Lemma 3. *Let B be a bag in a node t_B of the tree decomposition of G. Consider the connected components $T_1, \ldots, T_k$ of $T \setminus \{t_B\}$. Then the following hold:*

1. *Either B is a separator in G or all but one T_i consist solely of bags that are subsets of B.*
2. *Each path from a vertex $u \notin B$ covered with a bag in T_i to a vertex $v \notin B$ covered with a bag in T_j $(i \neq j)$ goes through a vertex in B.*

Observe that a vertex not belonging to B can be covered by bags from at most one T_i. This is because the set of bags covering a given vertex forms a connected subgraph of T.

3 Algorithms for MDPs with Constant Tree-Width

In this section we will first present an algorithm for computing the ASR set, whose running time depends linearly on the size of the input MDP graph, where the input graph has constant treewidth. We will then present the linear-time algorithm for MEC decomposition for MDPs with constant treewidth graphs. The algorithms require that a tree decomposition of the graph of width k is given and run in time that is exponential in k. If k is a constant, the decomposition can be computed in linear time (see Theorem 1). To simplify presentation, we use Lemma 2 to transform the decomposition to a *nice* one.

3.1 Almost-Sure Reachability

Our algorithm for the ASR set computation is based on the following separation property.

Lemma 4. *Let B be a subset of $V(G)$, such that the target vertex s belongs to B. Denote the connected components of $G \setminus B$ by $C_1, \ldots, C_k$. Assume that we know the intersection of the ASR set A with B. For each $i = 1, \ldots, k$, construct the subgraph of G induced on $C_i \cup B$. Add to this graph a set of edges $\{vs | v \in A \cap B\}$, thus obtaining a* patched *component $\overline{C}_i$. Denote by A_i the ASR set in $\overline{C}_i$. Then we have $A = A_1 \cup \ldots \cup A_k$.*

Lemma 4 says that if we know $A \cap B$, then we can compute the ASR set independently in each (patched) connected component of $G \setminus B$ and then simply merge the results. Since we assume that G has low treewidth, it also has separators of small size. Thus, in the algorithm we can guess $A \cap B$, by checking all possibilities. We do not prove the separation property explicitly. Instead, we give the algorithm inspired with this property and then prove its correctness. The property will follow from Lemma 6. Let us now describe the details.

Denote the nice tree decomposition of G by T. We add the target vertex s to every bag of T. Note that this might increase the width of the decomposition by at most one, but the modified T is still a valid tree decomposition. However, T is no longer a nice decomposition, as the leaf nodes now contain two vertices. We fix this, by adding a child $\{s\}$ to every leaf. Then we choose an arbitrary leaf as the root.

The algorithm is based on a bottom-up dynamic programming on T. Fix a node d of T, and assume that it contains a bag B_d. Denote by G_d the subgraph of G induced on the vertices enclosed in the bags from the subtree rooted at d. By Lemma 3, B_d separates $G_d \setminus B_d$ from the rest of the graph.

Now, according to Lemma 4, for each subset $B' \subseteq B_d$ we should add edges $\{vs | v \in B'\}$ to G_d and compute the ASR set of the obtained graph. However, we do a slightly different thing: instead of adding edges, we just treat all vertices of B' as target vertices (note that this has the same effect as adding edges from vertices in B' to s). This motivates the following definition of a partial solution. Partial solution is defined with respect to a subgraph of $G_d \subseteq G$, and, informally, it is the set of vertices from G_d that will be included in the ASR set.

Definition 4. *A* partial solution *for a node d is a subset of $V(G_d)$. A partial solution P is called* valid, *if the following hold.*

   i. For every $v \in P \cap V_P$ and every edge $vu \in E(G_d)$, we have $u \in P$.
  ii. For every $v \in P$ there exists a path in P that connects v to some vertex in $P \cap B_d$.

We denote by $P(B', d)$ the maximal (w.r.t. inclusion) valid partial solution (for node d) which satisfies $P(B', d) \cap B_d = B'$.[1] Observe that the definition is unambiguous, since the union of two valid partial solutions is a valid partial solution. However, it might be the case that for some choice of B' there are no feasible valid partial solutions. In such a case we set $P(B', d) = \perp$. We later show that if $B' = A \cap B_d$, then $P(B', d) = A \cap V(G_d)$.

    The algorithm considers possible ways of including a subset of B_d in the ASR set, by iterating through all *valid* subsets $B' \subseteq B_d$. A subset $B' \subseteq B_d$ is valid, if it contains the target s and for each $v \in B' \cap V_P$ and every edge $vu \in E \cap (B_d \times B_d)$, we have $u \in B'$. In particular, for any valid partial solution P containing s, the set $P \cap B_d$ is a valid subset.

    In addition to $P(B', d)$, for each valid $B' \subseteq B$ and each pair of vertices $x, y \in B'$, we compute whether there exists an x-to-y path consisting of vertices contained in $P(B', d)$. Formally, we compute the transitive closure of $G[P(B', d)]$, restricted to B'. In the following this transitive closure is denoted by $TC(B', d)$. Note that it is a subset of $B_d \times B_d$.

    The algorithm is run bottom-up on T. For a given node d and each valid subset B' it computes $P(B', d)$ and $TC(B', d)$, using the values from the children of d. There are four cases to consider, one for each type of node. In the description, we assume that the value $\perp$ is *propagating*. This means, that the result of any set operation involving $\perp$ is $\perp$.

- **Leaf** The bag contains a single vertex s (the target), so the transitive closure is empty and we set $P(\{s\}, d) = \{s\}$.
- **Join** Denote the children of d by c_1 and c_2. In this case, we set $P(B', d) = P(B', c_1) \cup P(B', c_2)$, so the transitive closures from the children have to be combined, i.e. $TC(B', d) = (TC(B', c_1) \cup TC(B', c_2))^*$. The asterisk denotes the operation of computing the transitive closure.
- **Introduce** Denote the introduced vertex by w and the child of d by c. For all valid subsets $B' \subseteq B_d$ that do not contain w, we set $P(B', d) = P(B', c)$ and $TC(B', d) = TC(B', c)$. If $w \in B'$, then $P(B', d) = P(B' \setminus \{w\}, c) \cup \{w\}$. Thus, to compute the transitive closure in this case, we take $TC(B' \setminus \{w\}, c)$, add all edges incident to w and compute the transitive closure of the obtained set. Hence, $TC(B', d) = (TC(B' \setminus \{w\}, c) \cup \{wz \in E(G) | z \in B'\} \cup \{zw \in E(G) | z \in B'\})^*$.
- **Forget** Denote the vertex that is forgotten by w and the child of d by c. Hence, the bag in the child B_c is equal to $B_d \cup \{w\}$. We check whether we

---

[1] In the end we prove slightly less about the values $P(\cdot, \cdot)$ that are computed by the algorithm, but it is convenient to think about them this way.

can include w in $P(B', d)$. For this, condition (ii) (of Definition 4) has to hold, i.e., there has to be a path in $P(B', d)$ that connects w to some vertex in B'. We claim that it suffices to check, whether w has any out-edges in $TC(B' \cup \{w\}, c)$. If this is the case, then w is connected to some vertex from B' in $P(B' \cup \{w\}, c)$, so $P(B', d) = P(B' \cup \{w\}, c)$ and we can set $TC(B', d) = TC(B' \cup \{w\}, c) \cap (B' \times B')$. Otherwise, we just copy the result from the child, that is set $P(B', d) = P(B', c)$ and $TC(B', d) = TC(B', c)$.

Finally, the ASR set computed by the algorithm is stored in $P(\{s\}, r)$. We now prove the correctness of the algorithm with the following two lemmas (proof of Lemma 5 in [11]).

Lemma 5. *For each node d and each valid subset $B' \subseteq B_d$, if $P(B', d) \neq \perp$, then $P(B', d)$ is a valid partial solution and $TC(B', d)$ is computed correctly.*

Lemma 6. *Let A be the maximum ASR set. For each node d, $P(A \cap B_d, d) = A \cap V(G_d)$.*

Proof. The proof proceeds by induction on the depth of the subtree rooted in d. First, it is easy to see that $A \cap B_d$ is a valid subset for d. Moreover, $A \cap V(G_d)$ is a valid partial solution for d. Let us check condition (ii) of Definition 4. For each $v \in A$ there exists an v-to-s path p in A. Denote by v_l the last vertex of p that lies inside $A \cap V(G_d)$. By Lemma 3, $v_l \in B_d$ and consequently also $v_l \in A \cap B_d$.

- **Leaf** $P(A \cap B_d, d) = P(\{s\}, d) = \{s\} = A \cap V(G_d)$.
- **Join** By induction hypothesis we have $P(A \cap B_{c_i}, c_i) = A \cap V(G_{c_i})$, for $i = 1, 2$. From the definition $P(A \cap B_d, d) = P(A \cap B_d, c_1) \cup P(A \cap B_d, c_2) = P(A \cap B_{c_1}, c_1) \cup P(A \cap B_{c_2}, c_2) = (A \cap V(G_{c_1})) \cup (A \cap V(G_{c_2})) = A \cap (V(G_{c_1}) \cup V(G_{c_2})) = A \cap V(G_d)$.
- **Introduce** If A does not contain the introduced vertex w, then $P(A \cap B_d, d) = P(A \cap B_c, c) = A \cap V(G_c) = A \cap (V(G_d) \setminus \{w\}) = A \cap V(G_d)$. Otherwise, if $w \in A$ we have $P(A \cap B_d, d) = P((A \cap B_d) \setminus \{w\}, c) \cup \{w\} = (A \cap V(G_c)) \cup \{w\} = A \cap (V(G_d) \setminus \{w\}) \cup \{w\} = A \cap V(G_d)$.
- **Forget** Denote the forgotten vertex by w.
  We claim that $w \in A$ iff $A \cap B_c$ is a valid subset of B_c and w has some out-edges in $TC(A \cap B_c, c)$. ($\Rightarrow$) It follows immediately that $A \cap B_c$ is a valid subset. Moreover, since there is a path from w to s in A, by Lemma 3, there has to be a path that connects w to some vertex in $(A \cap B_c) \setminus \{w\}$ in $P(A \cap B_c, c)$. ($\Leftarrow$) Assume that $w \notin A$. We show that $A \cup P((A \cap B_c) \cup \{w\}, c)$ is an almost-sure reachable set that is larger than A. Indeed, we know that from every vertex in $P((A \cap B_c) \cup \{w\}, c)$ there is a path to a vertex in $(A \cap B_c) \cup \{w\}$, hence also a path to $A \cap B_c$. In addition, from every vertex in $A \cap B_c$ there is a path to s. It follows easily that condition (ii) of being an ASR set also holds, which shows the desired.
  Now, if $w \in A$, the algorithm sets $P(A \cap B_d, d) = P((A \cap B_d) \cup \{w\}, c) = P(A \cap B_c, c) = A \cap V(G_c) = A \cap V(G_d)$. On the other hand, if $w \notin A$, we have $P(A \cap B_d, d) = P(A \cap B_d, c) = P(A \cap (B_c \setminus \{w\}), c) = P(A \cap B_c, c) = A \cap V(G_c) = A \cap V(G_d)$. $\qquad\square$

By applying Lemma 6 to the root r of the tree decomposition, we obtain that $P(A \cap V(G), r) = A \cap V(G) = A$. Let us now analyze the running time.

Running time analysis. We represent $TC(\cdot, \cdot)$ with a $(k+2) \times (k+2)$ matrix. (In the original tree decomposition bags had size $k+1$, but then we added the vertex s to every bag.) The sets $P(\cdot, \cdot)$ can be represented implicitly, that is for a set $P(B, d)$ we store how it can be obtained from the respective sets contained in the children of d. This requires constant memory for each set. We iterate through $O(2^k)$ subsets of each bag. Checking whether a set is valid boils down to inspecting all edges inside a bag, which can be done in $O(k^2)$ time. The most costly operation performed for each valid subset is the computation of the transitive closure of a graph containing $O(k)$ vertices. This can be achieved in $O(k^{2.38})$ time by using fast matrix multiplication ([12], [30]).[2] Restoring the result takes time that is linear in the size of the tree decomposition. By Lemma 2, the decomposition consists of $O(n)$ nodes. Hence, the algorithm runs in $O(n \cdot 2^k \cdot k^{2.38})$ time. We obtain the following result.

Theorem 2. *Given an MDP and its tree decomposition of width k of the MDP graph, the ASR set can be computed in $O(n \cdot 2^k \cdot k^{2.38})$ time, where n is the number of states (vertices).*

3.2   MEC Decomposition

The algorithm is similar to the one for the ASR set in that it is also based on dynamic programming on a tree decomposition. Again, we assume that we have a nice tree decomposition with a bag of size 1 in the root. This time we obviously do not add the target vertex to every bag, as there is no distinguished vertex.

   As in the previous algorithm, we define a partial solution for a node d to be a subset of $V(G_d)$. This subset consists of vertices that are to form a single MEC. A partial solution P is valid, if three conditions hold.

1. For every $v \in P \cap V_P$ and every edge $vu \in E(G_d)$, we have $u \in P$.
2. For every $v \in P$ there exists a path in P from v to some vertex in $P \cap B_d$.
3. For every $v \in P$ there exists a path in P from some vertex in $P \cap B_d$ to v.

Note that the only difference from the algorithm for ASR set is that we have added the third condition. As a result we can use the dynamic programming scheme from the previous section, with only a slight change. When we perform a check that depends on the second condition (while processing a **forget** node), we need to run two symmetric checks instead of one. Let $P(B', d)$ denote the maximal partial solution for d such that $P(B', d) \cap B_d = B'$.

   We use the following two lemmas to show the correctness of the algorithm, and the proofs are analogous to lemmas in the previous section.

Lemma 7. *For each node d and each valid subset $B' \subseteq B_d$, $P(B', d)$ is a valid partial solution and $TC(B', d)$ is computed correctly.*

---

[2] In practice, a simple k^3 algorithm might a better choice than algebraic algorithms for multiplying matrices.

Lemma 8. *For every node d and MEC M such that $M \cap B_d \neq \emptyset$, we have $P(M \cap B_d, d) = M \cap V(G_d)$.*

The difference in this algorithm is in obtaining the result after the dynamic programming step is finished. First, we find the rootmost (that is, the one closest to the root) node d_1 and a vertex $v_1 \in B_{d_1}$, such that $P(\{v_1\}, d_1) \neq \perp$. In case of a tie, we can choose any node. We claim that $M_1 = P(\{v_1\}, d_1)$ is a MEC . We repeat this procedure, without taking into account vertices from M_1. This process is continued, as long as a feasible node and vertex can be found. We now show that it is correct.

Lemma 9. *For each node d and $v \in B_d$, if $P(\{v\}, d) \neq \perp$, then $P(\{v\}, d)$ is an end-component of G.*

Proof. From the definition of $P(\cdot, \cdot)$, we have that for every $u \in P(\{v\}, d) \cap V_P$ and every $ux \in E$, it holds that $x \in P(\{v\}, d)$. Moreover, from each vertex of $P(\{v\}, d)$ there is a path to v and from v there is a path to each vertex of $P(\{v\}, d)$. It follows that there is a path between any pair of vertices in $P(\{v\}, d)$, so it is a strongly connected set in G, thus also an end-component. $\square$

This implies that our algorithm finds a collection of end-components. We now show that each such end-component is a MEC. Let M be an arbitrary MEC and let d be the rootmost node, such that $B_d \cap M \neq \emptyset$. Since the tree decomposition is nice, $B_d \cap M$ contains a single vertex v. From Lemma 8 it follows that $M = P(\{v\}, d)$. It is easy to see that when the algorithm picks a first vertex from M, it picks the vertex v defined above, and thus finds a MEC M. It follows easily that every MEC is eventually found by the algorithm.

Let us now discuss the running time. As before, the dynamic programming step requires $O(n \cdot 2^k \cdot k^{2.38})$ time. Retrieving all MECs from their implicit representations requires time that is bounded by the total time of building these representations. Moreover, the process of finding rootmost nodes requires time that is linear in the size of the tree decomposition. Hence, the running time is bounded by the time of the dynamic programming and amounts to $O(n \cdot 2^k \cdot k^{2.38})$.

Theorem 3. *Given an MDP and the tree decomposition of width k of the MDP graph, the MEC decomposition can be computed in $O(n \cdot 2^k \cdot k^{2.38})$ time, where n is the number of states (vertices).*

4 Static and Decremental Algorithms for MEC Decomposition and Almost-Sure Reachability

In this section we will present the $O(m \cdot k \cdot \log n)$-time static algorithms for the MEC decomposition and the ASR set computation, and the decremental algorithms. The key would be to present two simple algorithms for the problems that we will view as decremental graph algorithmic problems (decremental scc computation for MEC decomposition, and decremental directed reachability for

ASR computation). We will then use dynamic graph algorithmic techniques to obtain the desired result. We start with the two basic algorithms. The most straightforward implementations of both these algorithms are not efficient, but we later show that they can be speeded up significantly for graphs with low treewidth using dynamic graph algorithmic techniques.

4.1   Basic Algorithms

MEC Decomposition. We first give an algorithm (formal description as Algorithm 1) for computing MEC decomposition. Here, COMPUTESCCS denotes a function, which computes an array SCC that maps the vertices v into unique identifiers $SCC[v]$ of the strongly connected components in the graph.

Algorithm 1. MEC(G)

1: $G' := G$
2: $SCC := \text{COMPUTESCCS}(G')$
3: **while** $\exists_{u \in V_P \cap V(G')} \exists_{uv \in E(G)} SCC[u] \neq SCC[v]$ **do**
4:     remove u from G'
5:     $SCC := \text{COMPUTESCCS}(G')$

Lemma 10. *Algorithm 1 is correct.*

Proof. The algorithm removes a subset of vertices of G, thus obtaining a graph G'. It follows clearly that once the algorithm terminates, the strongly connected components of G' form a MEC decomposition of G'. Moreover, they are end-components in G (note that we use $E(G)$ instead of $E(G')$ in the condition in the third line). To show that these sets form a MEC decomposition for G (i.e., they are maximal with respect to inclusion), we prove that every vertex u that is removed does not belong to any MEC of G. If u belongs to some MEC M, then v must also belong to M. But, by the definition of a strongly connected component, u is not reachable from v, so they cannot belong to the same MEC. Hence, u is not contained in any MEC.   □

Algorithm 2. ASR(G, s)

1: $G' := G$
2: $A := \text{FINDREACHABLE}(G', s)$
3: **while** $\exists_{u \in V_P \cap A} \exists_{uv \in E(G)} v \notin A$ **do**
4:     remove u from G'
5:     $A := \text{FINDREACHABLE}(G', s)$

Almost-Sure Reachability. A similar algorithm to the one above can be given for ASR. Procedure FINDREACHABLE computes the set of vertices that are connected to s with a path in G. The formal description is given as Algorithm 2.

Lemma 11. *Algorithm 2 is correct.*

Proof. The algorithm removes a subset of vertices of G, thus obtaining a graph G'. It follows clearly that once the algorithm terminates, the set of vertices from which there is a path to s is an ASR set in G' that satisfies both global and local conditions. To show that it is also an ASR set in G (i.e. it is maximal with respect to inclusion), we prove that every vertex u that is removed cannot belong to the ASR set. If u belonged to the set, then v would also belong to it. But there is no path from v to s in G, so v cannot belong to the ASR set, and neither can u. ☐

4.2   Static Algorithms for MEC and ASR

This section describes efficient implementations of algorithms from Section 4.1 that work for graphs with low treewidth.

MEC Decomposition. In order to compute MEC decomposition, we need to give an efficient implementation of Algorithm 1. This consists in maintaining the array SCC under a sequence of vertex deletions. Note that instead of removing vertices, we might well just remove all its incident edges.

   To maintain strongly connected components we use a data structure by Łącki [23]. Given the tree decomposition of a graph of width k, it can maintain the SCC array subject to edge deletions. The total running time of all delete operations is $O(m \cdot k \cdot \log n)$, and every query to the array is answered in constant time. Thus, if $\Omega(m)$ edges are deleted, the amortized time of one update is $O(k \cdot \log n)$.

   After each update, if a strongly connected component decomposes into multiple strongly connected components, some edges that used to be contained in a single strongly connected component now connect different strongly connected components. It is easy to see that it suffices to check the condition from the third line of the algorithm just for these edges. The algorithm maintaining strongly connected components can be easily extended to report the desired edges with no additional overhead. This way, we obtain an algorithm that computes the MEC decomposition in $O(m \cdot k \cdot \log n)$ total time.

Almost-Sure Reachability. We now describe an efficient implementation of Algorithm 2. This time it suffices to give an efficient algorithm that maintains the subset $A \subseteq V$ of vertices, such that for every $r \in A$ there exists an r-to-s path in G. After reversing all edges in the graph this becomes a single-source reachability problem. We show that by modifying the algorithm of Łącki [23], this can be achieved in $O(k \cdot \log n)$ amortized time. We describe the details of the algorithm below.

Decremental single-source reachability. Given a directed graph G with a designated *source* $s \in V(G)$, the goal is to maintain the set of vertices reachable from s when the edges of G are deleted. Moreover, we assume that we are given the tree decomposition of G of width k.

   The algorithm is a simplified version of the algorithm for decremental all-pairs reachability by Łącki [23]. The description in [23] contains an error in the running

time analysis of the all-pairs reachability. However, the problem disappears, if there is only a single source.

One of the ingredients of the algorithm is an algorithm for decremental single-source reachability in a DAG. The algorithm is very simple. In the beginning we delete all vertices that are not reachable from the source. Then, after an edge is deleted, we delete vertices (different from s) whose in-degree is 0, until all remaining vertices have positive in-degree. Note that deleting a vertex might decrease the in-degree of other vertices and trigger further deletions. The correctness of the algorithm follows easily. Moreover, it can be implemented, so that the total running time is linear in the number of edges of the initial graph. This is because every edge is examined when its start vertex is deleted and this means that the edge itself also gets deleted.

We can now proceed to the algorithm dealing with the general case. It maintains the subgraph of the initial graph that is reachable from s. In the description we treat G as a variable denoting this subgraph. To represent G we store its *condensation* G_c, that is the graph obtained from G by contracting all strongly connected components. It is easy to see that a condensation of an arbitrary graph is acyclic. Hence, we can use the algorithm given above to maintain it. On the other hand, to maintain the strongly connected components of G, we use the data structure by Łącki [23].

When an edge belonging to the condensation is deleted, we can simply update the condensation DAG, deleting some vertices, if necessary. All other edges are contained inside strongly connected components, so the deletion is handled by the data structure. This might cause some strongly connected component to break. In such case the data structure can report the condensation of the subgraph obtained from breaking the component with no additional overhead. This subgraph is then planted in place of the appropriate vertex in the condensation. The details are given in [23].

The total running time of processing all edge deletions is $O(m \cdot k \cdot \log n)$ and the set of reachable vertices is maintained explicitly. Also recall that for treewidth k we have $m = O(n \cdot k)$.

Theorem 4. *Given an MDP and its tree decomposition of width k, the MEC decomposition and the ASR set can be computed in time $O(m \cdot k \cdot \log n)$, where n is the number of states (vertices) and m is the number of edges.*

4.3 Decremental Algorithms

Both algorithms that we have described can be easily extended to decremental algorithms that support edge deletions. However, only deleting edges $uv \in E$ such that $u \in V_1$ is allowed. This assures that the ASR set can only shrink and that every end-component in the MEC decomposition is a subset of a some end-component from the graph before the deletion.

Almost-Sure Reachability. The algorithm first runs Algorithm 2 during the initialization phase and computes the initial set A. The set A is maintained by a single-source decremental reachability algorithm. The very same high-level

algorithm can be used to update the set A after an edge is deleted. We run this algorithm whenever an edge is deleted. Observe that if we detect that A shrinks, i.e. a subset $U \subseteq A$ of vertices is removed from A, we need to check the condition in the third line only for edges that are entering this set. Thus, each edge is inspected at most once during the entire course of the algorithm. Hence, the dominating operation is the running time of the decremental single-source reachability algorithm, which requires $O(m \cdot k \cdot \log n)$ time over all deletions or $O(k \cdot \log n)$ amortized time for a single deletion, if $\Omega(m)$ edges are deleted. The proof of correctness is analogous to the one in Lemma 11.

MEC Decomposition. We use the same idea as for the decremental algorithm for the ASR set. In this case Algorithm 1 can be used both for the initialization and after an edge is deleted. By maintaining the array SCC with a data structure for decremental SCC maintenance, we get that the amortized time of processing a single update is $O(k \cdot \log n)$.

Theorem 5. *Given an MDP and its tree decomposition of width k, the MEC decomposition and the ASR set can be computed under the deletion of $\Omega(m)$ player-1 edges, in amortized time $O(k \cdot \log n)$ per edge deletion, where n is the number of states (vertices) and m is the number of edges.*

Concluding remarks. In this work, we presented faster static and decremental algorithms for two core algorithmic problems for MDPs when the treewidth is low. An interesting question for future work is whether the algorithms can be extended to MDPs with low DAG-width (as done for parity games in [1]).

References

1. Berwanger, D., Dawar, A., Hunter, P., Kreutzer, S., Obdrzálek, J.: The DAG-width of directed graphs. J. Comb. Theory, Ser. B 102(4), 900–923 (2012)
2. Bianco, A., de Alfaro, L.: Model checking of probabilistic and nondeterministic systems. In: Thiagarajan, P.S. (ed.) FSTTCS 1995. LNCS, vol. 1026, pp. 499–513. Springer, Heidelberg (1995)
3. Bodlaender, H.L.: A linear-time algorithm for finding tree-decompositions of small treewidth. SIAM J. Comput. 25(6), 1305–1317 (1996)
4. Bodlaender, H.L.: Treewidth: Algorithmic techniques and results. In: Privara, I., Ružička, P. (eds.) MFCS 1997. LNCS, vol. 1295, pp. 19–36. Springer, Heidelberg (1997)
5. Brázdil, T., Brozek, V., Chatterjee, K., Forejt, V., Kucera, A.: Two views on multiple mean-payoff objectives in Markov decision processes. In: LICS, pp. 33–42 (2011)
6. Chatterjee, K., Henzinger, M.: Faster and dynamic algorithms for maximal end-component decomposition and related graph problems in probabilistic verification. In: SODA, pp. 1318–1336 (2011)

7. Chatterjee, K., Henzinger, M.: An O(n^2) time algorithm for alternating büchi games. In: SODA, pp. 1386–1399 (2012)
8. Chatterjee, K., Henzinger, T.A.: Probabilistic systems with limsup and liminf objectives. In: Archibald, M., Brattka, V., Goranko, V., Löwe, B. (eds.) ILC 2007. LNCS, vol. 5489, pp. 32–45. Springer, Heidelberg (2009)
9. Chatterjee, K., Henzinger, T.A., Jobstmann, B., Singh, R.: Measuring and synthesizing systems in probabilistic environments. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 380–395. Springer, Heidelberg (2010)
10. Chatterjee, K., Jurdziński, M., Henzinger, T.: Quantitative stochastic parity games. In: SODA 2004, pp. 121–130. SIAM (2004)
11. Chatterjee, K., Łącki, J.: Faster algorithms for Markov decision processes with low treewidth CoRR abs/1304.0084 (2013), http://arxiv.org/abs/1304.0084
12. Coppersmith, D., Winograd, S.: Matrix multiplication via arithmetic progressions. J. Symb. Comput. 9(3), 251–280 (1990)
13. Courcoubetis, C., Yannakakis, M.: The complexity of probabilistic verification. Journal of the ACM 42(4), 857–907 (1995)
14. de Alfaro, L.: Formal Verification of Probabilistic Systems. PhD thesis, Stanford University (1997)
15. de Alfaro, L., Faella, M., Majumdar, R., Raman, V.: Code-aware resource management. In: EMSOFT 2005. ACM (2005)
16. Fearnley, J., Lachish, O.: Parity games on graphs with medium tree-width. In: Murlak, F., Sankowski, P. (eds.) MFCS 2011. LNCS, vol. 6907, pp. 303–314. Springer, Heidelberg (2011)
17. Fearnley, J., Schewe, S.: Time and parallelizability results for parity games with bounded treewidth. In: Czumaj, A., Mehlhorn, K., Pitts, A., Wattenhofer, R. (eds.) ICALP 2012, Part II. LNCS, vol. 7392, pp. 189–200. Springer, Heidelberg (2012)
18. Filar, J., Vrieze, K.: Competitive Markov Decision Processes. Springer (1997)
19. Hinton, A., Kwiatkowska, M., Norman, G., Parker, D.: PRISM: A tool for automatic verification of probabilistic systems. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 441–444. Springer, Heidelberg (2006)
20. Howard, H.: Dynamic Programming and Markov Processes. MIT Press (1960)
21. Kloks, T.: Treewidth, Computations and Approximations. LNCS, vol. 842. Springer (1994)
22. Kwiatkowska, M., Norman, G., Parker, D.: Verifying randomized distributed algorithms with prism. In: WAVE 2000 (2000)
23. Łącki, J.: Improved deterministic algorithms for decremental transitive closure and strongly connected components. In: SODA, pp. 1438–1445 (2011)
24. Obdržálek, J.: Fast mu-calculus model checking when tree-width is bounded. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 80–92. Springer, Heidelberg (2003)
25. Pogosyants, A., Segala, R., Lynch, N.: Verification of the randomized consensus algorithm of Aspnes and Herlihy: a case study. Dist. Comp. 13(3), 155–186 (2000)
26. Robertson, N., Seymour, P.D.: Graph minors. iii. planar tree-width. J. Comb. Theory, Ser. B 36(1), 49–64 (1984)
27. Stoelinga, M.: Fun with FireWire: Experiments with verifying the IEEE1394 root contention protocol. In: Formal Aspects of Computing (2002)
28. Thomas, W.: Languages, automata, and logic. In: Handbook of Formal Languages, vol. 3, ch. 7, pp. 389–455. Springer, Heidelberg (1997)
29. Thorup, M.: All structured programs have small tree-width and good register allocation. Inf. Comput. 142(2), 159–181 (1998)
30. Williams, V.V.: Multiplying matrices faster than coppersmith-winograd. In: STOC, pp. 887–898 (2012)