Faster and Dynamic Algorithms For Maximal End-Component Decomposition And Related Graph Problems In Probabilistic Verification

Krishnendu Chatterjee* Monika Henzinger†

Abstract

We present faster and dynamic algorithms for the following problems arising in probabilistic verification: Computation of the maximal end-component (mec) decomposition of Markov decision processes (MDPs), and of the almost sure winning set for reachability and parity objectives in MDPs. We achieve the following running time for static algorithms in MDPs with graphs of n vertices and m edges: (1) $O(m \cdot \min\{\sqrt{m}, n^{2/3}\})$ for the mec decomposition, improving the longstanding $O(m \cdot n)$ bound; (2) $O(m \cdot n^{2/3})$ for reachability objectives, improving the previous $O(m \cdot \sqrt{m})$ bound for $m > n^{4/3}$; and (3) $O(m \cdot \min\{\sqrt{m}, n^{2/3}\} \cdot \log(d))$ for parity objectives with d priorities, improving the previous $O(m \cdot \sqrt{m} \cdot d)$ bound. We also give incremental and decremental algorithms in linear time for mec decomposition and reachability objectives and $O(m \cdot \log d)$ time for parity objectives.

Keywords. *(1) Probabilistic verification; (2) Parity objectives; (3) Graph decomposition; (4) Dynamic graph algorithms; (5) Maximal end-components.*

1 Introduction

We study algorithmic problems on graphs and a generalization of them, called *Markov decision processes (MDPs)*, that arise in probabilistic verification. The input to a probabilistic verification problem is a system that exhibits probabilistic behavior and a specification (set of desired behaviors), and the algorithmic problem is to answer whether the system satisfies the specification [6].

We first present a graph problem that lies at the core of many algorithms in probabilistic verification. Given a directed graph $G = (V, E)$ with a finite set V of vertices, a set $E \subseteq V \times V$ of directed edges, and a partition (V_1, V_P) of V, an *end-component* $U \subseteq V$ is a set of vertices such that (a) the graph $(U, E \cap U \times U)$ is strongly connected; (b) for all $u \in U \cap V_P$ and all $(u, v) \in E$ we have $v \in U$; and (c) either $|U| \geq 2$, or $U = \{v\}$ and there is a self-loop at v (i.e., $(v, v) \in E$). Note that if U_1 and U_2 are end-components with $U_1 \cap U_2 \neq \emptyset$, then $U_1 \cup U_2$ is an end-component. A *maximal end-component (mec)* is an end-component that is maximal under set inclusion. Every vertex of V belongs to *at most* one maximal 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* maximal end-component. Maximal end-components generalize strongly connected components[1] for directed graphs ($V_P = \emptyset$) and closed recurrent sets for Markov chains ($V_1 = \emptyset$). We present faster and dynamic algorithms for computing the maximal end-component decomposition.

In probabilistic verification, systems are frequently modeled as a generalization of graphs, called *Markov decision processes (MDPs)*. The generalization is needed to model two different kind of "behaviors" at vertices [10]. More specifically there are two types of vertices, namely the vertices in V_1, that are regular vertices in graph algorithmic setting, i.e., where the algorithm can choose which out-edge to follow, and the vertices in V_P, that are vertices where the out-edge is chosed randomly according to a given distribution δ. The former vertices are called *player-1 vertices*, the latter are called *random vertices*, and the probability distribution is called *probabilistic transition function.* The probabilistic transition function is a distribution over all out-neighbors of a vertex[2] and can be different for different random vertices. More formally, a *Markov decision process (MDP)* $\mathsf{P} = ((V, E), (V_1, V_P), \delta)$ consists of a directed *MDP graph* (V, E), a partition (V_1, V_P) of the *finite* set V of vertices, and a probabilistic transition function δ: $V_P \to \mathcal{D}(V)$, where $\mathcal{D}(V)$ denotes the set of probability distributions over the vertex set V. Note that (a) a directed graph is a special case of an MDP with $V_P = \emptyset$ and (b) a Markov chain is a special case of an MDP with $V_1 = \emptyset$. MDPs are used to model and solve control problems in systems such as stochastic systems [9], concurrent probabilistic systems [6], probabilistic systems operating in open environments [17],

*IST Austria (Institute of Science and Technology Austria) Email: krish.chat@ist.ac.at

†Fakultät für Informatik, Universität Wien, Austria. Email: monika.henzinger@univie.ac.at

[1] In this paper we use *scc* or *strongly connected component* for a *maximal strongly connected component.*

[2] More formally we require that for all $u \in V_P$ and all $v \in V$ we have $(u, v) \in E$ iff $\delta(u)(v) > 0$.

and under-specified probabilistic systems [1].

There are two types of problems, called *objectives*, on MDPs that this paper addresses: (1) *Reachability objectives* and (2) *parity objectives*. Formally, an *objective* ψ is a (measurable) subset of *infinite* walks in the MDP graph. Given an objective ψ, *qualitative analysis* asks for the computation of the *almost-sure winning set* for ψ (denoted by $\langle\langle 1 \rangle\rangle_{almost}(\psi)$), which is the set of vertices A such that player 1 can ensure that a walk started at A belongs to ψ with probability converging to 1 as the length of the walk goes to ∞. We say that ψ can be *ensured with probability 1*. Given an MDP and a set $T \subseteq V$ of target vertices a *Reachability objective*, denoted by Reach(T), consists of all infinite walks in the MDP graph that visit a vertex in T at least once. Given an MDP and a priority function $p : V \to \{ 0, 1, \ldots, d \}$ that maps vertices to integer priorities a *Parity objectives*, denoted by Parity(p), consists of all infinite walks in the MDP graph for which the minimum priority vertex that is visited infinitely often on the walk is even.

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. Hence there is a clear motivation to obtain dynamic algorithms for qualitative analysis and mec decomposition for MDPs that achieve a better running time than recomputation from scratch when edges (u, v) with $u \in V_1$ are inserted or deleted.

Applications. Parity objectives are a canonical way to define desired behaviors of systems, such as safety, liveness, fairness, etc [22]. Thus MDPs with parity objectives provide the theoretical framework to study problems such as the verification and the control of stochastic systems. Furthermore, qualitative analysis of MDPs is important as there are many applications where we need to know whether the correct behavior arises with probability 1. For instance, when analyzing a randomized embedded scheduler, we are interested in whether every thread progresses with probability 1 [8]. And even in settings where it suffices to satisfy certain specifications with probability $p < 1$, the correct choice of p is a challenging problem, due to the simplifications introduced during modeling. For example, in the analysis of randomized distributed algorithms it is quite common to require correctness with probability 1 (see, e.g., [15, 14, 19]). Furthermore qualitative analysis is robust to numerical perturbations and modeling errors in the transition probabilities, and consequently the algorithms for qualitative analysis are combinatorial. Finally, for MDPs with parity objectives, the best known algorithms and all algorithms used in practice first perform the qualitative analysis, and then perform quantitative analysis on the result of qualitative analysis. In short, qualitative analysis for MDPs with parity objectives is one of the most fundamental and core problem in verification of probabilistic systems, and as we show here its algorithms crucially depend on the maximal end-component problem.

In addition, several algorithms for *quantitative* analysis of MDPs with quantitative objectives such as lim sup and lim inf objectives [2], and combination of mean-payoff and parity objectives [3], rely on the maximal end-component decomposition problem.

1.1 Our contributions. In this work, we use techniques from dynamic graph algorithms to present novel algorithms for mec decomposition, and qualitative analysis of reachability and parity objectives. We present both improved (static) algorithms as well as the first incremental and decremental algorithms for maintaining the mec decomposition for MDPs and qualitative analysis of MDPs with reachability and parity objectives as the MDP is changed. The details of our results are as follows, where $n = |V|$ and $m = |E|$ (see also Table 1).

1. We present a $O(m \cdot \min\{\sqrt{m}, n^{2/3}\})$-time algorithm for the mec decomposition of an MDP, improving the $O(m \cdot n)$ bound from 1995 [6, 7]. This is the first algorithm that breaks the $O(m \cdot n)$ barrier for the mec decomposition problem.
2. We give a $O(m \cdot n^{2/3})$-time algorithm for the qualitative analysis for reachability objectives, improving the previously known $O(m \cdot \sqrt{m})$ bound [4] for $m > n^{4/3}$.
3. We present a $O(m \cdot \min\{ \sqrt{m}, n^{2/3} \} \cdot \log(d))$-time algorithm for the qualitative analysis for parity objectives with d priorities, improving the previously known $O(m \cdot \sqrt{m} \cdot d)$ bound [5].

In addition we give the first incremental and decremental algorithms for these problems when an edge (u, v) with $u \in V_1$ is inserted or deleted.

1. We show how to maintain the mec decomposition after an edge insertion or deletion in time linear in the size of the graph. For the decremental case the running time bound is amortized, whereas for the incremental case we give a worst case bound. Note that the problem of maintaining a mec decomposition generalizes the problem of maintaining a scc decomposition, and our results match the best known bounds for incremental and decremental scc decomposition.
2. We present amortized $O(m)$-time algorithms for incremental and decremental qualitative analysis for reachability objectives. For the decremental case we present a reduction to the decremental reachability in directed graphs. The reduction

	Previous Algorithm	Our Algorithm	Incremental	Decremental
Max. End-component	$O(m \cdot n)$	$\mathbf{O(m \cdot \min\{ \sqrt{m}, n^{2/3} \})}$	$\mathbf{O(m)}$	$\mathbf{O(m)}$
Qual. Reachability	$O(m \cdot \sqrt{m})$	$\mathbf{O(m \cdot \min\{ \sqrt{m}, n^{2/3} \})}$	$\mathbf{O(m)}$	$\mathbf{O(m)}$
Qual. Parity	$O(m \cdot \sqrt{m} \cdot d)$	$\mathbf{O(m \cdot \min\{ \sqrt{m}, n^{2/3} \} \cdot \log d)}$	$\mathbf{O(m \cdot \log d)}$	$\mathbf{O(m \cdot \log d)}$

Table 1: Running time analysis: Our results are in bold font.

has two implications: (a) From the results of [16] we obtain a randomized decremental algorithm whose expected amortized running time is $O(n)$. (b) Any improvement in decremental reachability for directed graphs imply the same improvement for the more general problem of decremental almost-sure reachability in MDPs.

3. We give amortized $O(m \cdot \log(d))$-time algorithms for incremental and decremental qualitative analysis for parity objectives with d priorities.

Our main technical contributions are as follows. (1) A *bottom scc* C is a scc that has no edge leaving C. Our first algorithm for mec decomposition repeatedly finds bottom scc's using the scc decomposition algorithm of [20] and we show that by lock-step search from a specially chosen set of start vertices we can achieve a $O(m \cdot \sqrt{m})$ bound. (2) Our second algorithm for mec decomposition reduces the number of these lock-step searches. To achieve this the algorithm makes reachability queries in a graph that is repeatedly modified. However, using the fastest known dynamic algorithms for reachability in directed graphs would lead to a running time of $\Omega(m \cdot n)$. Instead we store "a compressed version" of the graph in a dynamic tree data structure [18], which we update dynamically, and use it to answer the necessary reachability queries. (3) We show how to modify the algorithm for mec decomposition to solve the almost-sure reachability problem. (4) The incremental algorithm for the almost-sure reachability objectives is obtained proving novel graph theoretic properties of the almost-sure winning set and the incremental mec decomposition. The decremental almost-sure reachability is obtained through a reduction to decremental reachability in directed graphs. Thus, any improvement in the running time for the latter problem will also improve the decremental almost-sure reachability problem. (5) The algorithm for almost-sure parity objectives is obtained using the mec decomposition, almost-sure reachability, and a hierarchical clustering technique of [21]. Without the hierarchical clustering technique, the naive algorithm requires time $O(d)$ times the mec decomposition algorithm followed by one call to the almost-sure reachability algorithm. We show that with the hierarchical clustering technique the almost-sure winning set for parity objective can be computed in time $O(\log d) \cdot T_M(m,n) + T_R(m,n)$, where $T_M(m,n)$ is the time complexity of computing the mec decomposition, and $T_R(m,n)$ is the time complexity of computing the almost-sure winning set for reachability objectives.

In the rest of the paper we use the following notations. If $(v,w) \in E$, then we call w a *successor* of v; if $v \in V_P$, then we call an edge (v,w) a *random edge*, and if $v \in V_1$, we call it a *player-1 edge*. We denote by $E(v) = \{ w \mid (v,w) \in E \}$ the set of successors of v. Furthermore a *trivial* end-component consists of a single vertex.

2 Algorithms for Maximal End-components Decomposition

In this section we present two improved static algorithms for computing the maximal end-component (mec) decomposition of a graph (V,E) with partition (V_1,V_P) of V, and the first incremental and decremental algorithms to maintain the mec decomposition. By abuse of notation we use mec decomposition of an MDP to mean the mec decomposition of the MDP graph with partition (V_1,V_P). For technical convenience we make two assumptions about the MDP graph: (1) Every vertex v has at least one out-going edge, i.e. $E(v) \neq \emptyset$, because a vertex without out-going edges does not belong to any end-component. (2) We will consider MDPs such that random vertices do not have self-loops. Note that a vertex with a self-loop that does not belong to any other mec forms its own trivial mec. Thus, if a MDP graph with self-loops at random vertices is given, its mec decomposition can be computed as follows: First remove all self-loops at random vertices and compute the mec decomposition of the resulting graph. For every random vertex with a self-loop that does not belong to any other mec, form a trivial mec consisting only of the vertex. We could proceed in the same way with self-loops of vertices $v \in V_1$, but we need to allow self-loops of player-1 vertices for technical reasons in the incremental maintenance of the mec decomposition.

2.1 Algorithms for Computing Maximal End-components.

We first define attractors, closed sets and prove two lemmata about them. Then we present the classic algorithm and our improved algorithms.

Random and player-1 attractor. Given an MDP P, let $U \subseteq V$ be a subset of vertices. The *random attractor* $Attr_R(U)$ is defined inductively as follows: $U_0 = U$, and for $i \geq 0$, let $U_{i+1} = U_i \cup \{ v \in V_P \mid E(v) \cap U_i \neq \emptyset \} \cup \{ v \in V_1 \mid E(v) \subseteq U_i \}$. In other words, U_{i+1} consists of (a) vertices in U_i, (b) random vertices that have at least one edge to U_i, and (c) player-1 vertices such that all their successors are in U_i. Then $Attr_R(U) = \bigcup_{i \geq 0} U_i$. The definition of *player-1 attractor* $Attr_1(U)$ is obtained by exchanging the role of random vertices and player-1 vertices in the above definition. A (random or player-1) attractor A can be computed in time $O(\sum_{v \in A} \text{indeg}(v))$ [11], where $\text{indeg}(v)$ is the number of incoming edges of v.

Closed set. A set $X \subseteq V$ of vertices is a *closed set* if for all random edges (u, v) with $u \in X$ we have $v \in X$. Thus a set U is a mec if U is strongly connected and closed.

Property of attractors. The first lemma below establishes that the random attractor of a mec and the random attractor of certain vertices of an scc do not belong to any mec and that it, thus, can be removed without affecting the mec decomposition of the remaining graph. Hence, the lemma can be used to identify vertices that do not belong to *any* mec. The second lemma below shows under which condition an scc is an mec. Thus, it can be used to identify vertices that *form* a mec.

LEMMA 2.1. *Let P be an MDP, and let (V, E) be the MDP graph.*

1. *Let C be a scc in (V, E). Let $U = \{ v \in C \cap V_P \mid E(v) \cap (V \setminus C) \neq \emptyset \}$ be the random vertices in C with edges out of C. Let $Z = Attr_R(U) \cap C$. Then for all non-trivial mec's X in P we have $Z \cap X = \emptyset$ and for any edge (u, v) with $u \in X$ and $v \in Z$, u must belong to V_1.*
2. *Let C be a mec in P. Let $Z = Attr_R(C) \setminus C$. Then for all non-trivial mec's X with $X \neq C$ in P we have $Z \cap X = \emptyset$ and for any edge (u, v) with $u \in X$ and $v \in Z$, u must belong to V_1.*

Proof. We present both parts of the proof.

Part 1. Assume by contradiction that there is a non-trivial mec X such that $X \cap Z \neq \emptyset$. Since (a) $X \cap Z \subseteq X \cap C \neq \emptyset$, (b) X must be strongly connected, and (c) C is a scc; it follows that $X \subseteq C$. As X must be closed, and random vertices in U have edges out of C (hence also out of X), we must have $X \cap U = \emptyset$. Hence (a) X is closed (for all $u \in X \cap V_P$ we have $E(u) \subseteq X$) and (b) X does not contain any vertex in U ($X \cap U = \emptyset$). We use these two properties to show by induction that $X \cap Attr_R(U) = X \cap Z = \emptyset$. We use the following inductive claim: For all $i \geq 0$ we have $U_i \cap X = \emptyset$. The base case $i = 0$ follows as $U_0 = U$ and by (b) we have $X \cap U_0 = \emptyset$. For $i > 0$ we assume that $X \cap U_i = \emptyset$, and show that $X \cap U_{i+1} = \emptyset$. We have $U_{i+1} = U_i \cup \{ v \in V_P \mid E(v) \cap U_i \neq \emptyset \} \cup \{ v \in V_1 \mid E(v) \subseteq U_i \}$. Consider a vertex $u \in X$: (a) if $u \in V_1$, then since $|X| \geq 2$ and X is strongly connected, there exists a $v \in X$ with $(u, v) \in E$, and since $X \cap U_i = \emptyset$ it follows that $E(u)$ is not a subset of U_i and hence $u \notin U_{i+1}$; (b) if $u \in V_P$, then since X is closed (hence $E(u) \subseteq X$) and $X \cap U_i = \emptyset$, we have $E(u) \cap U_i = \emptyset$, and hence $u \notin U_{i+1}$. It follows that for all $i \geq 0$ we have $X \cap U_{i+1} = \emptyset$, and thus $X \cap Attr_R(U) = X \cap Z = \emptyset$. Hence we have a contradiction. For a vertex $u \in X$, if there is an edge (u, v) with $v \in Z$, then $u \notin Z$. Thus u cannot belong to V_P as vertices of V_P are not allowed to have outgoing edges leaving their mec.

Part 2. Assume by contradiction that there is a non-trivial mec X such that $Z \cap X \neq \emptyset$. Since X is a mec, X must be closed. Since X is closed and X does not contain any vertex in C, it follows from the inductive proof of the previous case that $X \cap Attr_R(C) = X \cap Z = \emptyset$, and hence we have a contradiction. As above for an edge (u, v) with $u \in X$ and $v \in Z$, we must have $u \in V_1$.

The desired result follows. ∎

LEMMA 2.2. *Let P be an MDP, and let (V, E) be the MDP graph. Let C be a scc in (V, E) such that for all $v \in C \cap V_P$ we have $E(v) \subseteq C$. Then C is a mec.*

Proof. It follows that C is closed, and since C is a scc it follows that C is a mec. ∎

It is an easy corollary that every bottom scc is a mec.

Previous algorithm for maximal end-component decomposition. The previous algorithm to compute an mec decomposition of an MDP is as follows: (a) Given an MDP P consider the MDP graph (V, E), and compute the scc decomposition of (V, E) in $O(m)$ time. (b) For every scc C with random edges leaving C, let U be the set of random vertices in C with edges out of C. Remove $Attr_R(U) \cap C$ from the graph (by Lemma 2.1 these vertices belong to no mec). (c) All scc's C that have no random edges going out of C are mec's (by Lemma 2.2). Note that there is always at least one such scc since every graph has a bottom scc. We remove $Attr_R(C)$ and recursively compute mec in the smaller sub-MDP. Each iteration takes $O(m)$ time and removes at least one vertex. Thus the running time of the algorithm is $O(m \cdot n)$. We will refer this algorithm as the *simple static* algorithm for mec decomposition.

Improved algorithm MAXECDE1**.** Our first improved algorithm for mec decomposition is obtained by

combining the simple static algorithm for mec decomposition along with a *lock-step (or dovetail)* linear-time depth-first search (DFS) to find a bottom scc. Specifically, each of the searches that is executed uses the dfs-based scc algorithm of Tarjan [20], which has the property that if it started at a vertex in a bottom scc it finds this bottom scc and stops in time linear in the number of edges in the scc. In this paper we will use the term *lock-step search* with the following meaning: for k searches, in one step the lock-step search each search can process exactly one edge. Thus it is ensured that in l lock-steps each search explores exactly l edges.[3] The algorithm iteratively removes vertices from the graph for which either the mec was found or for which it was identified that they belong to no mec, until all vertices are removed. At iteration i, we denote the remaining subgraph as (V_i, E_i), where V_i is the set of remaining vertices and E_i is the set of remaining edges. The algorithm considers two cases: (a) Case 1 is similar to the simple static algorithm, and (b) Case 2 is the lock-step exploration of a bottom scc. The algorithm maintains the set L_{i+1} of vertices that were removed from the graph since the last iteration of Case 1, and the set J_{i+1} of vertices that lost an edge to vertices removed from the graph since last iteration of Case 1. Initially, $(V_0, E_0) := (V, E)$, $L_0 := J_0 := \emptyset$, and $i := 0$. We describe our algorithm.

1. *Case 1.* If $((|J_i| \geq \sqrt{m})$ or $i = 0)$, then
   (a) Compute the scc decomposition of the current MDP graph (V_i, E_i).
   (b) For all scc's C that have a random edge leaving out of C, let U be the subset of random vertices in C that have an edge leaving C. The set $Attr_R(U) \cap C$ is removed from the graph.
   (c) For all scc's C that do not have a random edge leaving C, the scc C is identified as a mec and $Attr_R(C)$ is removed from the graph.
   (d) The set L_{i+1} is the set of vertices removed from the graph in this iteration and J_{i+1} be the set of vertices in the remaining graph with an edge to L_{i+1}.
   (e) $i := i + 1$; if $V_i = \emptyset$, then stop the algorithm, else go to the next iteration.
2. *Case 2.* Else $(|J_i| \leq \sqrt{m})$, then
   (a) We do a lock-step search using the scc algorithm of Tarjan [20] from every vertex v in J_i to obtain a bottom scc that contains v. Let C be the first bottom scc discovered in the lock-step search. The lock-step search ends when the first bottom scc C is discovered.
   (b) The bottom scc C is a mec and we remove $Attr_R(C)$ from the graph. Let the set L_{i+1} be the set of vertices removed from the graph since the last iteration of Case 1 (i.e., $L_{i+1} := L_i \cup Attr_R(C)$, where C is the bottom scc removed in step 2(b) of this iteration) and let J_{i+1} be the set of vertices in the remaining graph with an edge to L_{i+1}, i.e., $J_{i+1} := (J_i \setminus Attr_R(C)) \cup Q_i$, where Q_i is the subset of vertices of V_i with an edge to $Attr_R(C)$. Thus the set J_{i+1} is the set of vertices in the graph that lost an edge to the vertices removed since the last iteration that executed Case 1.
   (c) $i := i + 1$; if $V_i = \emptyset$, then stop the algorithm, else go to the next iteration.

Correctness and running time analysis. We now present the correctness argument and running time analysis.

LEMMA 2.3. *The algorithm* MAXECDE1 *correctly computes the maximal end-component decomposition of an MDP* P.

Proof. The algorithm repeatedly removes bottom sccs and their random attractors. Since every bottom scc is a mec (by Lemma 2.2) and in each step a random attractor is removed (hence in the current graph all the outgoing edges for random vertices are preserved), the correctness of the algorithm follows from Lemma 2.1 and Lemma 2.2. ∎

LEMMA 2.4. *For every iteration i and for every bottom scc C of the graph (V_i, E_i) there is a vertex in J_i that belongs to C.*

Proof. We consider an iteration i of the algorithm. We show that in the graph (V_i, E_i) the intersection of J_i and each bottom scc of (V_i, E_i) is non-empty. The proof of the claim is as follows: consider a bottom scc C in the graph (V_i, E_i). Then there is no edge that leaves C in the graph (V_i, E_i). Let $j < i$ be the last iteration before iteration i such that Case 1 was executed in iteration j (and in all iterations between j and i Case 2 is executed). If $C \cap J_i$ is empty, then it follows that none of the vertices in C has lost an edge since and including iteration j. Since C is a bottom scc in (V_i, E_i), C must also have been a bottom scc in (V_j, E_j) and, thus, it must have been discovered as a mec in step 1(b) of iteration j. Hence we have a contradiction. It follows that we always have a witness vertex in J_i that is in a bottom scc. ∎

[3] In algorithm MAXECDE2 processing an edge will require operations on a dynamic tree and hence may require $O(\log n)$ time, whereas other edges may be processed in constant time. The lock-step search will still allow each search to process exactly one edge at every step, even if this might take a different amount of time.

An easy consequence of this lemma is that J_i always contains a vertex in a mec in the graph (V_i, E_i).

LEMMA 2.5. *The running time of algorithm* MAXECDE1 *on an MDP* P *with* m *edges is* $O(m \cdot \sqrt{m})$.

Proof. We now analyze the running time of MAXECDE1. The total work of the algorithm when Case 1 is executed over all iterations is at most $O(m \cdot \sqrt{m})$: this follows because between two iterations of Case 1 at least $O(\sqrt{m})$ edges must have been removed from the graph (since $|J_i| \geq \sqrt{m}$ everytime Case 1 is executed other than the case when $i = 0$), and each iteration can be achieved in $O(m)$ time (since scc decomposition can be computed in $O(m)$ time) [20]. We now show that the total work of the algorithm when Case 2 is executed over all iterations is at most $O(m \cdot \sqrt{m})$. The argument is as follows: consider an iteration i such that Case 2 is executed. Let C be the bottom scc discovered in iteration i while executing Case 2. Let $E(C) = \bigcup_{v \in C} E(v)$. The algorithm of [20] for scc decomposition ensures that if the starting vertex is in the bottom scc, then the bottom scc is identified in time proportional to the number of edges of the bottom scc. The lock-step search ensures that the edges explored in this iteration is at most $O(|J_i| \cdot |E(C)|) \leq O(\sqrt{m} \times |E(C)|)$. Since C is identified as a mec and removed from the graph we charge the work of $O(\sqrt{m} \cdot |E(C)|)$ to edges in $E(C)$, charging work $O(\sqrt{m})$ to each edge. Since there are at most m edges, the total charge of the work over all iterations when Case 2 is executed is at most $O(m \cdot \sqrt{m})$. ∎

THEOREM 2.1. *Given an MDP* P*, the algorithm* MAXECDE1 *computes the mec decomposition of* P *in time* $O(m \cdot \sqrt{m})$.

Second improved algorithm MAXECDE2**.** Our second algorithm modifies MAXECDE1 by reducing the number of DFS searches that are executed in lock-step. It exploits more properties of the ome-pass scc algorithm SCC based on DFS from [20]than MAXECDE1. In SCC a vertex y *is being visited by a DFS* if the vertex is popped off the DFS stack and $dfs(y)$ has been called, but not yet completed. The vertex y *has been visited* when $dfs(y)$ has completed. We call the most recent vertex for which $dfs(x)$ was started the *current* vertex of the DFS. A scc has been *detected* after the visit of all vertices in the scc has completed. If SCC is started on a vertex in a bottom scc C, then it detects C in time linear in its size. It maintains the following invariant: *(I) Every vertex that has been or is still visited by the DFS but whose scc has not yet been detected has a (directed) path to the current vertex of the DFS*[4].

We call a vertex x from which a lock-step search is started a *start vertex* and the corresponding DFS $dfs(x)$. The algorithm MAXECDE1 determines in each iteration a set J_i of *active* vertices and uses them as start-vertices for the next iteration of Case 2. The algorithm MAXECDE2 modifies Case 2 as follows: In Step 1 SCC is started from *all* active vertices. As we show if $dfs(x)$ visits a vertex u that was already visited by $dfs(y)$ with $y \neq x$, then $dfs(x)$ can be stopped under certain conditions without affecting the correctness of the algorithm. Thus during Step 1 some of the active vertices become *passive* and their DFS searches are stopped. Let k be a suitable parameter. Step 1 executes k lock-steps for all searches, i.e. each $dfs(x)$ started in Step 1 runs for k steps or it is stopped before it completed k steps (because it visited a vertex that was already visited by another DFS). Thus we can bound the time spent in Step 1 by $O(km)$. In Step 2 the remaining DFS searches that were still running at the end of Step 1 are allowed to run for completion. We prove that there are at most $nO(/\sqrt{k})$ such DFS searches. Thus the total time spent in Step 2 can be bound by $O(mn/\sqrt{k})$. Setting $k = n^{2/3}$ gives a bound of $O(m \cdot n^{2/3})$ for both steps.

We first introduce our notation: If $dfs(x)$ visits in Step 1 a new vertex it checks first whether it has to stop. If not, then it *labels* the vertex it visits by x. These labels are necessary to know whether $dfs(x)$ has to stop or not. If $dfs(x)$ stops when visiting vertex u, then x becomes *passive* and u becomes the *stop vertex* $s(x)$ of x. Note that for a non-passive vertex y, $s(y)$ is not defined.

If a bottom scc is detected in Step 1, it is not removed in Step 1, but its vertices are marked as *special* and removed at the beginning of Step 2. This might make some more vertices active. As long as there are at most $2n/\sqrt{k}$ remaining active vertices, Step 2 repeatedly performs lock-step searches and removes the bottom sccs detected by them and their random attractors.

We use the following data structure: (1) Every vertex u keeps bits indicating whether the vertex is active or passive or neither, and special or not. (2) Every vertex u keeps a list of all passive vertices x such that $u = s(x)$. (3) During Step 1 we keep a list of vertices labeled in Step 1 to unlabel them in Step 2. (4) During Step 1 we maintain "condensed reachability information" by keeping a rooted forest of all vertices in V in a dynamic tree data structure W [18]. At the beginning of Step 1 every vertex in W is a 1-vertex tree in W. At the beginning of Step 2, W is deleted. During

[4]Note that the vertices whose visit has not yet ended are still on the DFS stack and thus this condition is trivially fulfilled.

Step 1 W fulfils the following three invariants:

(DT1) If a vertex is active or special, then it is the root of a tree of W. Every non-root vertex in W is a passive and non-special vertex.

(DT2) If x is a child of an active vertex y, then $s(x)$ is a non-special vertex labeled by y. If x is a child of a passive, non-special vertex y, then $s(x)$ is a non-special vertex, there exists a path from $s(x)$ to $s(y)$ in the graph and y can reach $s(x)$. If x is a child of a special vertex y in W, then $y = s(x)$.

(DT3) If a vertex x is passive, then the root of the tree containing x is a vertex y such that either (a) y is special and there exists a path from $s(x)$ to y or (b) y is an active vertex such that there exists a path from $s(x)$ to a non-special vertex labeled by y.

Given a vertex x a *witness query* in W returns the root of the tree containing x. Since W is stored in a dynamic tree data structure, every witness query takes time $O(\log n)$. Making a vertex the child of another vertex or removing the parent of a vertex takes the same time.

We next describe the algorithm.

Case 1. If (($|L_i| \geq n/k$) or $i = 0$), then

1. Execute the same steps as in Case 1 of algorithm MaxECDe1

Case 2. Else ($|L_i| \leq n/k$), then

1. *Step 1:* If in iteration i Case 1 was executed, then the vertices in J_i are labeled as active vertices. The algorithm SCC is executed in lock-step starting from *every* active vertex for up to k steps per active vertex. In addition to the regular scc operations the following steps are performed at each step of $dfs(x)$:
   - If $dfs(x)$ visits a vertex u that fulfils one of the following conditions, then x becomes passive, $dfs(x)$ stops, and u becomes the *stop vertex* $s(x)$ of $dfs(x)$. If none of the conditions hold, then u is labeled by x and $dfs(x)$ continues. The conditions are: *Vertex u is (i) a special vertex, (ii) is a non-special vertex labeled by an active vertex $y \neq x$, or (iii) is a non-special vertex labeled by a passive vertex v and the witness query of v returns a vertex $y \neq x$.* In Case (i) x becomes a child of u in W, in Case (ii) x becomes a child of y in W, and in Case (iii) x becomes a child of v in W. Note that this check happens as first step in the visit of u and that checking whether x should become passive and if yes, stopping and if no, labeling u by x is an atomic step in the lock-step search process that cannot be separated[5].
   - If an scc is detected, then its vertices are marked as *special*. For each newly special vertex u these steps are executed: (a) If it exists, the link from u to its parent in W is cut. (b) For all vertices v with $u = s(v)$, the link from y to its parent in W is cut and u becomes the new parent of v. (c) If u is an active vertex, it becomes passive, $dfs(u)$ is stopped and $s(u)$ is set to u.
2. *Step 2:* Set A to the set of current vertices of the DFS of active vertices at the end of Step 1 and set $B = \emptyset$. Unlabel all labeled vertices in the graph. Mark all special vertices as non-special and cut all links in W. If a bottom scc C was detected during Step 1, then remove $Attr_R(C)$ from the graph, set $L_{i+1} := L_i \cup Attr_R(C)$, and $B := J_{i+1} := (J_i \setminus Attr_R(C)) \cup Q_i$, where Q_i is the subset of vertices of V_i with an edge to $Attr_R(C)$. Then repeatedly execute the following step as long as $|A| + |B| \leq 2 \cdot n/\sqrt{k}$:
   (a) Perform lock-step searches started at all vertices in $A \cup B$ until a bottom scc is detected. These DFS do not label vertices and no vertices become passive. When a bottom scc is detected we stop all lock-step searches, remove $Attr_R(C)$ from the graph, set $L_{i+1} := L_{i+1} \cup Attr_R(C)$, and $B := J_{i+1} := (J_{i+1} \setminus Attr_R(C)) \cup Q_i$, where Q_i is the subset of vertices in the current graph with an edge to $Attr_R(C)$.
3. *Step 3:* $i := i+1$; if $V_i = \emptyset$, then stop the algorithm, else go to the next iteration.

We establish first the correctness of MaxECDe2 and then analyze its running time.

Lemma 2.6. *The algorithm* MaxECDe2 *correctly computes the maximal end-component decomposition of an MDP* P.

Proof. The algorithm repeatedly removes bottom sccs and their random attractors. Since every bottom scc is a mec (by Lemma 2.2) and in each step a random attractor is removed (hence in the current graph all the outgoing edges for random vertices are preserved), the correctness of the algorithm follows from Lemma 2.1 and Lemma 2.2. ∎

For the running time analysis we need to show that the set $A \cup B$ contains a vertex of every bottom scc of the

[5] This avoids that the following "race condition" where multiple active vertices labl a vertex without knowing of each other: The $dfs(x)$ checks u and determines that x should remain active. Immediately after the test the label of u is changed by some other DFS and now the outcome of the earlier test is wrong, i.e. x should become passive and should not label u.

graph at the beginning of Step 2 Case 2. We show this through the following lemmata.

LEMMA 2.7. *A vertex v labeled by a vertex y can only be relabeled by a vertex $z \neq y$ if y is passive.*

Proof. Assume $dfs(z)$ visits a vertex v labeled by an active vertex y. If $z \neq y$, then z becomes passive and $dfs(z)$ stops. Thus v is not labeled by z. Hence v can only be labeled by z if y is passive. ∎

LEMMA 2.8. *If a vertex becomes special, then all vertices reachable from it either are already special or become special as well.*

Proof. In SCC when a vertex x becomes special, i.e., when its scc C is detected, then all vertices reachable from x must have been visited. Furthermore all edges leaving C go to previously detected scc's. Thus, all vertices reachable from x either belong to the same scc as x or have become special before. ∎

LEMMA 2.9. *Invariants (DT1), (DT2), and (DT3) hold throughout the algorithm.*

Proof. We prove that (DT1) holds by induction on the number of time steps of the algorithm. At the beginning of Step 1 of MAXECDE2 all vertices are roots of their trees in W and the claim holds trivially. Assume the claim holds after time step t. If in time step $t+1$ a vertex becomes special, then it cuts the link to its parent. Whenever a vertex becomes passive, but is not special then it becomes the child of a vertex in W, i.e., passive non-special vertices cannot be the root of a tree of W. Additionally a vertex only becomes a non-root vertex in W in time step $t+1$ if it either becomes passive in time step $t+1$ or has already a stop vertex and the stop vertex becomes a special vertex. Thus (DT1) holds also after time step $t+1$.

To prove (DT2) assume the passive vertex x is a child of y in W. If y is an *active* vertex, then $dfs(x)$ visited a non-special vertex u labeled by y and stopped. Thus $u = s(x)$ and the claim holds. If y is *passive* and *non-special*, then consider the time step that makes x passive. Either (a) y was already passive and $dfs(x)$ visited a non-special vertex u labeled by y, or (b) y was an active vertex that became passive after x, which as we showed above implies that $s(x)$ is a non-special vertex labeled by y. Thus in both cases $u = s(x)$ was labeled by y when x became passive. This implies (1) by (I) that $s(x)$ has a path to $s(y)$, (2) that y can reach $s(x)$, and (3) that $s(x)$ is a non-special vertex. Consider finally the case that y is a *special* vertex. A special vertex y receives additional children in W (a) when y becomes special or (b) when a $dfs(x)$ visits y. In Case (a) all vertices x such that $s(x) = y$ become children of y. In Case (b) if $dfs(x)$ visits y and stops, then again $y = s(x)$ and the claim holds. Finally we also have to consider what happens to the possibly already existing children z of a non-special, active or passive vertex y when it becomes special. By Lemma 2.8 when a vertex y becomes special then all the vertices reachable from y also are or become special. If y was active when it becomes special, then $s(z)$ is labeled by y and hence reachable from y. If y was passive before, then as we showed above y can reach $s(z)$. Thus, in both cases y can reach $s(z)$. It follows that $s(z)$ either is already special or also becomes special. If $s(z)$ is already special, then z cannot be the child of an active or passive vertex because in both cases $s(z)$ is required to be non-special. Thus this contradicts our assumption that z is the child of y. If $s(z)$ becomes special in the same time step as y, then z will become a child of $s(z)$ and only stays a child of y is $y = s(z)$. Hence in all cases the claim holds.

We prove (DT3) by induction on the length l of the path from the passive vertex x to the root y of the tree. If $l = 1$ and the root is a special vertex, then the claim follows from (DT2). If the root is an active vertex y, then by (DT2) $s(x)$ is a non-special vertex labeled by y and, thus, the empty path proves the claim. Thus, it is not possible that a passive vertex is a root. If $l > 1$, let z be the parent of x in W. Since z is not the root of the tree, z is a passive vertex. By induction (DT3) holds for z, i.e., there exists a suitable path for z and by (DT2) there exists a path P' from $s(x)$ to $s(z)$. Combining P' with P proves (DT3) for x. ∎

COROLLARY 2.1. *At every point in Step 1 every passive vertex x has a path to either a special vertex or a vertex labeled by an active vertex $y \neq x$.*

Proof. By invariant (I) and the definition of a stop vertex there exists a path from x to $s(x)$ and by (DT3) there exists a path from $s(x)$ to either a special vertex or a vertex labeled by an active vertex $y \neq x$. Combining the two paths proves the lemma. ∎

Before showing the crucial two lemmata we need to introduce some more notation. Consider the i-th iteration of the outer loop in the algorithm and let G_i be the graph at the beginning of the i-th iteration. If iteration i executes Case 2 then let A_i be the value of A at the beginning of Step 2 in iteration i. Recall that Step 2 of Case 2 can execute multiple lock-step searches as long as $|A| + |B| \leq 2 \cdot n/\sqrt{k}$. After each lock-step search the bottom scc detected in the lock-step search and its random attractor are removed from the graph. Let $G_{i,t}$ be the graph after the removal of the vertices corresponding to the t-th lock-step search in iteration i.

Furthermore, let $B_{i,t}$ be the set B after this removal, i.e., the set of vertices in $G_{i,t}$ that have lost an edge due to the deletion of vertices since the beginning of iteration i. Note that A_i is not modified by the removal. We define $G_{i,0} = G_i$ and $B_{i,0} = \emptyset$.

LEMMA 2.10. *For any i and for every bottom scc C in G_i either C is detected in the i-th iteration of Step 1 or the set A_i contains a vertex in C.*

Proof. Lemma 2.4 showed that J_{i+1} contains a vertex of every bottom scc of G_i. Thus for every bottom scc C of G_i one vertex of C becomes an active vertex at the beginning of Step 1 of Case 2. Let C be a bottom scc in G_i and let x be an active vertex in C. If x is still active at the end of Step 1, then all vertices visited by $dfs(x)$ must belong to C, also its current vertex, which is added to A_i. If x is not active, then by Corollary 2.1 at the end of Step 1 x has a path to a vertex u and either u is a special vertex or u is labeled by an active vertex $y \neq x$. All paths from x remain in C, i.e., $u \in C$. Additionally if one vertex in an scc is marked as special, then all are marked as special. Thus, if u is special, it follows that x is marked as special as well and C was detected in Step 1 and the claim holds. If u is not special then let y be the label of u. Since y is active but u is not marked as special, $dfs(y)$ visited a vertex a of C at the end of Step 1, but has not yet detected C. Thus a belongs to A_i and the claim holds. ∎

LEMMA 2.11. *For any i and t and for every bottom scc C in $G_{i,t}$ the set $A_i \cup B_{i,t}$ contains a vertex of C.*

Proof. Consider any fixed i. We show the claim by induction on t: For $t = 0$, the claim follows from Lemma 2.10.

Next consider $t > 0$. During the t-th execution of Step 2 at least one bottom scc and its random attractor are removed from $G_{i,t-1}$ resulting in the graph $G_{i,t}$. Every bottom scc C in $G_{i,t}$ either was already a bottom scc in $G_{i,t-1}$ or not. If C was already a bottom scc in $G_{i,t-1}$, then C was not removed in the t-th iteration of Step 2. Thus by induction $A_i \cup B_{i,t-1}$ contains a vertex in C. Since $B_{i,t-1} \subseteq B_{i,t}$, the claim holds for C. If C was not a bottom scc in $G_{i,t-1}$ but is one in G_t, it must contain a vertex x with an edge (x, y) to a vertex y which was removed during iteration t. But then by definition x belongs to $B_{i,t}$. Thus the claim holds also in this case.

∎

LEMMA 2.12. *In all executions of Step 2 in Case 2 $|A| \leq n/\sqrt{2 \cdot k}$.*

Proof. If x and y are both active vertices at the end of Step 1, then it is not possible that both labeled the same vertex u as the visit of u of the latter of the two vertices would have caused the latter vertex to become passive. During a DFS of k steps an active vertex visits and labels at least $\sqrt{2 \cdot k}$ vertices. Thus there are at most $n/\sqrt{2 \cdot k}$ active vertices at the end of Step 1. ∎

THEOREM 2.2. *Given an MDP P, the algorithm* MAXECDE2 *computes the mec decomposition of P in time $O(m \cdot n^{2/3})$.*

Proof. The correctness follows from Lemma 2.6. We now present the running time analysis. Case 1 is executed every time that n/k vertices were deleted since the last execution of Case 1. Thus, this happens at most k times, i.e., the total time spent in Case 1 is $O(m \cdot k)$.

We analyze next the time spent in Step 1 of Case 2. Assume Case 2 is executed in iteration i. In Step 1 we start a DFS from each active vertex, i.e. each vertex in J_i. Note that a vertex is only added into J_i when it looses an outgoing edge. Thus, $\sum_i |J_i| \leq m$. The time spent per active vertex is $O(k + \log n)$ since the parent in W changes at most once for each vertex. Thus the total time spent in Step 1 of Case 2 over all iterations of the outer loop is $O(m(k + \log n))$.

We analyze next the time spend in Step 2 of Case 2. The cost of unlabeling vertices and cutting links in W is bounded by the work done in Step 1. The remaining work in Step 2 are the lock-step searches and the removal of vertices. By Lemma 2.11 $A \cup B$ contains a vertex of every bottom scc in the current graph and thus also a vertex of the smallest bottom scc C. Furthermore, if started on a vertex in C the algorithm SCC can detect C in time linear in the number of edges adjacent to C. The vertices of C and its attractor can be removed in time linear in the number of edges adjacent to them. Let us denote this number by $m(C)$. Thus the work done for detecting and removing a bottom scc and its attractor is $O(m(C) \cdot n/\sqrt{k})$. Since each edge is removed only once the total work spent in Step 2 of Case 2 is $O(m(k + \log n + n/\sqrt{k}))$. Choosing $k = n^{2/3}$ gives the claimed running time bound. ∎

2.2 Dynamic Maximal End-component Decomposition in Amortized Linear Time. We present algorithms for maintaining the mec decomposition of an MDP under the following operations: (a) *incremental algorithm:* addition of an edge (u, v) with $u \in V_1$; (b) *decremental algorithm:* deletion of an edge (u, v) with $u \in V_1$.

Given a graph $G = (V, E)$ with vertex partition (V_1, V_P), the *collapsed graph* $G_C = (V_C, E_C)$ with vertex partition (V_1^C, V_P^C) is defined as follows: Every mec C is collapsed to a single vertex that belongs to player 1, and all outgoing (resp. incoming) edges from (resp.

to) C are added to the graph, removing parallel edges. Formally, let $\mathcal{C}_m = \{ C \mid C \text{ is an mec} \}$ be the set of all mec's. Let $M = \bigcup_{C \in \mathcal{C}_m} C$. Then $V_C = \mathcal{C}_m \cup (V \setminus M)$ with $V_1^C = \mathcal{C}_m \cup (V_C \cap V_1)$ and $V_P^C = V_C \setminus V_1^C$.

$$
\begin{aligned}
E_C = & \{ (u,v) \mid u,v \in (V \setminus M), (u,v) \in E \} \\
& \cup \{ (C,v) \mid C \in \mathcal{C}_m, v \in (V \setminus M), \exists u \in C.(u,v) \in E \} \\
& \cup \{ (u,C') \mid C' \in \mathcal{C}_m, u \in (V \setminus M), \exists v \in C'.(u,v) \in E \} \\
& \cup \{ (C,C') \mid C, C' \in \mathcal{C}_m, \exists u \in C, \exists v \in C.(u,v') \in E \}
\end{aligned}
$$

An end-component C is *non-trivial* if $|C| \geq 2$, otherwise it is a *trivial* end-component. The collapsed graph with vertex partition (V_1^C, V_P^C) has the following property:

LEMMA 2.13. *The collapsed graph G_C with vertex partition (V_1^C, V_P^C) has no non-trivial end-components.*

Proof. If there is a non-trivial end-component in the collapsed graph G_C with the partition (V_1^C, V_P^C), then the union of the set of vertices of the end-component is an end-component in the original graph $G = (V, E)$ with partition (V_1, V_P), and this contradicts that the collapsed graph was obtained after the mec decomposition. ∎

The following lemma shows that if an edge (u, v) is added to a graph *with no non-trivial end-components*, then there is at most one non-trivial mec in the resulting graph. Thus, when an edge (u, v) with $u \in V_1$ is added to a graph G, then the insertion either (i) does not affect the collapsed graph at all (if u and v belonged to the same mec), or (ii) an edge is inserted into G_C but G_C still has no non-trivial mec's or (iii) the edge is inserted into G_C and G_C has now one non-trivial mec. This fact holds because the insertion of a player-1 edges does not split up any existing mec. However, the insertion of a random edge (u, v) with $u \in V_P$ will split up the mec containing u into a potentially large number of mec's if v does not belong to it. Thus, the following lemma holds for both player-1 and random edges only because it makes the strong assumption that the graph has no non-trivial end-component.

LEMMA 2.14. *Consider a graph $G = (V, E)$ with vertex partition (V_1, V_P) that has no non-trivial end-component. If we add an edge $e = (u, v)$ then $(V, E \cup \{ e \})$ with partition (V_1, V_P) either (a) still has no non-trivial end-component or (b) has at most one non-trivial maximal end-component. Additionally, for every scc C in the graph with the inserted edge if $u \notin C$, then the mec decomposition of C before and after the insertion are identical.*

Proof. Consider the mec decomposition after the edge insertion and assume C is a non-trivial mec that does not contain u. Then the insertion of (u, v) neither changed the edges between two vertices in C nor the edges leaving C. Thus C was also an end component before the insertion of (u, v). However, this contradicts the assumption that the MDP P does not any non-trivial mec's before the insertion. Thus, the insertion can have created at most one new mec, namely the mec containing u. Furthermore, the mec decomposition of at most one scc, namely the scc containing u in the updated graph, was changed by the edge insertion. The result follows. ∎

Incremental algorithm. Our incremental algorithm maintains as data structures (called IMEC data structures) (a) the collapsed graph $G_C = (V_C, E_C)$, (b) stores for every vertex in V_C the set of edges that are mapped to it, and (3) stores at every vertex $v \in V$ the vertex $v' \in V_C$ to which v is mapped. When an edge (u, v) with $u \in V_1$ is inserted it executes the following steps:

1. Compute the scc decomposition of the MDP graph (V_C, E_C) of G_C.
2. Consider the scc C that contains the vertex u.
3. If $|C| = |\{ u \}| = 1$, then stop since C is the new trivial mec.
4. Determine the set U of random vertices in C that have outgoing edges leaving C.
5. Compute $Z = \bigcup_{i \geq 0} Z_i$ with $Z_0 = U$ and for $i \geq 0$, $Z_{i+1} = Z_i \cup \{ v \in V_P \mid E(v) \cap Z_i \neq \emptyset \} \cup \{ v \in V_1 \mid E(v) \cap C \subseteq Z_i \cup \{ v \} \}$ [6]
6. Compute the scc decomposition of $C \setminus Z$ in the collapsed graph. If $C \setminus Z \neq \emptyset$ then there is a bottom scc C' with $|C'| \geq 2$ (see Lemma 2.15 below) and C' is the new unique non-trivial mec. Update the data structures accordingly.

LEMMA 2.15. *In Step 6, if $C \setminus Z \neq \emptyset$, then there is a unique bottom scc C' in $C \setminus Z$ with $|C'| \geq 2$.*

Proof. We assume that $U = C \setminus Z \neq \emptyset$. The following assertions must hold: (a) for all $u \in U \cap V_1$ we must have $E(u) \cap U \neq \emptyset$ (otherwise u would have been included in Z); (b) for all $u \in U \cap V_P$ we must have $E(u) \subseteq U$ (otherwise u would have been included in Z). It follows that every vertex in U has an out-edge in U, and hence the sub-graph induced by U must have a bottom scc. Consider a bottom scc C' in the sub-graph of U. If $|C'| = |\{ v \}| = 1$, then v must have a self-loop. Since by assumption random vertices do not have self-loops we must have $v \in V_1$. Then we have $v \in V_1$ and $E(v) \cap C \subseteq Z \cup \{ v \}$, and hence v must have been

[6] The definition of Z_{i+1} is similar to random attractor, the only difference is for a player-1 vertex v if all edges in C other than the self-loop is in Z_i, then v is included in Z_{i+1}.

included in Z, and this contradicts that $v \in C \setminus Z$. It follows that $|C'| \geq 2$. Since $|C'|$ is a bottom scc it follows from Lemma 2.2 that C' is a non-trivial mec. Since by Lemma 2.14 it follows that there is at most one non-trivial mec, it follows that C' is the unique non-trivial mec. ∎

By Lemma 2.1 the vertices in Z do not belong to any non-trivial mec. Thus, if $C \setminus Z = \emptyset$, then none of the vertices in C belong to an mec and thus no new mec was created in G_C. If $C \setminus Z \neq \emptyset$, then by Lemma 2.15 there exists a unique bottom scc in $C \setminus Z$, which according to Lemma 2.2 is a mec. Since Lemma 2.14 showed that the addition of an edge (u, v) with $u \in V_1$ generates at most one new non-trivial mec in G_C there are no further new mec's. Each step of the algorithm takes time $O(m)$. This result is summarized in Theorem 2.3. Part(1).

Note: The correctness and the running time analysis of the incremental algorithm only use the fact that the change in the graph modified the mec decomposition inside at most one scc and that the change created at most one new mec. Thus, the same algorithm can be used for updating the mec decomposition after an edge deletion, as long as it is guaranteed that the operation modifies the mec decomposition of at most one scc and creates at most one new mec. We will use this observation in Subsection 4.2.

Decremental Maximal End-component Decomposition. We consider maintaining the mec decomposition of an MDP under edge deletion for player-1 vertices. We argue that the simple static algorithm works in amortized linear time.

Decremental algorithm. We show that the simple static algorithm can be modified to handle the deletion of an edge (u, v) with $u \in V_1$ in amortized linear time. The observation is as follows: under player-1 edge deletion, the mec's of an MDP can only be decomposed into smaller mec's, and the size of the mec's do not increase. Given the mec decomposition of an MDP, we consider an edge deletion e for player 1. Then we have the following steps.

1. *Step 1.* If the edge e does not belong to any existing mec, then the edge e is deleted and we are done.
2. *Step 2.* If the edge e belongs to a mec C, then we run the simple static mec decomposition algorithm on the MDP induced by C.

The key argument to obtain amortized linear time analysis is as follows. In step 2 of the algorithm when the simple static algorithm is run. Recall that this algorithm runs in multiple iteration, spending linear-time in each iteration and stopping whenever an iteration does not remove any further vertices. Note that once the vertex v is removed, i.e., it falls out of the mec, it will never be part of any other mec. Thus, we charge the time for the last iteration of the simple static algorithm to the edge deletion operation and all previous iterations to the vertices that are removed in the previous iterations. Each vertex is charged at most once and hence the total time for deleting m edges is $O(nm + m^2) = O(m^2)$. Hence we have Theorem 2.3 part(2).

THEOREM 2.3. *The following assertions hold:*

1. *Let* P *be an MDP such that* P *has no non-trivial end-component. If we add an edge* (u, v) *with* $u \in V_1$*, then the maximal end-component decomposition can be computed in time* $O(m)$*.*
2. *Given an initial MDP with* m *edges, the maximal end-component decomposition can be maintained under the deletion of* $O(m)$ *edges* (u, v) *with* $u \in V_1$ *in total time* $O(m^2)$*.*

3 Definitions for the rest of the paper

We present all the formal definitions and notations related to strategies, objectives and almost-sure winning in Markov decision processes.

Plays and strategies. An infinite path, or a *play*, of the game graph G is an infinite sequence $\omega = \langle v_0, v_1, v_2, \ldots \rangle$ of vertices such that $(v_k, v_{k+1}) \in E$ for all $k \in \mathbb{N}$. We write Ω for the set of all plays, and for a vertex $v \in V$, we write $\Omega_v \subseteq \Omega$ for the set of plays that start from the vertex v. A *strategy* for player 1 is a function $\sigma\colon V^* \cdot V_1 \to V$ that chooses the next successor for all finite sequences $\vec{w} \in V^* \cdot V_1$ of vertices ending in a player-1 vertex (the sequence represents a prefix of a play). A strategy must respect the edge relation: for all $\vec{w} \in V^*$ and $v \in V_1$ we have $(v, \sigma(\vec{w} \cdot v)) \in E$. Player 1 follows the strategy σ if in each player-1 move, given that the current history of the game is $\vec{w} \in V^* \cdot V_1$, she chooses the next vertex according to $\sigma(\vec{w})$. We denote by Σ the set of all strategies for player 1. A *memoryless* player-1 strategy does not depend on the history of the play but only on the current vertex; i.e., for all $\vec{w}, \vec{w'} \in V^*$ and for all $v \in V_1$ we have $\sigma(\vec{w} \cdot v) = \sigma(\vec{w'} \cdot v)$. A memoryless strategy can be represented as a function $\sigma\colon V_1 \to V$. We denote by Σ^M the set of all memoryless strategies for player 1. The class of memoryless strategies represents the class of simple strategies without memory, and corresponds to the class of simple controllers in probabilistic systems.

Once a starting vertex $v \in V$ and a strategy $\sigma \in \Sigma$ is fixed, the outcome of the MDP is a random walk ω_v^σ for which the probabilities of events are uniquely defined, where an *event* $\mathcal{A} \subseteq \Omega$ is a measurable set of plays. For

a vertex $v \in V$ and an event $\mathcal{A} \subseteq \Omega$, we write $\Pr_v^\sigma(\mathcal{A})$ for the probability that a play belongs to $\mathcal{A}$ if the game starts from the vertex v and player 1 follows the strategy σ.

Objectives. We specify *objectives* for player 1 by providing a set of *winning* plays $\Phi \subseteq \Omega$. We say that a play ω *satisfies* the objective Φ if $\omega \in \Phi$. We consider *ω-regular objectives* [22], specified as parity conditions. We also define reachability objectives, which is an important special class of ω-regular objectives.

- *Reachability objectives.* Given a set $T \subseteq V$ of "target" vertices, the reachability objective requires that some vertex of T be visited. The set of winning plays is $\mathrm{Reach}(T) = \{ \langle v_0, v_1, v_2, \ldots \rangle \in \Omega \mid v_k \in T \text{ for some } k \geq 0 \}$.
- *Parity objectives.* For $c, d \in \mathbb{N}$, we write $[c..d] = \{ c, c+1, \ldots, d \}$. Let $p\colon V \to [0..d]$ be a function that assigns a *priority* $p(v)$ to every vertex $v \in V$, where $d \in \mathbb{N}$. For a play $\omega = \langle v_0, v_1, \ldots \rangle \in \Omega$, we define $\mathrm{Inf}(\omega) = \{ v \in V \mid v_k = v \text{ for infinitely many } k \}$ to be the set of vertices that occur infinitely often in ω. The *parity objective* is defined as $\mathrm{Parity}(p) = \{ \omega \in \Omega \mid \min\big(p(\mathrm{Inf}(\omega))\big) \text{ is even } \}$. In other words, the parity objective requires that the minimum priority visited infinitely often is even. In sequel we will use Φ to denote parity objectives.

Qualitative analysis: almost-sure winning. Given a player-1 objective Φ, a strategy $\sigma \in \Sigma$ is *almost-sure winning* for player 1 from the vertex v if $\Pr_v^\sigma(\Phi) = 1$. The *almost-sure winning set* $\langle\!\langle 1 \rangle\!\rangle_{almost}(\Phi)$ for player 1 is the set of vertices from which player 1 has an almost-sure winning strategy. The qualitative analysis of MDPs correspond to the computation of the almost-sure winning set for a given objective Φ. It follows from the results of [6, 7] that for all MDPs and all reachability and parity objectives, if there is an almost-sure winning strategy, then there is a memoryless almost-sure winning strategy.

THEOREM 3.1. ([6, 7]) *For all MDPs* P*, and all reachability and parity objectives* Φ*, there exists a memoryless strategy* σ_* *such that for all* $v \in \langle\!\langle 1 \rangle\!\rangle_{almost}(\Phi)$ *we have* $\Pr_v^{\sigma_*}(\Phi) = 1$.

Markov chains, closed recurrent sets. A Markov chain is a special case of MDP with $V_1 = \emptyset$, and hence for simplicity a Markov chain is a tuple $((V, E), \delta)$ with a probabilistic transition function $\delta : V \to \mathcal{D}(V)$, and $(u, v) \in E$ iff $\delta(u, v) > 0$. A *closed recurrent* set C of a Markov chain is a *bottom* maximal strongly connected component (scc) in the graph (V, E) (a bottom scc C is an scc with no outgoing edge from C to $V \setminus C$). Let $\mathcal{C} = \bigcup_{C \text{ is closed recurrent}} C$. It follows from the results on Markov chains [12] that for all $v \in V$, the set $\mathcal{C}$ is reached with probability 1 in finite time, and for all C such that C is closed recurrent, for all $u \in C$ and for all $v \in C$, if the starting vertex is u, then the vertex v is visited infinitely often with probability 1.

Derived Markov chain from a MDP and memoryless strategy. Given an MDP $\mathsf{P} = ((V, E), (V_1, V_P), \delta)$ and a memoryless strategy $\sigma_* : V_1 \to V$ a *derived Markov chain* $\mathsf{P}' = ((V, E'), \delta')$ is defined as follows: $E' = E \cap (V_P \times V) \cup \{ (u, v) \mid u \in V_1, v = \sigma_*(s) \}$; and $\delta'(u, v) = \delta(u, v)$ for $u \in V_P$, and $\delta'(u, v) = 1$ for $u \in V_1$ if $\sigma_*(u) = v$. In other words, for player-1 vertices we only keep the edges according to σ_* and assign them transition probability 1. We will denote by P_{σ_*} the derived Markov chain obtained from an MDP P by fixing a memoryless strategy σ_* in the MDP.

Avoiding a set U. Given a set U of vertices, we say that player 1 *can force to avoid* U from a vertex v if there is a strategy σ such that $\Pr_v^\sigma(\mathrm{Reach}(U)) = 0$, i.e., U is reached with probability 0. We say that player 1 *cannot force to avoid* U from v if for all strategies σ we have $\Pr_v^\sigma(\mathrm{Reach}(U)) > 0$, i.e., irrespective of the strategy of player 1 the set U is reached with positive probability.

Key property of attractors. Given an MDP P, and a set U of vertices, let $A = Attr_R(U)$. Then from A player 1 cannot force to avoid U, in other words, for all vertices in A and for all player-1 strategies, the set U is reached with positive probability. For $A = Attr_1(U)$ player 1 can ensure (with a memoryless strategy) that the set U is always reached.

4 Algorithms for Qualitative Analysis of Reachability Objectives

In this section we present an improved static algorithm and the first incremental and decremental algorithms for maintaining the almost-sure winning set for reachability objectives (i.e., $\langle\!\langle 1 \rangle\!\rangle_{almost}(\mathrm{Reach}(T))$. As usual, without loss of generality, we assume that every state in T is an absorbing vertex (vertex with only a self-loop). Since once a vertex in T is reached, the objective $\mathrm{Reach}(T)$ is satisfied, the assumption is made without loss of generality, and only for technical simplicity in the proofs.

4.1 Algorithm for almost-sure winning for reachability.

Given an MDP P and a set T of target vertices, we present a new algorithm to compute

$\langle\langle 1\rangle\rangle_{almost}(\text{Reach}(T))$. The algorithm of [4] works in $O(m \cdot \sqrt{m})$ time. We show that we can adapt MaxECDe2 to obtain an $O(m \cdot n^{2/3})$ time algorithm.

Improved algorithm for almost-sure reachability. Our algorithm first computes the set Q_0 of vertices from which there is no path to T. Then from the set $Attr_R(Q_0)$ player 1 cannot ensure to reach T with probability 1, and hence the set $Attr_R(Q_0)$ is removed from the graph. The set of vertices that lost an edge to $Attr_R(Q_0)$ becomes the *active* vertices in MaxECDe2. Then our algorithm iteratively detects bottom scc's such that the bottom scc C does not contain a target vertex (i.e., $C \cap T = \emptyset$). For a bottom scc C without a target vertex, for all vertices in C and all player-1 strategies the set T cannot be reached with probability 1. Thus, our algorithm removes $Attr_R(C)$ from the graph. The repeated bottom scc detection uses the MaxECDe2 algorithm with the following changes: In Case 2 Step 2, if a DFS tree reaches a vertex in T, then the DFS is stopped, and the other DFS are continued until a bottom scc is obtained; and in Case 2, Step 3, if all DFS reach a vertex in T, then the algorithm is stopped. We refer to the algorithm as AlmostReachNew.

Formal description of AlmostReachNew. We now present the formal description of algorithm AlmostReachNew. We will stop DFS searches when a vertex in T is reached.

Case 1. If (($|L_i| \geq n^{1/3}$) or $i = 0$), then

1. Compute the set R_i of vertices of the current graph (V_i, E_i) that can reach a vertex in T.
2. Let $Q_i = V_i \setminus R_i$ be the set of vertices such that there is no path to a vertex in T in the current graph (there is no edge from Q_i to R_i).
3. The set $Attr_R(Q_i)$ is removed from the graph.
4. The set L_{i+1} is the set of vertices removed from the graph in this iteration and J_{i+1} be the set of vertices in the remaining graph with an edge to L_{i+1}.
5. $i := i + 1$; if $Attr_R(Q_i) = V_i$ or $Q_i = \emptyset$, then stop the algorithm, else go to the next iteration.

Case 2. Else ($|L_i| \leq n^{1/3}$), then

1. *Step 1:* Execute Step 1 of Case 2 of MaxECDe2.
2. *Step 2:* We set A to the set of current vertices of the DFS of *active* vertices at the end of Step 1 and we set $B = \emptyset$. We unlabel all labeled vertices in the graph. We mark all special vertices as non-special and cut all links in W. If a bottom scc was detected during Step 1, then we execute the following steps (a)-(b) below.
   (a) The bottom scc Q_i such that $Q_i \cap T = \emptyset$ and its random attractor $Attr_R(Q_i)$ is removed from the graph. The set L_{i+1} is the set of vertices removed from the graph since the last iteration of Case 1 (i.e., $L_{i+1} := L_i \cup Attr_R(Q_i)$), and J_{i+1} be the set of vertices in the remaining graph with an edge to L_{i+1}, i.e., $J_{i+1} := (J_i \setminus Attr_R(Q_i)) \cup O_i$, where O_i is the subset of vertices of V_i with an edge to $Attr_R(Q_i)$.
   (b) Perform lock-step searches started at all vertices in $A \cup B$ until either (a) a bottom scc is detected or (b) all DFS reach a vertex in T. These DFS do not label vertices and no vertices become passive. If a DFS reaches a vertex in T, then the DFS is stopped. When a bottom scc is detected we stop all lock-step searches. If $|A| + |B| \leq 2 \cdot n^{2/3}$, we go to step (a) above, else we go to Case 1.
3. *Step 3:* $i := i+1$; if $Attr_R(Q_i) = V_i$ or $Q_i = \emptyset$, then stop the algorithm, else go to the next iteration.

Correctness and running time analysis. The detailed correctness proof is presented in the following theorem. The running time analysis of the algorithm is exactly the same as for algorithm MaxECDe2.

Theorem 4.1. *The algorithm* AlmostReachNew *computes the set* $\langle\langle 1\rangle\rangle_{almost}(Reach(T))$ *for an MDP* P *in time* $O(m \cdot n^{2/3})$.

Proof. We prove the correctness of algorithm AlmostReachNew. Let $Z_i = Attr_R(Q_i)$ be the set of vertices removed in iteration i. If Q_i is identified Case 1, then there is no path from Q_i to T, and if Q_i is identified in Case 2, then it is a bottom scc without a state in T (hence again there is no path from Q_i to T). Thus the algorithm ensures that in every iteration i, for the set of vertices Q_i identified by the algorithm there is no path to the set T. Hence from Q_i the set T cannot be reached with positive probability (which implies that T cannot be reached with probability 1). Since from $Attr_R(Q_i)$ the set Q_i is reached with positive probability against all strategies for player 1, it follows that from $Attr_R(Q_i)$ the set T cannot be ensured to be reached with probability 1. It follows that for the set Z_i of vertices removed we have $Z_i \subseteq V \setminus \langle\langle 1\rangle\rangle_{almost}(\text{Reach}(T))$. To complete the correctness argument we show that when the algorithm stops, the remaining set is $\langle\langle 1\rangle\rangle_{almost}(\text{Reach}(T))$. When the algorithm stops, let V_* be the set of remaining vertices. We first show that the following assertions hold: (a) for all $v \in V_* \cap V_P$ we have $E(v) \subseteq V_*$, and (b) for all vertices $v \in V_*$ there is a path to the set T. We prove (a) as follows: whenever the algorithm removes a set Z_i, it is a random attractor, and thus if a vertex $v \in V_* \cap V_P$ has an edge (v, z) with $z \in V \setminus V_*$, then v would have been included in $V \setminus V_*$, and thus (a) follows. We prove (b) as follows: if the algorithm stops in Case 1, then $Q_i = \emptyset$, and it follows that every

vertex in V_* can reach T. We next consider the case when the algorithm stops in Case 2: In this case every active vertex in J_i has a path to T, and it follows that there is no bottom scc in the graph induced by V_* that does not intersect with T. Since every vertex in V_* has an out-going edge, it follows every vertex in V_* has a path to T. Hence (b) follows. We now use assertions (a) and (b) to prove the result. Consider the shortest path (or the BFS tree) from all vertices in V_* to T, and for a vertex $v \in V_* \cap V_1$, let u be a successor for the shortest path. We consider the memoryless strategy σ_* that chooses the shortest path successor u to T for all vertices $v \in V_* \cap V_1$. Let $\ell = |V_*|$ and let α be the minimum of the positive transition probability of the MDP. For all vertices $v \in V_*$, the probability that T is reached within ℓ steps is at least α^ℓ, and it follows that the probability that T is not reached within $j \cdot \ell$ steps is at most $(1-\alpha^\ell)^j$, and this goes to 0 as j goes to ∞. It follows that for all $v \in V_*$ the memoryless strategy σ_* ensures that T is reached with probability 1. Hence the correctness follows. ∎

4.2 Dynamic almost-sure reachability. We present algorithms that maintain the almost-sure reachability set under the following operations: (a) *incremental algorithm:* addition of an edge (u, v) with $u \in V_1$; and (b) *decremental algorithm:* deletion of an edge (u, v) with $u \in V_1$.

Incremental algorithm. The incremental algorithm maintains a partition (A, B) of V with $A = \langle\langle 1 \rangle\rangle_{almost}(\text{Reach}(T))$ of the current MDP P and $B = V \setminus A$. Note that when an edge (u, v) with $u \in V_1$ is added, the set A can only grow. Thus once vertices are added to A they are never removed from A. Also observe that there are no edges from vertices $u \in B \cap V_1$ to A, as otherwise u would have been in $\langle\langle 1 \rangle\rangle_{almost}(\text{Reach}(T))$. Hence only edges from random vertices can cross from B to A. Consider the sub-MDP graph G^B induced by the set B of vertices. The algorithm keeps (1) the collapsed graph G^B_C (defined in Subsection 2.2) of G^B, (2) a mapping C of every vertex u of B to its vertex $C(u)$ in G^B_C, and (3) the mec decomposition of G^B_C in the incremental mec data structure IMEC of Subsection 2.2. As we show below adding an edge (u, v) with $u \in V_1$ to the graph G of P can only (i) insert an edge in G^B_C or (ii) delete edges (x, y) with $x \in V^C_P$ from G^B_C and collapse vertices in G^B_C to one vertex. We will show that the incremental mec algorithm of Subsection 2.2 can be used to handle (a) the insertion of a player-1 edge *as well as* (b) the deletion of a random edge as long as the MDP has no non-trivial mec and (c) the collapsing of vertices of a mec to a single vertex, each in time $O(m)$. Thus it can be used to maintain the mec decomposition of G^B_C in the incremental almost-sure reachability algorithm.

When an edge $e = (u, v)$ with $u \in V_1$ is added, the following steps are executed:

1. If $u \in A$, then we do nothing.
2. If $u \in B$ and $v \in A$, then move the vertices in $Attr_1(\{ C(u) \})$ in G^B_C to A and update the data structure for G^B_C (but not IMEC) accordingly. The set P of edges to be processed is set to $\{ (x, y) \mid x \in B, C(x) \in V^C_P,\ C(y) \in Attr_1(\{ C(u) \}) \}$. Then go to Step (4).
3. If $u \in B$ and $v \in B$, then add $(C(u), C(v))$ to G^B_C and update IMEC. If there is a new mec C in G^B_C, and there is a vertex $z \in C$ with $E(z) \cap A \neq \emptyset$, then move the vertices in $Attr_1(C)$ in G^B_C to A and update the data structure for G^B_C (but not IMEC) accordingly. We set $P = \{ (x, y) \mid x \in B, C(x) \in V^C_P,\ C(y) \in Attr_1(\{ C \}) \}$.
4. Until P is empty, execute the following step.
   (a) Remove an edge (u', v') from P and remove $(C(u'), C(v'))$ from G^B_C in IMEC.
   (b) Use IMEC to test if the deletion created a new mec C in G^B_C.
      i. If there is a new mec C in G^B_C and if there is a vertex $z \in C$ with $E(z) \cap A \neq \emptyset$, then move the vertices in $Attr_1(C)$ in G^B_C to A and update the data structure for G^B_C (but not IMEC) accordingly. Set $P = P \cup \{ (x, y) \mid x \in B, C(x) \in V^C_P,\ C(y) \in Attr_1(\{ C \}) \}$.
      ii. If there is a new mec C, but it does not not have the above edge to A then all of its vertices are collapsed into one vertex in G^B_C and IMEC is updated accordingly.

Note that there are at most as many vertex collapsing operations as edge deletion operation in IMEC. Correctness is established using the following lemma.

LEMMA 4.1. *Let P be an MDP and let $A \subseteq V$ and $B = V \setminus A$ be such that (a) for all $v \in B \cap V_1$ we have $E(v) \subseteq B$; (b) for all $v \in B$ we have $E(v) \cap B \neq \emptyset$. Consider the sub-MDP G^B induced by the vertex set B. Then the following assertions hold.*

1. **Property 1.** *If for all mec's C in the sub-MDP G^B, for all $v \in C \cap V_P$ we have $E(v) \cap A = \emptyset$, then for all $v \in B$, for all strategies σ for player 1 in the original MDP we have $\Pr^\sigma_v(Reach(A)) < 1$.*
2. **Property 2.** *For a mec C in the sub-MDP, if there exists a vertex $u \in C \cap V_P$ such that $E(u) \cap A \neq \emptyset$, then there exists a strategy σ such that for all $v \in C$ we have $\Pr^\sigma_v(Reach(A)) = 1$.*

Proof. We prove both the cases below.

Property 1. By contradiction assume that there exists a vertex $v \in B$ and a strategy σ such that

$\Pr_v^{\sigma}(\text{Reach}(A)) = 1$, and then by existence of memoryless strategies for almost-sure winning, there exists a memoryless strategy σ^* such that $\Pr_v^{\sigma^*}(\text{Reach}(A)) = 1$ (Theorem 3.1). Consider the derived Markov chain P_{σ_*} obtained by fixing the memoryless strategy σ^*, and let $Z \subseteq B$ be the set of vertices that reaches A with probability 1. Since from all vertices in $V \setminus (Z \cup A)$, the set of vertices A is reached with probability less than 1, it follows that (a) for all $u \in Z \cap V_1$ we have $\sigma^*(u) \in Z \cup A$ (and since $E(u) \cap A = \emptyset$ it follows $\sigma^*(u) \in Z$); and (b) for all $u \in Z \cap V_P$ we have $E(u) \subseteq (Z \cup A)$. It follows that in the derived Markov chain obtained by fixing σ^*, every vertex $u \in Z$ has an edge in Z. Consider the scc decomposition of the sub-graph induced by Z in P_{σ_*}, and let $C \subseteq Z$ be a bottom scc in the sub-graph of the derived Markov chain P_{σ_*}. Since the set A is reached from C with probability 1, it follows for some vertex $z \in C$ we have $E(z) \cap A \neq \emptyset$. Since by assumption for all $t \in B \cap V_1$ we have $E(t) \subseteq B$, it follows that the vertex $z \in Z \cap V_P$. The set C is an end-component in the sub-MDP G^B induced by B. Consider the mec C' in the sub-MDP G^B induced by B such that C is contained in C'. Then $z \in C \subseteq C'$ and $E(z) \cap A \neq \emptyset$. This contradicts the assumption, and hence the result follows.

Property 2. Consider a mec C in the sub-MDP G^B such that there is a vertex $z \in C \cap V_P$ with $E(z) \cap A \neq \emptyset$. Since C is an end-component in the sub-MDP G^B induced by B, it follows that for all $v \in C \cap V_P$ we have $E(v) \subseteq C \cup A$. Since C is strongly connected from every vertex $v \in C$ there is a path to z. We fix a memoryless strategy σ_* as follows: for every vertex $v \in C \cap V_1$ choose as successor the first vertex on a shortest path from v to z. Let $|C| = k$, and let α be the minimum positive transition probability of the MDP. Given the memoryless strategy σ_*, it follows that for all vertices $v \in C$, the vertex z is reached with probability at least α^k in k-steps, and hence A is reached with probability at least α^{k+1} within k-steps. Hence the probability that A is not reached after $(k+1)\cdot\ell$ steps from any vertex in C is at most $(1-\alpha^{k+1})^{\ell}$, and this goes to 0 as ℓ goes to ∞. It follows that for all $v \in C$ we have $\Pr_v^{\sigma^*}(\text{Reach}(A)) = 1$.

The result follows. ∎

LEMMA 4.2. *Let P be an MDP with a set T of vertices and let $A = \langle\langle 1 \rangle\rangle_{almost}(Reach(T))$. Consider the MDP P' obtained by addition of an edge $e = (u, v)$, with $u \in V_1$, to P. Let A' be the output of the incremental algorithm. Then $A' = \langle\langle 1 \rangle\rangle_{almost}(Reach(T))$ in P'.*

Proof. Let $B' = V \setminus A'$. When the algorithm terminates, for all mec's C in $G_C^{B'}$ and for all $v \in C \cap V_P$ we have $E(v) \cap A' = \emptyset$. It follows from Lemma 4.1 that for all $v \in B'$ and all strategies σ we have $\Pr_v^{\sigma}(\text{Reach}(T)) \leq \Pr_v^{\sigma}(\text{Reach}(A')) < 1$ in P' (since $T \subseteq A'$). By Lemma 4.1 for all vertices $v \in A'$ there is a strategy σ to ensure that A is reached with probability 1 in P', i.e., $\Pr_v^{\sigma}(\text{Reach}(A)) = 1$ in P'. Since A is already known to be almost-sure winning, it follows that $A' \subseteq \langle\langle 1 \rangle\rangle_{almost}(\text{Reach}(T))$ in P'. Thus $A' = \langle\langle 1 \rangle\rangle_{almost}(\text{Reach}(T))$ in P'. ∎

We still need to show how to update the IMEC data structure after edge deletions and vertex collapsing. We prove that the deletion of edges (u, v) with $u \in V_P$ can be handled in the same way as the incremental mec decomposition algorithm, for MDPs with no non-trivial end-component. The correctness proof of the incremental mec algorithm depends on Lemma 2.14, and we prove an equivalent lemma for the case of the deletion of an edge (u, v) with $u \in V_P$ in an MDP with no non-trivial end-component.

LEMMA 4.3. *Consider an MDP P that has no non-trivial end-component. If we delete an edge $e = (u, v)$ then the following assertion holds: (a) either the MDP still has no non-trivial end-component or (b) the MDP has at most one non-trivial maximal end-component. Additionally, for every scc C in the graph after the edge deletion if $u \notin C$, then the mec decomposition of C before and after the insertion are identical.*

Proof. Consider the mec decomposition after the edge deletion and assume C is a non-trivial mec that does not contain u. Then the deletion of (u, v) neither changed the edges between two vertices in C nor the edges leaving C. Thus C was also an end component before the deletion of (u, v). However, this contradicts the assumption that the MDP P does not any non-trivial mec's before the edge deletion. Thus, the deletion can have created at most one new mec, namely the mec containing u. Thus the mec decomposition of at most one scc, namely the scc containing u, was changed by the edge deletion. The result follows. ∎

We now consider the running time analysis. From Lemma 4.3 and the incremental mec decomposition algorithm (Theorem 2.3) we have the following corollary.

COROLLARY 4.1. *Let P be an MDP such that P has no non-trivial end-component. If we add an edge (u, v) with $u \in V_1$, or delete an edge (u, v) with $u \in V_P$, then the maximal end-component decomposition can be computed in time $O(m)$.*

LEMMA 4.4. *The operation of collapsing vertices of a mec C to a single vertex can be supported for the* IMEC *data structure in $O(m)$ time.*

Proof. Recall that the IMEC data structure consists of (a) the collapsed graph (V_C, E_C), (b) for every vertex u' in V_C a set of vertices of V that are mapped to u', and (c) a mapping of every vertex $v \in V$ to the vertex $v' \in G_C$ representing it (if any). The three pieces of the data structure are updated as follows: (a) To update the collapsed graph let $\{u_1, \ldots, u_l\}$ be the set of vertices that have to be collapsed. We create a new vertex u' and make all edges incident to any u_i incident to u'. (b) We union the vertex sets of all the vertices u_i and give the new set S to u'. (c) Then we update the mapping stored at every vertex $v \in S$, making it point to u' instead of a vertex u_i. ∎

The work done by the algorithm for each edge processed is $O(m)$, and an edge (u, v) is processed only when the vertex v has been included in A. Thus an edge is never processed twice, and hence we obtain the bound of Theorem 4.2.part(1).

Decremental almost-sure reachability. We present a decremental reachability algorithm for the almost-sure winning set for reachability objectives. The correctness argument of Theorem 4.1 shows that for the almost-sure winning set $V_* = \langle\langle 1 \rangle\rangle_{almost}(\text{Reach}(T))$, for all $v \in V_* \cap V_P$ we have $E(v) \subseteq V_*$ and for all $v \in V_*$ there is a path to T. For decremental reachability (under deletion of edges for player 1), the almost-sure set can only shrink, i.e., once a vertex is removed from the almost-sure winning set, it never gets included again. The simple algorithm for decremental reachability is as follows: the algorithm maintains a reachability tree to the set T under edge deletion. Once an edge is deleted, the reachability tree is updated, and if there is a new set Q that cannot reach T, then $Attr_R(Q)$ is removed from the graph (which leads to possibly more edge deletion), until there is no removal of vertices. The work done is as follows: (a) work done for maintaining the reachability tree to T and (b) the work done for the attractor computation which works exactly on the edges removed from the graph. A decremental reachability tree can be maintained in $O(m)$ worst case time, and $O(n)$ expected time with a randomized algorithm [16]. This gives us Theorem 4.2.part(2).

THEOREM 4.2. *The following assertions hold:*

1. *Given an initial MDP with m_1 edges, the $\langle\langle 1 \rangle\rangle_{almost}(Reach(T))$ computation under the addition of further m_2 edges (u, v) with $u \in V_1$ can be achieved in total time $O((m_1 + m_2)^2)$.*
2. *Given an initial MDP with m edges, the $\langle\langle 1 \rangle\rangle_{almost}(Reach(T))$ computation under deletion of $O(m)$ edges (u, v) with $u \in V_1$ can be achieved in total time $O(m^2)$ and expected total time $O(m \cdot n)$.*

Thus, if $m_1 = m_2 = m$, then the amortized time per insertion is $O(m)$.

5 Algorithms for Qualitative Analysis of Parity Objectives

In this section we consider the qualitative analysis of MDPs with parity objectives. Recall that a parity objective consists of a priority function $p : V \to \{0, 1, \ldots, 2d\}$, and the parity objective Parity(p) defines the measurable subset of infinite walks such that the minimum priority visited infinitely often is even. We present improved algorithm to compute $\langle\langle 1 \rangle\rangle_{almost}(\text{Parity}(p))$ based on the algorithms of the previous sections and the hierarchical clustering technique of Tarjan [21] (see also [13]).

Given a priority function $p : V \to \{0, 1, \ldots, 2d\}$, for $0 \leq m \leq d$, let $V_{\leq m} = \{v \in V \mid p(v) \leq m\}$ denote the set of vertices with priority at most m. Given an MDP P, let P_i denote the MDP obtained by removing $Attr_R(V_{\leq 2i-1})$ the set of vertices with priority less than $2i$ and its random attractor. A mec C is a *winning mec* in P_i if there exists $u \in C$ such that $p(u) = 2i$. Let WE_i be the union of the vertices of winning maximal end-component in P_i, and let $\mathsf{WE} = \cup_{0 \leq i \leq d} \mathsf{WE}_i$. In the following key lemma we present a reduction of $\langle\langle 1 \rangle\rangle_{almost}(\text{Parity}(p))$ to the almost-sure reachability of WE. Thus, it suffices to present an algorithm to compute the set WE.

LEMMA 5.1. *Given an MDP P we have $\langle\langle 1 \rangle\rangle_{almost}(\text{Parity}(p)) = \langle\langle 1 \rangle\rangle_{almost}(Reach(\mathsf{WE}))$.*

Proof. We present two directions of the proof.

$\langle\langle 1 \rangle\rangle_{almost}(Reach(\mathsf{WE})) \subseteq \langle\langle 1 \rangle\rangle_{almost}(\text{Parity}(p))$. We first show $\mathsf{WE} \subseteq \langle\langle 1 \rangle\rangle_{almost}(\text{Parity}(p))$. For that we first show that all winning mec's C in P_i are a subset of $\langle\langle 1 \rangle\rangle_{almost}(\text{Parity}(p))$, for all $0 \leq i \leq d$. Consider a winning mec C in P_i, for $0 \leq i \leq d$. Let $z \in C$ be such that $p(z) = 2i$. Since C is a mec, it is strongly connected, i.e., every vertex in C has a path to z. We fix a memoryless strategy as follows: for every vertex $v \in C \cap V_1$ we choose a successor that shortens the distance to z, and for z choose a successor (if $z \in V_1$) in C. Once the strategy is fixed we obtain a derived Markov chain and from every vertex in C, a closed recurrent set C' is reached with probability 1 such that z is in the closed recurrent set C'. Since $p(z)$ is even, and there is no smaller priority in P_i it follows from C the parity objective is satisfied with probability 1 in P_i. Observe that P_i is obtained by removing a random attractor, and hence it follows that in P_i, for all random edges (u, v) from vertices u in P_i, we have v also in P_i. It follows that the parity objective is

satisfied with probability 1 also in P. It follows that $\mathsf{WE} \subseteq \langle\langle 1\rangle\rangle_{almost}(\text{Parity}(p))$. Hence it also follows that $\langle\langle 1\rangle\rangle_{almost}(\text{Reach}(\mathsf{WE})) \subseteq \langle\langle 1\rangle\rangle_{almost}(\text{Parity}(p))$.

$\langle\langle 1\rangle\rangle_{almost}(\text{Parity}(p)) \subseteq \langle\langle 1\rangle\rangle_{almost}(\mathit{Reach}(\mathsf{WE}))$. We now show the converse, i.e., $\langle\langle 1\rangle\rangle_{almost}(\text{Parity}(p)) \subseteq \langle\langle 1\rangle\rangle_{almost}(\text{Reach}(\mathsf{WE}))$. For a vertex $v \in \langle\langle 1\rangle\rangle_{almost}(\text{Parity}(p))$ we fix a memoryless strategy σ_* that ensure Parity(p)) with probability 1 (such a memoryless strategy exists by Theorem 3.1). Consider the derived Markov chain obtained by fixing σ_*, and consider any closed recurrent set C of the derived Markov chain. Then we have $\min(p(C))$ is even, and let it be $2i$, for $0 \leq i \leq d$. It follows that C is an end-component in P_i, and the mec in P_i that contains C is a winning mec. It follows that for every closed recurrent set C in the derived Markov chain we have $C \subseteq \mathsf{WE}$. Since the strategy σ_* ensures that the set of closed recurrent sets is reached with probability 1, it follows that σ_* ensures that the set WE is reached with probability 1. Hence we have $v \in \langle\langle 1\rangle\rangle_{almost}(\text{Reach}(\mathsf{WE}))$.

The result follows. ∎

Informal description of the algorithm. If two vertices u, v belong to the same mec in P_i, they also belong to the same mec in P_{i-1}. Thus the mec's of P_i refine the ones of P_{i-1}, which can be exploited using the hierarchical clustering technique. Formally, we will compute WE by the recursive procedure WinMaxEC(P, p, i, j). The procedure takes an MDP, and two indices i and j, and outputs $\bigcup_{i \leq 2k \leq j} \mathsf{WE}_{2k}$. To obtain WE we invoke WinMaxEC$(\mathsf{P}, p, 0, 2d)$. Given the MDP P, and indices i, j, the procedure first computes the mec's of P_m, where $m = \lceil \frac{i+j}{2} \rceil$. If m is even, then the set WE_m of P_m is computed. Then we recursively call the procedures WinMaxEC$(\mathsf{P}_u, p, m+1, j)$ and WinMaxEC$(\mathsf{P}_\ell, p, i, m-1)$, where P_u is a subMDP containing only the edges *inside* the mec's of P_m and the MDP P_ℓ is obtained by collapsing each mec in P_m to a single vertex, thus containing only edges *outside* the mec's of P_m. We now present the formal description. Since the computation does not depend on the transition function δ of the MDP we will only consider the graph (V, E) of the MDP and the partition (V_1, V_P) of V.

Procedure WinMaxEC$(\mathsf{P} = ((V, E), (V_1, V_P)), p, i, j)$

1. If $j < i$, return $\emptyset$;
2. $m = \lceil \frac{i+j}{2} \rceil$;
3. $W = \emptyset$;
4. (a) $V_m = \{ v \in V \mid p(v) \geq m \}$; (b) $X_m = \mathit{Attr}_R(V \setminus V_m)$; (c) $Z_m = V \setminus X_m$; and (d) $E_m = E \cap (Z_m \times Z_m)$;
5. Find the mec decomposition of $\mathsf{P}_m = ((Z_m, E_m), (V_1 \cap Z_m, V_P \cap Z_m))$. For each $v \in V$, if v belongs to a mec in P_m, then we denote by $C_m(v)$ the mec that v belongs to. Let us denote by $\mathcal{C}_m$ the set of all mec's in P_m.
6. If m is even, then for all $v \in Z_m$, if v belongs to a mec in P_m and there is a vertex $v' \in C_m(v)$ such that $p(v') = m$, then include v in W.
7. Computing $\bigcup_{m+1 \leq k \leq j} \mathsf{WE}_k$.
   (a) *Vertex set.* $V_u := \{ v \in Z_m \mid p(v) \geq m+1$ and v belongs to a mec in $G_m \}$;
   (b) *Edges.* $E_u := \{ (u, v) \in E_m \cap (V_u \times V_u) \mid C_m(u) = C_m(v) \}$.
   (c) *Partition.* $(V_1^u, V_P^u) := (V_1 \cap V_u, V_P \cap V_u)$.
   (d) $W := W \cup$ WinMaxEC$(\mathsf{P}_u = ((V_u, E_u), (V_1^u, V_P^u)), p, m+1, j)$.
8. Computing $\bigcup_{i \leq k \leq m-1} \mathsf{WE}_k$.
   (a) Let $M = \bigcup_{C \in \mathcal{C}_m} C$;
   (b) *Vertex set.* $V_\ell = \mathcal{C}_m \cup (V \setminus M)$;
   (c) *Edges.* The set of edges are defined as follows:

$$E_\ell = \{ (u, v) \mid u, v \in (V \setminus M), (u, v) \in E \}$$
$$\cup \{ (C, v) \mid C \in \mathcal{C}_m, v \in (V \setminus M), \exists u \in C.(u, v) \in E \}$$
$$\cup \{ (u, C') \mid C' \in \mathcal{C}_m, u \in (V \setminus M), \exists v \in C'.(u, v) \in E \}$$
$$\cup \{ (C, C') \mid C, C' \in \mathcal{C}_m, \exists u \in C, \exists v \in C.(u, v') \in E \}$$

   (d) *Partition.* $V_1^\ell := \mathcal{C}_m \cup ((V \setminus M) \cap V_1)$ and $V_P^\ell := V_\ell \setminus V_1^\ell$.
   (e) $p_\ell(v) = p(v)$ for $v \in V$ and $p_\ell(C) = \min_{v \in C} p(v)$ for $C \in \mathcal{C}_m$;
   (f) $\widehat{W}_\ell := \text{WinMaxEC}(\mathsf{P}_\ell = ((V_\ell, E_\ell), (V_1^\ell, V_P^\ell)), p_\ell, i, m-1)$;
   (g) $W_\ell := (\widehat{W}_\ell \cap (V \setminus M)) \cup (\bigcup_{C \in \mathcal{C}_m \cap \widehat{W}_\ell} C)$.
   (h) $W := W \cup W_\ell$.
9. Return W.

Explanation and correctness. We explain steps 7 and 8, and then show correctness. In step 7, we create subMDPs of the mec's in P_m. The mec's of P_k, for $k > m$ are contained in the mec's of P_m. In step 8, we collapse all mec's $C \in \mathcal{C}_m$ to a single vertex. The vertex set of P_ℓ consists of (a) one vertex each for a mec $C \in \mathcal{C}_m$, and (b) the vertices in $V \setminus M$, where M is the union of the vertices of the mec's in P_m. The edges between vertices in $(V \setminus M)$ are according to the original edges of the graph, and the edges for vertices corresponding to mec's are added existentially (if there is a vertex in the mec that satisfy the edge relationship). We now present the correctness of the algorithm.

Lemma 5.2. (Correctness) *We have* WinMaxEC$(\mathsf{P}, p, 0, 2d) = \mathsf{WE}$*, i.e., the output of the algorithm* WinMaxEC *is the set of winning end-components.*

Proof. The correctness of the algorithm is by induction on $j - i$. The inductive claim is that $\textsc{WinMaxEC}(\mathsf{P}, p, i, j) = \bigcup_{i \leq k \leq j} \mathsf{WE}_k$. If $j = i$ the claim holds as the algorithm exactly outputs WE_j. Assume that the results hold for $j - i \leq k$, and we consider $j - i = k + 1$. If m is even, then

$$\bigcup_{i \leq k \leq j} \mathsf{WE}_k = \mathsf{WE}_m \cup \bigcup_{i \leq k \leq m-1} \mathsf{WE}_k \cup \bigcup_{m+1 \leq k \leq j} \mathsf{WE}_k,$$

otherwise (m is odd), and then

$$\bigcup_{i \leq k \leq j} \mathsf{WE}_k = \bigcup_{i \leq k \leq m-1} \mathsf{WE}_k \cup \bigcup_{m+1 \leq k \leq j} \mathsf{WE}_k.$$

We consider the following cases.

1. For all $k > m$ the mec's C_k of P_k are contained in the mec's C_m of P_m. Additionally, no random vertex in C_m can have an edge leaving C_m and thus no random vertex in C_k can have a random edge leaving C_m. Thus $\textsc{WinMaxEC}(\mathsf{P}_u, p, m + 1, j) = \textsc{WinMaxEC}(\mathsf{P}, p, m + 1, j)$. Combined with the inductive hypothesis we get $\bigcup_{m+1 \leq k \leq j} \mathsf{WE}_k = \textsc{WinMaxEC}(\mathsf{P}_u, p, m + 1, j)$.
2. For $k < m$, consider a mec C_k in P_k. If C_k contains a vertex v such that v belongs to a mec in P_m, then $C_m(v) \subseteq C_k$ (i.e., all vertices of the mec of v also belongs to C_k): this follows from maximality of the end-component C_k. By induction it follows that $\bigcup_{i \leq k \leq m-1} \mathsf{WE}_k = W_\ell$, where W_ℓ is computed in step 8.g.

Hence we have that $\textsc{WinMaxEC}(\mathsf{P}, p, i, j) = \bigcup_{i \leq k \leq j} \mathsf{WE}_k$. The results follows. ∎

Running time analysis. Given a MDP P with n vertices, m edges and a parity objective with d priorites, let us denote by $T(m, n, d)$ the running time of WinMaxEC on P. We observe that in E_u consists of edges with in mec's, and such edges are not present in E_ℓ. Thus we obtain the following recurrence relation for the running time $T(m, n, d)$ of WinMaxEC:

$$T(m,n,d) = T_M(m,n) + T(m_u, n, \lfloor \frac{d-1}{2} \rfloor) + T(m_\ell, n, \lceil \frac{d-1}{2} \rceil),$$

with $m_\ell + m_u \leq m$, and $T_M(m, n)$ denotes the time complexity of mec decomposition with m edges and n vertices. The $O(m \cdot \min\{ \sqrt{m}, n^{2/3} \})$ running time for the mec decomposition follows from Theorem 2.1 and Theorem 2.2. It is straightforward to show that $T(m, n, d) = O(m \cdot \min\{ \sqrt{m}, n^{2/3} \} \cdot \log(d))$.

For maintaining the almost-sure winning set under player-1 edge addition or deletion, we apply the procedure WinMaxEC, but with the incremental and decremental maintenance of the mec. For each P_i, for $0 \leq i \leq d$, we maintain the mec decomposition of P_i under edge insertion or edge deletion as described in Subsection 2.2. We apply the WinMaxEC algorithm and instead of the mec decomposition of P_m in Step 5 we invoke the incremental (resp. decremental) mec decomposition algorithm for edge insertions (resp. deletions). The addition (resp. deletion) of an edge may result in an addition (resp. deletion) of an edge in P_i in the recursive call. Hence Step 5 in the recursive call needs to maintain the mec decomposition incrementally (resp. decrementally). As above we obtain the following recurrence relation for the running time $T^I(m, n, d)$ of WinMaxEC for a MDP with m edges, n vertices, and parity objective with d-priorities for the incremental algorithm:

$$T^I(m,n,d) = T^I_M(m,n) + T^I(m_u, n, \lfloor \frac{d-1}{2} \rfloor) + T^I(m_\ell, n, \lceil \frac{d-1}{2} \rceil),$$

with $m_\ell + m_u \leq m$, and $T^I_M(m, n)$ denotes the time complexity of the incremental mec decomposition algorithm with m edges and n vertices. The recurrence relation for the decremental algorithm is similar. Using the bounds of Theorem 2.3 for the incremental and decremental mec decomposition algorithm we obtain Theorem 5.1.

Theorem 5.1. *Given an MDP P with a parity objective* $\text{Parity}(p)$ *with d priorities, the set* $\langle\langle 1 \rangle\rangle_{almost}(\text{Parity}(p))$ *can be computed in* $O(m \cdot \min\{ \sqrt{m}, n^{2/3} \} \cdot \log(d))$ *time. Given an initial MDP with m_1 edges, the* $\langle\langle 1 \rangle\rangle_{almost}(\text{Parity}(p))$ *computation under the addition of further m_2 edges (u, v) with $u \in V_1$ can be achieved in total time* $O((m_1 + m_2)^2 \cdot \log(d))$. *Given an initial MDP with m edges, the* $\langle\langle 1 \rangle\rangle_{almost}(\mathit{Reach}(T))$ *computation under deletion of $O(m)$ edges (u, v) with $u \in V_1$ can be achieved in total time* $O(m^2 \cdot \log(d))$.

If $m_1 = m_2 = m$, then the amortized time per insertion is $O(m \cdot \log(d))$.

Acknowledgements. We thank Tom Henzinger for his useful comments.

References

[1] A. Bianco and L. de Alfaro. Model checking of probabilistic and nondeterministic systems. In *FSTTCS 95: Software Technology and Theoretical Computer Science*, volume 1026 of *Lecture Notes in Computer Science*, pages 499–513. Springer-Verlag, 1995.

[2] K. Chatterjee and T. A. Henzinger. Probabilistic systems with limsup and liminf objectives. In *ILC*, pages 32–45, 2007.

[3] K. Chatterjee, T. A. Henzinger, B. Jobstmann, and R. Singh. Measuring and synthesizing systems in probabilistic environments. In *CAV 10*. Springer, 2010.

[4] K. Chatterjee, M. Jurdziński, and T.A. Henzinger. Simple stochastic parity games. In *CSL'03*, volume 2803 of *LNCS*, pages 100–113. Springer, 2003.

[5] K. Chatterjee, M. Jurdziński, and T.A. Henzinger. Quantitative stochastic parity games. In *SODA'04*, pages 121–130. SIAM, 2004.

[6] C. Courcoubetis and M. Yannakakis. The complexity of probabilistic verification. *Journal of the ACM*, 42(4):857–907, 1995.

[7] L. de Alfaro. *Formal Verification of Probabilistic Systems*. PhD thesis, Stanford University, 1997.

[8] L. de Alfaro, M. Faella, R. Majumdar, and V. Raman. Code-aware resource management. In *EMSOFT 05*. ACM, 2005.

[9] J. Filar and K. Vrieze. *Competitive Markov Decision Processes*. Springer-Verlag, 1997.

[10] H. Howard. *Dynamic Programming and Markov Processes*. MIT Press, 1960.

[11] N. Immerman. Number of quantifiers is better than number of tape cells. *Journal of Computer and System Sciences*, 22:384–406, 1981.

[12] J.G. Kemeny, J.L. Snell, and A.W. Knapp. *Denumerable Markov Chains*. D. Van Nostrand Company, 1966.

[13] V. King, O. Kupferman, and M. Y. Vardi. On the complexity of parity word automata. In *FoSSaCS*, pages 276–286. Springer, 2001.

[14] M. Kwiatkowska, G. Norman, and D. Parker. Verifying randomized distributed algorithms with prism. In *Workshop on Advances in Verification (WAVE'00)*, 2000.

[15] A. Pogosyants, R. Segala, and N. Lynch. Verification of the randomized consensus algorithm of Aspnes and Herlihy: a case study. *Distributed Computing*, 13(3):155–186, 2000.

[16] L. Roditty and U. Zwick. Improved dynamic reachability algorithms for directed graphs. In *FOCS*, pages 679–688. IEEE, 2002.

[17] R. Segala. *Modeling and Verification of Randomized Distributed Real-Time Systems*. PhD thesis, MIT, 1995. Technical Report MIT/LCS/TR-676.

[18] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, 1983.

[19] M.I.A. Stoelinga. Fun with FireWire: Experiments with verifying the IEEE1394 root contention protocol. In *Formal Aspects of Computing*, 2002.

[20] R. E. Tarjan. Depth first search and linear graph algorithms. *SIAM J. Computing*, 1(2):146–160, 1972.

[21] R. E. Tarjan. A hierarchical clustering algorithm using strong components. *Inf. Process. Lett.*, 14(1):26–29, 1982.

[22] W. Thomas. Languages, automata, and logic. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3, Beyond Words, chapter 7, pages 389–455. Springer, 1997.