Near-Linear Time Algorithms for Streett Objectives in Graphs and MDPs

Krishnendu Chatterjee
IST Austria, Klosterneuburg, Austria
krish.chat@ist.ac.at

Wolfgang Dvořák
Institute of Logic and Computation, TU Wien, Austria
dvorak@dbai.tuwien.ac.at

Monika Henzinger
Theory and Application of Algorithms, University of Vienna, Austria
monika.henzinger@univie.ac.at

Alexander Svozil
Theory and Application of Algorithms, University of Vienna, Austria
alexander.svozil@univie.ac.at

Abstract

The fundamental model-checking problem, given as input a model and a specification, asks for the algorithmic verification of whether the model satisfies the specification. Two classical models for reactive systems are graphs and Markov decision processes (MDPs). A basic specification formalism in the verification of reactive systems is the strong fairness (aka Streett) objective, where given different types of requests and corresponding grants, the requirement is that for each type, if the request event happens infinitely often, then the corresponding grant event must also happen infinitely often. All ω-regular objectives can be expressed as Streett objectives and hence they are canonical in verification. Consider graphs/MDPs with n vertices, m edges, and a Streett objectives with k pairs, and let b denote the size of the description of the Streett objective for the sets of requests and grants. The current best-known algorithm for the problem requires time $O(\min(n^2, m\sqrt{m \log n}) + b \log n)$. In this work we present randomized near-linear time algorithms, with expected running time $\widetilde{O}(m + b)$, where the $\widetilde{O}$ notation hides poly-log factors. Our randomized algorithms are near-linear in the size of the input, and hence optimal up to poly-log factors.

2012 ACM Subject Classification Mathematics of computing → Combinatorial algorithms; Software and its engineering → Formal software verification

Keywords and phrases model checking, graph games, Streett games

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2019.7

Funding A. S. is fully supported by the Vienna Science and Technology Fund (WWTF) through project ICT15-003. K.C. is supported by the Austrian Science Fund (FWF) NFN Grant No S11407-N23 (RiSE/SHiNE) and an ERC Starting grant (279307: Graph Games). For M.H the research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement no. 340506.

1 Introduction

In this work we present near-linear (hence near-optimal) randomized algorithms for the strong fairness verification in graphs and Markov decision processes (MDPs). In the fundamental model-checking problem, the input is a *model* and a *specification*, and the algorithmic verification problem is to check whether the model *satisfies* the specification. We first describe the models and the specifications we consider, then the notion of satisfaction, and then previous results followed by our contributions.

Models: Graphs and MDPs. Graphs and Markov decision processes (MDPs) are two classical models of reactive systems. The states of a reactive system are represented by the vertices of a graph, the transitions of the system are represented by the edges and non-terminating trajectories of the system are represented as infinite paths of the graph. Graphs are a classical model for reactive systems with nondeterminism, and MDPs extend graphs with probabilistic transitions that represent reactive systems with both nondeterminism and uncertainty. Thus graphs and MDPs are the standard models of reactive systems with nondeterminism, and nondeterminism with stochastic aspects, respectively [15, 2]. Moreover MDPs are used as models for concurrent finite-state processes [16, 28] as well as probabilistic systems in open environments [26, 23, 17, 2].

Specification: Strong fairness (aka Streett) objectives. A basic and fundamental specification formalism in the analysis of reactive systems is the *strong fairness condition*. The strong fairness conditions (aka Streett objectives) consist of k types of requests and corresponding grants, and the requirement is that for each type if the request happens infinitely often, then the corresponding grant must also happen infinitely often. Beyond safety, reachability, and liveness objectives, the most standard properties that arise in the analysis of reactive systems are Streett objectives, and chapters of standard textbooks in verification are devoted to it (e.g., [15, Chapter 3.3], [24, Chapter 3], [1, Chapters 8, 10]). In addition, ω-regular objectives can be specified as Streett objectives, e.g., LTL formulas and non-deterministic ω-automata can be translated to deterministic Streett automata [25] and efficient translations have been an active research area [6, 18, 22]. Consequently, Streett objectives are a canonical class of objectives that arise in verification.

Satisfaction. The notions of satisfaction for graphs and MDPs are as follows: For graphs, the notion of satisfaction requires that there is a trajectory (infinite path) that belongs to the set of paths specified by the Streett objective. For MDPs the satisfaction requires that there is a strategy to resolve the nondeterminism such that the Streett objective is ensured almost-surely (with probability 1). Thus the algorithmic model-checking problem of graphs and MDPs with Streett objectives is a central problem in verification, and is at the heart of many state-of-the-art tools such as SPIN, NuSMV for graphs [20, 14], PRISM, LiQuor, Storm for MDPs [23, 13, 17].

Our contributions are related to the algorithmic complexity of graphs and MDPs with Streett objectives. We first present previous results and then our contributions.

Previous results. The most basic algorithm for the problem for graphs is based on repeated SCC (strongly connected component) computation, and informally can be described as follows: for a given SCC, (a) if for every request type that is present in the SCC the corresponding grant type is also present in the SCC, then the SCC is identified as "good", (b) else vertices of each request type that have no corresponding grant type in the SCC are removed, and the algorithm recursively proceeds on the remaining graph. Finally, reachability to good SCCs is computed. The algorithm for MDPs is similar where the SCC computation is replaced with maximal end-component (MEC) computation, and reachability to good SCCs is replaced with probability 1 reachability to good MECs. The basic algorithms for graphs and MDPs with Streett objective have been improved in several works, such as for graphs in [19, 10], for MEC computation in [7, 8, 9], and MDPs with Streett objectives in [5]. For graphs/MDPs with n vertices, m edges, and k request-grant pairs with b denoting the size to describe the request grant pairs, the current best-known bound is $O(\min(n^2, m\sqrt{m \log n}) + b \log n)$.

Our contributions. In this work, our main contributions are randomized near-linear time (i.e. linear times a polylogarithmic factor) algorithms for graphs and MDPs with Streett objectives. In detail, our contributions are as follows:

- First, we present a near-linear time randomized algorithm for graphs with Streett objectives where the expected running time is $\widetilde{O}(m+b)$, where the $\widetilde{O}$ notation hides poly-log factors. Our algorithm is based on a recent randomized algorithm for maintaining the SCC decomposition of graphs under edge deletions, where the expected total running time is near linear [4].
- Second, by exploiting the results of [4] we present a randomized near-linear time algorithm for computing the MEC decomposition of an MDP where the expected running time is $\widetilde{O}(m)$. We extend the results of [4] from graphs to MDPs and present a randomized algorithm to maintain the MEC decomposition of an MDP under edge deletions, where the expected total running time is near linear [4].
- Finally, we use the result of the above item to present a near-linear time randomized algorithm for MDPs with Streett objectives where the expected running time is $\widetilde{O}(m+b)$.

All our algorithms are randomized and since they are near-linear in the size of the input, they are optimal up to poly-log factors. An important open question is whether there are deterministic algorithms that can improve the existing running time bound for graphs and MDPs with Streett objectives. Our algorithms are deterministic except for the invocation of the decremental SCC algorithm presented in [4].

Table 1 Summary of Results.

Problem	New Running Time	Old Running Time
Streett Objectives on Graphs	$\widetilde{O}(m+b)$	$\widetilde{O}(\min(n^2, m\sqrt{m})+b)$ [11, 19]
Almost-Sure Reachability	$\widetilde{O}(m)$	$O(m \cdot n^{2/3})$ [5, 9]
MEC Decomposition	$\widetilde{O}(m)$	$O(m \cdot n^{2/3})$ [9]
Decremental MEC Decomposition	$\widetilde{O}(m)$	$O(nm)$ [9]
Streett Objectives on MDPs	$\widetilde{O}(m+b)$	$\widetilde{O}(\min(n^2, m\sqrt{m})+b)$ [5]

2 Preliminaries

A *Markov decision process (MDP)* $P=((V,E),\langle V_1,V_R\rangle,\delta)$ consists of a finite set of vertices V partitioned into the player-1 vertices V_1 and the random vertices V_R, a finite set of edges $E \subseteq (V \times V)$, and a probabilistic transition function δ. The probabilistic transition function maps every random vertex in V_R to an element of $\mathcal{D}(V)$, where $\mathcal{D}(V)$ is the set of probability distributions over the set of vertices V. A random vertex v has an edge to a vertex $w \in V$, i.e. $(v,w) \in E$ iff $\delta(v)[w] > 0$. An edge $e=(u,v)$ is a *random edge* if $u \in V_R$ otherwise it is a *player-1 edge*. W.l.o.g. we assume $\delta(v)$ to be the uniform distribution over vertices u with $(v,u) \in E$.

Graphs are a special case of MDPs with $V_R = \emptyset$. The set $In(v)$ ($Out(v)$) describes the set of predecessors (successors) of a vertex v. More formally, $In(v)$ is defined as the set $\{w \in V \mid (w,v) \in E\}$ and $Out(v) = \{w \in V \mid (v,w) \in E\}$. When U is a set of vertices, we define $E(U)$ to be the set of all edges incident to the vertices in U. More formally, $E(U) = \{(u,v) \in E \mid u \in U \vee v \in U\}$. With $G[S]$ we denote the subgraph of a graph $G=(V,E)$ induced by the set of vertices $S \subseteq V$. Let $\mathsf{GraphReach}(S)$ be the set of vertices in G that can reach a vertex of $S \subseteq V$. The set $\mathsf{GraphReach}(S)$ can be found in linear time using depth-first search [27]. When a vertex u can reach another vertex v and vice versa, we say that u and v are *strongly connected*.

A *play* is an infinite sequence $\omega = \langle v_0, v_1, v_2, \ldots \rangle$ of vertices such that each $(v_{i-1}, v_i) \in E$ for all $i \geq 1$. The set of all plays is denoted with Ω. A play is initialized by placing a token on an initial vertex. If the token is on a vertex owned by player-1, he moves the token along one of the outgoing edges, whereas if the token is at a random vertex $v \in V_R$, the next vertex is chosen according to the probability distribution $\delta(v)$. The infinite sequence of vertices (infinite walk) formed in this way is a play.

Strategies are recipes for player 1 to extend finite prefixes of plays. Formally, a player-1 *strategy* is a function $\sigma : V^* \cdot V_1 \mapsto V$ which maps every finite prefix $\omega \in V^* \cdot V_1$ of a play that ends in a player-1 vertex v to a successor vertex $\sigma(\omega) \in V$, i.e., $(v, \sigma(\omega)) \in E$. A player-1 strategy is *memoryless* if $\sigma_1(\omega) = \sigma_1(\omega')$ for all $\omega, \omega' \in V^* \cdot V_1$ that end in the same vertex $v \in V_i$, i.e., the strategy does not depend on the entire prefix, but only on the last vertex. We write Σ for the set of all strategies for player 1.

The *outcome of strategies* is defined as follows: In graphs, given a starting vertex, a strategy for player 1 induces a unique play in the graph. In MDPs, given a starting vertex v and a strategy $\sigma \in \Sigma$, the outcome of the game is a random walk w_v^σ for which the probability of every event is uniquely defined, where an *event* $\mathcal{A} \subseteq \Omega$ is a measurable set of plays [28]. For a vertex v, strategy $\sigma \in \Sigma$ and an event $\mathcal{A} \subseteq \Omega$, we denote by $\Pr_v^\sigma(\mathcal{A})$ the probability that a play belongs to $\mathcal{A}$ if the game starts at v and player 1 follows σ.

An *objective* $\phi \subseteq \Omega$ for player 1 is an event, i.e., objectives describe the set of winning plays. A play $\omega \in \Omega$ *satisfies* the objective if $\omega \in \phi$. In MDPs, a player-1 strategy $\sigma \in \Sigma$ is almost-sure (a.s.) winning from a starting vertex $v \in V$ for an objective ϕ iff $\Pr_v^\sigma(\phi) = 1$. The *winning set* $\langle\!\langle 1 \rangle\!\rangle_{a.s.}(\phi)$ for player 1 is the set of vertices from which player 1 has an almost-sure winning strategy. We consider Reachability objectives and k-pair Streett objectives.

Given a set $T \subseteq V$ of vertices, the *reachability objective* $\mathit{Reach}(T)$ requires that some vertex in T be visited. Formally, the sets of winning plays are $\mathit{Reach}(T) = \{\langle v_0, v_1, v_2, \ldots \rangle \in \Omega \mid \exists k \geq 0 \text{ s.t. } v_k \in T\}$. We say v can reach u almost-surely (a.s.) if $v \in \langle\!\langle 1 \rangle\!\rangle_{a.s.}(\mathit{Reach}(\{u\}))$.

The *k-pair Streett objective* consists of k-Streett pairs $(L_1, U_1), (L_2 U_2), \ldots, (L_k, U_k)$ where all $L_i, U_i \subseteq V$ for $1 \leq i \leq k$. An infinite path satisfies the objective iff for all $1 \leq i \leq k$ some vertex of L_i is visited infinitely often, then some vertex of U_i is visited infinitely often.

Given an MDP $P = (V, E, \langle V_1, V_R \rangle, \delta)$, an *end-component* is a set of vertices $X \subseteq V$ s.t. (1) the subgraph induced by X is strongly connected (i.e., $(X, E \cap X \times X)$ is strongly connected) and (2) all random vertices have their outgoing edges in X. More formally, for all $v \in X \cap V_R$ and all $(v, u) \in E$ we have $u \in X$. In a graph, if (1) holds for a set of vertices $X \subseteq V$ we call the set X strongly connected subgraph (SCS). An end-component, SCS respectively, is *trivial* if it only contains a single vertex with no edges. All other end-components, SCSs respectively, are *non-trivial*. A *maximal end-component* (MEC) is an end-component which is maximal under set inclusion. The importance of MECs is as follows: (i) it generalizes *strongly connected components* (SCCs) in graphs (with $V_R = \emptyset$) and closed recurrent sets of Markov chains (with $V_1 = \emptyset$); and (ii) in a MEC X, player-1 can almost-surely reach all vertices $u \in X$ from every vertex $v \in X$. The MEC-decomposition of an MDP is the partition of the vertex set into MECs and the set of vertices which do not belong to any MEC. The condensation of a graph G denoted by $\mathsf{CONDENSE}(G)$ is the graph where all vertices in the same SCC in G are contracted. The vertices of $\mathsf{CONDENSE}(G)$ are called *nodes* to distinguish them from the vertices in G.

Let C be a strongly connected component (SCC) of $G = (V, E)$. The SCC C is a *bottom SCC* if no vertex $v \in C$ has an edge to a vertex in $V \setminus C$, i.e., no outgoing edges. Consider an MDP $P = (V, E, \langle V_1, V_R \rangle, \delta)$ and notice that every bottom SCC C in the graph $G = (V, E)$ is a MEC because no vertex (and thus no random vertex) has an outgoing edge.

A *decremental graph algorithm* allows the deletion of player-1 edges while maintaining the solution to a graph problem. It usually allows three kinds of operations: (1) *preprocessing*, which is computed when the initial input is first received, (2) *delete*, which deletes a player-1 edge and updates the data structure, and (3) *query*, which computes the answer to the problem. The *query time* is the time needed to compute the answer to the query. The *update time* of a decremental algorithm is the cost for a *delete* operation. We sometimes refer to the delete operations as *update operation*. The running time of a decremental algorithm is characterized by the *total update time*, i.e., the sum of the update times over the entire sequence of deletions. Sometimes a decremental algorithm is randomized and assumes an *oblivious adversary* who fixes the sequence of updates in advance. When we use a decremental algorithm which assumes such an oblivious adversary as a subprocedure the sequence of deleted edges must not depend on the random choices of the decremental algorithm.

3 Decremental SCCs

We first recall the result about decremental strongly connected components maintenance in [4] (cf. Theorem 1 below) and then augment the result for our purposes.

▶ **Theorem 1** (Theorem 1.1 in [4]). *Given a graph $G = (V, E)$ with m edges and n vertices, we can maintain a data structure $\mathcal{A}$ that supports the operations*

- delete(u, v)*: Deletes the edge (u, v) from the graph G.*
- query(u, v)*: Returns whether u and v are in the same SCC in G,*

in total expected update time $O(m \log^4 n)$ and with worst-case constant query time. The bound holds against an oblivious adversary.

The preprocessing time of the algorithm is $O(m + n)$ using [27]. To use this algorithm we extend the query and update operations with three new operations described in Corollary 2.

▶ **Corollary 2.** *Given a graph $G = (V, E)$ with m edges and n vertices, we can maintain a data structure $\mathcal{A}$ that supports the operations*

- rep(u) *(query-operation): Returns a reference to the SCC containing the vertex u.*
- deleteAnnounce(E) *(update-operation): Deletes the set E of edges from the graph G. If the edge deletion creates new SCCs $C_1, \ldots, C_k$ the operation returns a list $Q = \{C_1, \ldots, C_k\}$ of references to the new SCCs.*
- deleteAnnounceNoOutgoing(E) *(update-operation): Deletes the set E of edges from the graph G. The operation returns a list $Q = \{C_1, \ldots, C_k\}$ of references to all new SCCs with no outgoing edges.*

in total expected update time $O(m \log^4 n)$ and worst-case constant query time for the first operation. The bound holds against an oblivious adaptive adversary.

The first function is available in the algorithm described in [4]. The second function can be implemented directly from the construction of the data structure maintained in [4]. The key idea for the third function is that when an SCC splits, we consider the new SCCs. We distinguish between the largest of them and the others which we call small SCCs. We then consider all edges incident to the small SCCs: Note that as the new outgoing edges in the large SCC are also incident to a small SCC we can also determine the outgoing edges of the large SCC. Observe that whenever an SCC splits all the small SCCs are at most half the size of the original SCC. That is, each vertex can appear only $O(\log n)$ times in small SCCs during the whole algorithm. As an edge is only considered if one of the incident vertices is in a small SCC each edge is considered $O(\log n)$ times and the additional running time is bounded by $O(m \log n)$. Furthermore, we define T_d as the running time of the best decremental SCC algorithm which supports the operations in Corollary 2. Currently, $T_d = O(m \log^4 n)$.

4 Graphs with Streett Objectives

In this section, we present an algorithm which computes the winning regions for graphs with Streett objectives. The input is a directed graph $G = (V, E)$ and k Streett pairs (L_j, U_j) for $j = 1, \ldots, k$. The size of the input is measured in terms of $m = |E|$, $n = |V|$, k and $b = \sum_{j=1}^{k}(|L_j| + |U_j|) \leq 2nk$.

Algorithm Streett and good component detection. Let C be an SCC of G. In the good component detection problem, we compute (a) a non-trivial SCS $G[X] \subseteq C$ induced by the set of vertices X, such that for all $1 \leq j \leq k$ either $L_j \cap X = \emptyset$ or $U_j \cap X \neq \emptyset$ or (b) that no such SCS exists. In the first case, there exists an infinite path that eventually stays in X and satisfies the Streett objective, while in the latter case, there exists no path which satisfies the Streett objective in C. From the results of [1, Chapter 9, Proposition 9.4] the following algorithm, called Algorithm Streett, suffices for the winning set computation:

1. Compute the SCC decomposition of the graph;
2. For each SCC C for which the good component detection returns an SCS, label the SCC C as satisfying.
3. Output the set of vertices that can reach a satisfying SCC as the winning set.

Since the first and last step are computable in linear time, the running time of Algorithm Streett is dominated by the detection of good components in SCCs. In the following, we assume that the input graph is strongly connected and focus on the good component detection.

Bad vertices. A vertex is *bad* if there is some $1 \leq j \leq k$ such that the vertex is in L_j but it is not strongly connected to any vertex of U_j. All other vertices are good. Note that a good vertex might become bad if a vertex deletion disconnects an SCS or a vertex of a set U_j. A good component is a non-trivial SCS that contains only good vertices.

Decremental strongly connected components. Throughout the algorithm, we use the algorithm described in Section 3 to maintain the SCCs of a graph when deleting edges. In particular, we use Corollary 2 to obtain a list of the new SCCs which are created by removing bad vertices. Note that we can "remove" a vertex by deleting all its incident edges. Because the decremental SCC algorithm assumes an oblivious adversary we sort the list of the new SCCs as otherwise the edge deletions performed by our algorithm would depend on the random choices of the decremental SCC algorithm.

Data structure. During the course of the algorithm, we maintain a decomposition of the vertices in $G = (V, E)$: We maintain a list Q of certain sets $S \subseteq V$ such that every SCC of G is contained in some S stored in Q. The list Q provides two operations: $Q.\texttt{add}(X)$ enqueues X to Q; and $Q.\texttt{deque}()$ dequeues an arbitrary element X from Q. For each set S in the decomposition, we store a data structure $D(S)$ in the list Q. This data structure $D(S)$ supports the following operations

1. $\texttt{Construct}(S)$: initializes the data structure for the set S
2. $\texttt{Remove}(S, B)$ updates S to $S \setminus B$ for a set $B \subseteq V$ and returns $D(S)$ for the new set S.
3. $\texttt{Bad}(S)$ returns a reference to the set $\{v \in S \mid \exists j \text{ with } v \in L_j \text{ and } U_j \cap S = \emptyset\}$
4. $\texttt{SCCs}(S)$ returns the set of SCCs currently in $G[S]$. We implement $\texttt{SCCs}(S)$ as a balanced binary search tree which allows logarithmic and updates and deletions.

In [19] an implementation of this data structure with functions (1)-(3) is described that achieves the following running times. For a set of vertices $S \subseteq V$, let $bits(S)$ be defined as $\sum_{j=1}^{k}(|S \cap L_j| + |S \cap U_j|)$.

▶ **Lemma 3** (Lemma 2.1 in [19]). *After a one-time preprocessing of time $O(k)$, the data structure $D(S)$ can be implemented in time $O(bits(S) + |S|)$ for* `Construct`(S)*, time $O(bits(B) + |B|)$ for* `Remove`(S, B) *and constant running time for* `Bad`(S).

We augment the data structure with the function `SCCs`(S) which runs in total time of a decremental SCC algorithm supporting the first function in Corollary 2.

Algorithm Description. The key idea is that the algorithm maintains the list Q of data structures $D(S)$ as described above when deleting bad vertices. Initially, we enqueue the data structure returned by `Construct`(V) to Q. As long as Q is non-empty, the algorithm repeatedly pulls a set S from Q and identifies and deletes bad vertices from $G[S]$. If no edge is contained in $G[S]$, the set S is removed as it can only induce trivial SCCs. Otherwise, the subgraph $G[S]$ is either determined to be strongly connected and output as a good component or we identify and remove an SCC with at most half of the vertices in $G[S]$. Consider Figure 1 for an illustration of an example run of Algorithm 1.

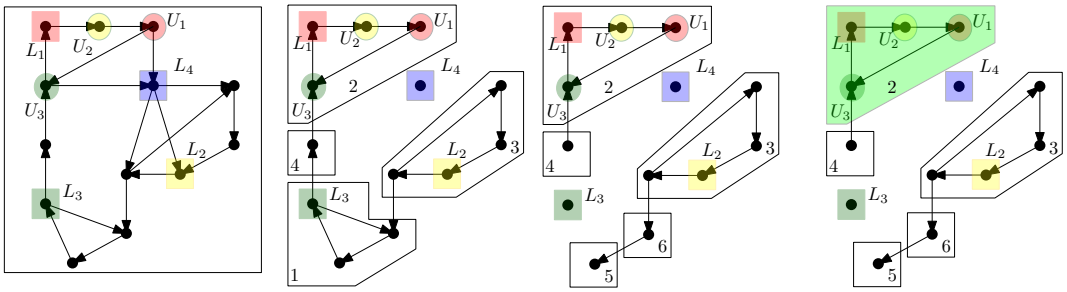

Figure 1 Illustration of one run of Algorithm 1: The vertex in the set L_4 is a bad vertex and we remove it from the SCC yielding four new SCCs. First, we look in the SCC containing L_3. The vertex in L_3 is a bad vertex because there is no vertex in U_3 in this SCC. Again two SCCs are created after its removal. The next SCC we process is the SCC containing L_1. It is a good component because the vertex in L_1 has a vertex in U_1 in the same SCC. No bad vertices are removed and the whole SCC is identified as a good component.

Outline correctness and running time. In the following, when we talk about the *input graph* $\hat{G}$ we mean the unmodified, strongly connected graph which we use to initialize Algorithm 1. In contrast, with the *current* graph G we refer to the graph where we already deleted vertices and their incident edges in the course of finding a good component. For the correctness of Algorithm 1, we show that if a good component exists, then there is a set S stored in list Q which contains all vertices of this good component.

To obtain the running time bound of Algorithm 1, we use the fact that we can maintain the SCC decomposition under deletions in $O(T_d)$ total time. With the properties of the data structure described in Lemma 3 we get a running time of $\widetilde{O}(n+b)$ for the maintenance of the data structure and identification of bad vertices over the whole algorithm. Combined, these ideas lead to a total running time of $\widetilde{O}(T_d + n + b)$ which is $\widetilde{O}(m + b)$ using Corollary 2.

▶ **Lemma 4.** *Algorithm 1 runs in expected time $\widetilde{O}(m + b)$.*

Algorithm 1: Algorithm GoodComp.

Input: Strongly connected graph $G = (V, E)$ and Streett pairs (L_j, U_j) for $j = 1, \ldots, k$
Output: a good component in G if one exists

1 Invoke an instance $\mathcal{A}$ of the decremental SCC algorithm; Initialize Q as a new list.;
2 $D(V) = \texttt{Construct}(V)$; $D(V).\texttt{SCCs}(V) \leftarrow \{\mathcal{A}.\texttt{rep}(x)\}$ for some $x \in V$;
3 $Q.\texttt{add}(D(V))$;
4 **while** Q *is not empty* **do**
5 $D(S) \leftarrow Q.\texttt{deque}()$;
6 **while** $D(S).\texttt{Bad}(S)$ *is not empty* **do**
7 $B \leftarrow D(S).\texttt{Bad}(S)$; $D(S) \leftarrow D(S).\texttt{Remove}(S, B)$;
  `// obtain SCCs after deleting bad vertices from` S
8 $D(S).\texttt{SCCs}(S) \leftarrow D(S).\texttt{SCCs}(S) \setminus \left(\bigcup_{b \in B}\{\mathcal{A}.\texttt{rep}(b)\}\right)$;
9 $D(S).\texttt{SCCs}(S) \leftarrow D(S).\texttt{SCCs}(S) \cup \mathcal{A}.\texttt{deleteAnnounce}(E(B))$;
10 **if** $G[S]$ *contains at least one edge* **then**
11 Initialize K as a new list;
12 **for** $X \leftarrow D(S).\texttt{SCCs}(S)$ **do**
13 **if** $X = S$ **then output** $G[S]$; `// good component found`
14 **if** $|X| \leq \frac{|S|}{2}$ **then** $K.\texttt{add}(X)$;
15 Sort the SCCs in K by vertex id (look at all the vertices in each SCC of K);
16 $R \leftarrow \emptyset$; `//` Build $D(X)$ for SCCs X in K and remove X from S,$D(S)$ and $\texttt{SCCs}(S)$
17 **for** $X \leftarrow K.\texttt{deque}()$ **do**
18 $R \leftarrow R \cup X$; $D(X) = \texttt{Construct}(X)$;
19 $D(X).\texttt{SCCs}(X) \leftarrow \{\mathcal{A}.\texttt{rep}(x)\}$ for some $x \in X$;
20 $D(S).\texttt{SCCs}(S) \leftarrow D(S).\texttt{SCCs}(S) \setminus \{\mathcal{A}.\texttt{rep}(x)\}$ for some $x \in X$;
21 $Q.\texttt{add}(D(X))$;
22 **if** $D(S).\texttt{SCCs}(S) \neq \emptyset$ **then** $Q.\texttt{add}(D(S).\texttt{Remove}(S, R))$;
23 **return** No good component exists.

Proof. The preprocessing and initialization of the data structure D and the removal of bad vertices in the whole algorithm takes time $O(m + k + b)$ using Lemma 3. Since each vertex is deleted at most once, the data structure can be constructed and maintained in total time $O(m)$. Announcing the new SCCs after deleting the bad vertices at Line 9 is in $O(T_d) = \widetilde{O}(m)$ total time by Corollary 2. Consider an iteration of the while loop at Line 4: A set S is removed from Q. Let us denote by n' the number of vertices of S. If $G[S]$ does not contain any edge after the removal of bad vertices, then S is not considered further by the algorithm. Otherwise, the for-loop at Line 12 considers all new SCCs. Note the we can implement the for-loop in a lockstep fashion: In each step for each SCC we access the i-th vertex and as soon as all of the vertices of an SCC are accessed we add it to the list K. When only one SCC is left we compute its size using the original set S and the sizes of the other SCCs. If its size is at most $|S|/2$ we add it to K. Note that this can be done in time proportional to the number of vertices in the SCCs in S of size at most $|S|/2$. The sorting operation at Line 15 takes time $O(|K| \log |K|)$ plus the size of all the SCCs in K, that is $\sum_{K_i \in K} |K_i|$. Note that $O(|K| \log |K|) = O((\sum_{K_i \in K} |K_i|) \log(\sum_{K_i \in K} |K_i|))$. Let $K_i \in K$ be an SCC stored in K. Note that during the algorithm each vertex can appear at most $\log(n)$ times in the list K. This is by the fact that K only contains SCCs that are at most half the size of the original set S. We obtain a running time bound of $O(n(\log n)^2)$ for Lines 12-15.

Consider the second for-loop at Line 17: Let $|X| = n_1$. The operations $\texttt{Remove}(\cdot)$ and $\texttt{Construct}(\cdot)$ are called once per found SCC $G[X]$ with $X \neq S$ and take by Lemma 3 $O(|X| + \mathit{bits}(X))$ time. Whenever a vertex is in X, the size of the set in Q containing v originally is reduced by at least a factor of two due to the fact that $|X| = n_1 \leq n'/2$. This happens at most $\lceil \log n \rceil$ times. By charging $O(1)$ to the vertices in X and, respectively, to $\mathit{bits}(X)$, the total running time for Lines 21 & 22 can be bounded by $O((n+b)\log n)$ as each vertex and bit is only charged $O(\log n)$ times. Combining all parts yields the claimed running time bound of $O(T_d + b \log n + n \log^2 n) = \widetilde{O}(m+b)$. ◀

The correctness of the algorithm is similar to the analysis given in [11, Lemmas 3.6 & 3.7] except that we additionally have to prove that $\texttt{SCCs}(S)$ holds the SCCs of $G[S]$. Lemma 5 shows that we maintain $\texttt{SCCs}(S)$ properly for all the data structures in Q.

▶ **Lemma 5.** *After each iteration of the outer while-loop every non-trivial SCC of the current graph is contained in one of the subgraphs $G[S]$ for which the data structure $D(S)$ is maintained in Q and $\texttt{SCCs}(S)$ stores a list of all SCCs contained in S.*

We prove the next Lemma by showing that we never remove edges of vertices of good components.

▶ **Lemma 6.** *After each iteration of the outer while-loop every good component of the input graph is contained in one of the subgraphs $G[S]$ for which the data structure $D(S)$ is maintained in the list Q.*

▶ **Proposition 7.** *Algorithm 1 outputs a good component if one exists, otherwise the algorithm reports that no such component exists.*

Proof. First consider the case where Algorithm 1 outputs a subgraph $G[S]$. We show that $G[S]$ is a good component: Line 10 ensures only non-trivial SCSs are considered. After the removal of bad vertices from S in Lines 6-9, we know that for all $1 \leq j \leq k$ that $U_j \cap S \neq \emptyset$ if $S \cap L_j \neq \emptyset$. Due to Line 13 there is only one SCC in $G[S]$ and thus $G[S]$ is a good component. Second, if Algorithm 1 terminates without a good component, by Lemma 6, we have that the initial graph has no good component and thus the result is correct as well. ◀

The running time bounds for the decremental SCC algorithm of [4] (cf. Corollary 2) only hold against an oblivious adversary. Thus we have to show that in our algorithm the sequence of edge deletions does not depend on the random choices of the decremental SCC algorithm. The key observation is that only the order of the computed SCCs depends on the random choices of the decremental SCC and we eliminate this effect by sorting the SCCs.

▶ **Proposition 8.** *The sequence of deleted edges does not depend on the random choices of the decremental SCC Algorithm but only on the given instance.*

Due to Lemma 4, Lemma 7 and Proposition 8 we obtain the following result.

▶ **Theorem 9.** *In a graph, the winning set for a k-pair Streett objective can be computed in $\widetilde{O}(m+b)$ expected time.*

5 Algorithms for MDPs

In this section, we present expected near-linear time algorithms for computing a MEC decomposition, deciding almost-sure reachability and maintaining a MEC decomposition in a decremental setting. In the last section, we present an algorithm for *MDPs* with Streett objectives by using the new algorithm for the decremental MEC decomposition.

Random attractor. First, we introduce the notion of a *random attractor* $\mathsf{attr}_{\mathsf{R}}(\mathsf{T})$ for a set $T \subseteq V$. The random attractor $A = \mathsf{attr}_{\mathsf{R}}(\mathsf{T})$ is defined inductively as follows: $A_0 = T$ and $A_{i+1} = A_i \cup \{v \in V_R \mid Out(v) \cap A_i \neq \emptyset\} \cup \{v \in V_1 \mid Out(v) \subseteq A_i\}$ for all $i > 0$. Given a set T, the random attractor includes all vertices (1) in T, (2) random vertices with an edge to A_i, (3) player-1 vertices with all outgoing edges in A_i. Due to [21, 3] we can compute the random attractor $A = \mathsf{attr}_{\mathsf{R}}(\mathsf{S})$ of a set S in time $O(\sum_{v \in A} In(v))$.

5.1 Maximal End-Component Decomposition

In this section, we present an expected near linear time algorithm for MEC decomposition. Our algorithm is an efficient implementation of the static algorithm presented in [9, p. 29]: The difference is that the bottom SCCs are computed with a dynamic SCC algorithm instead of recomputing the static SCC algorithm. A similar algorithm was independently proposed in an unpublished extended version of [12].

Algorithm Description. The MEC algorithm described in Algorithm 2 repeatedly removes bottom SCCs and the corresponding random attractor. After removing bottom SCCs the new SCC decomposition with its bottom SCCs is computed using a dynamic SCC algorithm.

Algorithm 2: MEC Algorithm.

Input: MDP $P = (V, E, \langle V_1, V_R \rangle, \delta)$, decremental SCC algorithm $\mathcal{A}$

1 Invoke an instance $\mathcal{A}$ of the decremental SCC algorithm;
2 Compute the SCC-decomposition of $G = (V, E)$: $C = \{C_1, \ldots, C_\ell\}$;
3 Let $M = \emptyset$; $Q \leftarrow \{C_i \in C \mid C_i \text{ has no outgoing edges}\}$;
4 **while** Q *is not empty* **do**
5 $\quad C \leftarrow \emptyset$;
6 $\quad$ **for** $C_k \in Q$ **do** $C \leftarrow C \cup C_k$; $M \leftarrow M \cup \{C_k\}$;
7 $\quad A \leftarrow \mathsf{attr}_{\mathsf{R}}(\mathsf{C})$;
8 $\quad Q \leftarrow \mathcal{A}.\mathsf{deleteAnnounceNoOutgoing}(E(A))$;
9 **return** M;

Correctness follows because our algorithm just removes attractors of bottom SCCs and marks bottom SCCs as MECs. This is precisely the second static algorithm presented in [9, p. 29] except that the bottom SCCs are computed using a dynamic data structure. By using the decremental SCC algorithm described in Subsection 3 we obtain the following lemma.

▶ **Lemma 10.** *Algorithm 2 returns the MEC-decomposition of an MDP P in expected time $\widetilde{O}(m)$.*

Proof. The running time of algorithm $\mathcal{A}$ is in total time $O(T_d) = \widetilde{O}(m)$ by Theorem 1 and Corollary 2. Initially, computing the SCC decomposition and determining the SCCs with no outgoing edges takes time $O(m + n)$ by using [27]. Each time we compute the attractor of a bottom SCC C_k at Line 7 we remove it from the graph by deleting all its edges and never process these edges and vertices again. Since we can compute the attractor A at Line 7 in time $O(\sum_{v \in A} In(A))$, we need $O(m + n)$ total time for computing the attractors of all bottom SCCs. Hence, the running time is dominated by the decremental SCC algorithm $\mathcal{A}$, which is $O(T_d) = \widetilde{O}(m)$. ◀

The algorithm uses $O(m + n)$ space due to the fact that the decremental SCC algorithm $\mathcal{A}$ uses $O(m + n)$ space and Q only contains vertices.

▶ **Theorem 11.** *Given an MDP the MEC-decomposition can be computed in $\widetilde{O}(m)$ expected time. The algorithm uses $O(m+n)$ space.*

Note that we can use the decremental SCC Algorithm $\mathcal{A}$ of [4] even though this algorithm only works against an oblivious adversary as the sequence of deleted edges does not depend on the random choices of the decremental SCC Algorithm.

5.2 Almost-Sure Reachability

In this section, we present an expected near linear-time algorithm for the almost-sure reachability problem. In the almost-sure reachability problem, we are given an MDP P and a target set T and we ask for which vertices player 1 has a strategy to reach T almost surely, i.e., $\langle\langle 1\rangle\rangle_{a.s.}(Reach(T))$. Due to [5, Theorem 4.1] we can determine the set $\langle\langle 1\rangle\rangle_{a.s.}(Reach(T))$ in time $O(m + \mathsf{MEC})$ where MEC is the running time of the fastest MEC algorithm. We use Theorem 11 to compute the MEC decomposition and obtain the following theorem.

▶ **Theorem 12.** *Given an MDP and a set of vertices T we can compute $\langle\langle 1\rangle\rangle_{a.s.}(Reach(T))$ in $\widetilde{O}(m)$ expected time.*

5.3 Decremental Maximal End-Component Decomposition

We present an expected near-linear time algorithm for the MEC-decomposition which supports player-1 edge deletions and a query that answers if two vertices are in the same MEC. We need the following lemma from [7] to prove the correctness of our algorithm. Given an SCC C we consider the set U of the random vertices in C with edges leaving C. The lemma states that for all non-trivial MECs X in P the intersection with U is empty, i.e., $\mathsf{attr}_{\mathsf{R}}(\mathsf{U}) \cap X = \emptyset$.

▶ **Lemma 13** (Lemma 2.1(1), [7])**.** *Let C be an SCC in P. Let $U = \{v \in C \cap V_R \mid E(v) \cap (V \setminus C) \neq \emptyset\}$ be the random vertices in C with edges leaving C. Let $Z = \mathsf{attr}_{\mathsf{R}}(\mathsf{U}) \cap C$. Then for all non-trivial MECs 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.*

The *pure MDP graph* P^P of an MDP $P = (V, E, \langle V_1, V_R\rangle, \delta)$ is the graph which contains only edges in non-trivial MECs of P. More formally, the pure MDP graph P^P is defined as follows: Let $M_1, \dots M_k$ be the set of MECs of P. $P^P = (V^P, E^P, \langle V_1^P, V_R^P\rangle, \delta^P)$ where $V^P = V, V_1^P = V_1, V_R^P = V_R$, $E^P = \bigcup_{i=1}^{k}\{(u, v) \in E \cap (M_i \times M_i)\}$ and for each $v \in V_R$: $\delta^P(v)$ the uniform distribution over vertices u with $(v, u) \in E^P$.

Throughout the algorithm, we maintain the pure MDP graph P^P for an input MDP P. Note that every non-trivial SCC in P^P is also a MEC due to the fact that there are only edges inside of MECs. Moreover, a trivial SCC $\{v\}$ is a MEC iff $v \in V_1$. Note furthermore that when a player-1 edge of an MDP P is deleted, existing MECs might split up into several MECs but no new vertices are added to existing MECs.

Initially, we compute the MEC-decomposition in $\widetilde{O}(m)$ expected time using the algorithm described in Section 5.1. Then we remove every edge that is not in a MEC. The resulting graph is the pure MDP graph P^P. Additionally, we invoke a decremental SCC algorithm $\mathcal{A}$ which is able to (1) announce new SCCs under edge deletions and return a list of their vertices and (2) is able to answer queries that ask whether two vertices v, u belong to the same SCC. When an edge (u, v) is deleted, we know that (i) the MEC-decomposition stays the same or (ii) one MEC splits up into new MECs and the rest of the decomposition stays the same. We first check if u and v are in the same MEC, i.e., if it exists in P^P. If not, we are done. Otherwise, u and v are in the same MEC C and either (1) the MEC C does

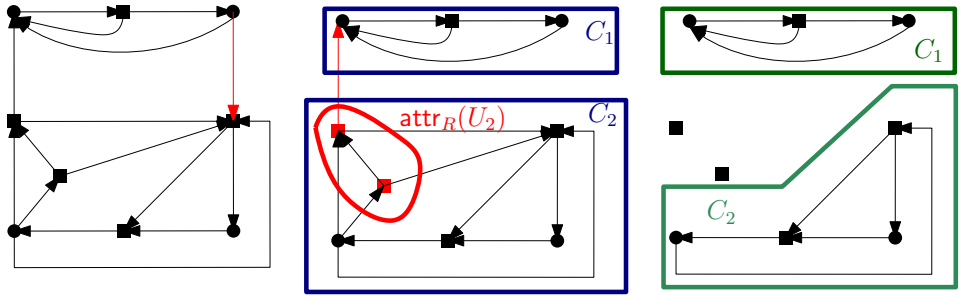

Figure 2 We delete an edge which splits the MEC into two new SCCs C_1 and C_2. The SCC C_2 is not a MEC. We thus compute and remove the attractor of U_2 and the resulting SCC is a MEC.

not split or (2) the MEC C splits. In the case of (1) the SCCs of the pure MDP graph P^P remain intact and nothing needs to be done. In the case of (2) we need to identify the new SCCs $C_1, \ldots, C_k$ in P^P using the decremental SCC algorithm $\mathcal{A}$. Let, w.l.o.g., C_1 be the SCC with the most vertices. We iterate through every edge of the vertices in the SCCs $C_2, \ldots, C_k$. By considering all the edges, we identify all SCCs (including C_1) which are also MECs. We remove all edges (y, z) where y and z are not in the same SCC to maintain the pure MDP graph P^P. For the SCCs that are not MECs let U be the set of random vertices with edges leaving its SCC. We compute and remove $A = \mathsf{attr}_R(U)$ (these vertices belong to no MEC due to Lemma 13) and recursively start the procedure on the new SCCs generated by the deletion of the attractor. The algorithm is illustrated in Figure 2.
Lemma 14 describes the key invariants of the while-loop at Line 3. We prove it with a straightforward induction on the number of iterations of the while-loop and apply Lemma 13.

▶ **Lemma 14.** *Assume that $\mathcal{A}$ maintains the pure MDP graph P^P before the deletion of $e = (u, v)$ then the while-loop at Line 3 maintains the following invariants:*

1. *For the graph stored in $\mathcal{A}$ and all lists of SCCs $\{C_1, \ldots, C_k\}$ in K there are only edges inside the SCCs or between the SCCs in the list, i.e., for each $(x, y) \in \bigcup_{j=0}^{k} E[C_j]$ we have $x, y \in \bigcup_{j=0}^{k} C_j$.*
2. *If a non-trivial SCC of the graph in $\mathcal{A}$ is not a MEC of the current MDP it is in K.*
3. *If M is a MEC of the current MDP then we do not delete an edge of M in the while-loop.*

▶ **Proposition 15.** *Algorithm 3 maintains the pure MDP graph P^P in the data structure $\mathcal{A}$ under player-1 edge deletions.*

Proof. We show that after deleting an edge using Algorithm 3 (i) every non-trivial SCC is a MEC and vice-versa, and (ii) there are no edges going from one MEC to another. Initially, we compute the pure MDP graph and both conditions are fulfilled.

When we delete an edge and the while-loop at Line 3 terminates (i) is true due to Lemma 14(2,3). That is, as we never delete edges within MECs they are still strongly connected and when the while-loop terminates, $K = \emptyset$ which means that all SCCs are MECs.

For (ii) notice that each SCC is once processed as a List J. Consider an arbitrary SCC C_i and the corresponding list of SCCs $J = \{C_1, \ldots, C_k\}$ of the iteration in which C_i was identified as a MEC. By Lemma 14(1) there are no edges to SCCs not in the list. Additionally, due to Line 10 we remove all edges from C_i to other SCCs in J. ◀

Now that we maintain the pure MDP graph P^P in $\mathcal{A}$, we can answer MEC queries of the form: $\mathsf{query}(u, v)$: Returns whether u and v are in the same MEC in P, by an SCC query $\mathcal{A}.\mathsf{query}(u, v)$ on the pure MDP graph P^P.

Algorithm 3: Decremental MEC-update.

Input: Player-1 Edge $e = (u, v)$
if $\mathcal{A}.\mathsf{query}(u, v) = true$ **then**
  List $K \leftarrow \{\mathcal{A}.\mathsf{deleteAnnounce}((u, v))\}$;
  while $K \neq \emptyset$ **do**
    pull a list J of SCCs from K and let C_1 be the largest SCC;
    $\{C_1, \dots C_k\} \leftarrow$ Sort all SCCs in J except C_1 by the smallest vertex id.;
    $\mathsf{MEC}^{C_1} =$ **True**, $U_1 \leftarrow \emptyset$;
    for $i = 2$; $i \leq k$; $i{+}{+}$ **do**
      $\mathsf{MEC}^{C_i} =$ **True**, $U_i \leftarrow \emptyset$;
      for $e = (s, t)$ *where* $e \in E(C_i)$ **do**
        if $(s \notin C_i) \vee (t \notin C_i)$ **then** $\mathcal{A}.\mathsf{delete}(e)$;
        if $(s \in V_R \wedge t \notin C_i)$ **then** $\mathsf{MEC}^{C_i} =$ **False**; $U_i \leftarrow U_i \cup \{s\}$;
        if $(s \in V_R \wedge s \in C_1)$ **then** $\mathsf{MEC}^{C_1} =$ **False**; $U_1 \leftarrow U_1 \cup \{s\}$;
      if $\mathsf{MEC}^{C_i} =$ **False then**
        $A \leftarrow \mathsf{attr}_{\mathsf{R}}(\mathsf{U_i}) \cap C_i$;
        $J \leftarrow \mathcal{A}.\mathsf{deleteAnnounce}(E(A)) \setminus \left(\bigcup_{a \in A} \mathcal{A}.\mathsf{rep}(a)\right)$;
        if $J \neq \emptyset$ **then** $K \leftarrow K \cup \{J\}$;
    if $\mathsf{MEC}^{C_1} =$ **False then**
      $A \leftarrow \mathsf{attr}_{\mathsf{R}}(\mathsf{U}_1) \cap C_1$;
      $J \leftarrow \mathcal{A}.\mathsf{deleteAnnounce}(E(A)) \setminus \left(\bigcup_{a \in A} \mathcal{A}.\mathsf{rep}(a)\right)$;
      if $J \neq \emptyset$ **then** $K \leftarrow K \cup \{J\}$;

The key idea for the running time of Algorithm 3 is that we do not look at edges of the largest SCCs but the new SCC decomposition by inspecting the edges of the smaller SCCs. Note that we identify the largest SCC by processing the SCCs in a lockstep manner. This can only happen $\lceil \log n \rceil$ times for each edge. Additionally, when we sort the SCCs, we only look at the vertex ids of the smaller SCCs and when we charge this cost to the vertices we need $O(n \log^2 n)$ additional time.

▶ **Proposition 16.** *Algorithm 3 maintains the MEC-decomposition of P under player-1 edge deletions in expected total time $\widetilde{O}(m)$. Algorithm 3 answers queries that ask whether two vertices v, u belong to the same MEC in $O(1)$. The algorithm uses $O(m + n)$ space.*

Due to the fact that the decremental SCC algorithm we use in Corollary 2 only works for an oblivious adversary, we prove the following proposition. The key idea is that we sort SCCs returned by the decremental SCC Algorithm. Thus, the order in which new SCCs are returned does only depend on the given instance.

▶ **Proposition 17.** *The sequence of deleted edges does not depend on the random choices of the decremental SCC Algorithm but only on the given instance.*

The algorithm presented in [4] fulfills all the conditions of Proposition 16 due to Corollary 2. Therefore we obtain the following theorem due to Proposition 15 and Proposition 16.

▶ **Theorem 18.** *Given an MDP with n vertices and m edges, the MEC-decomposition can be maintained under the deletion of $O(m)$ player-1 edges in total expected time $\widetilde{O}(m)$ and we can answer queries that ask whether two vertices v, u belong to the same MEC in $O(1)$ time. The algorithm uses $O(m + n)$ space. The bound holds against an oblivious adversary.*

5.4 MDPs with Streett Objectives

Similar to graphs we compute the winning region of Streett objectives with k pairs (L_i, U_i) (for $1 \leq i \leq k$) for an MDP P as follows:

1. We compute the MEC-decomposition of P.
2. For each MEC, we find good end-components, i.e., end-components where $L_i \cap X = \emptyset$ or $U_i \cap X \neq \emptyset$ for all $1 \leq i \leq k$ and label the MEC as satisfying.
3. We output the set of vertices that can almost-surely reach a satisfying MECs.

For 2., we find good *end-components* similar to how we find good components as in Section 4. The key idea is to use the decremental MEC-Algorithm described in Section 5.3 instead of the decremental SCC Algorithm. We modify the Algorithm presented in Section 4 as follows to detect good end-components: First, we use the decremental MEC-algorithm instead of the decremental SCC Algorithm. Towards this goal, we augment the decremental MEC-algorithm with a function to return a list of references to the new MECs when we delete a set of edges. Second, the decremental MEC-algorithm does not allow the deletion of arbitrary edges, but only player-1 edges. To overcome this obstacle, we create an equivalent instance where we remove player-1 edges when we remove "bad" vertices.

▶ **Lemma 19.** *Given an MDP $P = (V, E, \langle V_1, V_R \rangle, \delta)$ with m edges and n vertices, we can maintain a data structure that supports the operation*

- deleteAnnounce(E)*: Deletes the set of E of player-1 edges (u, v) from the MDP P. If the edge deletion creates new MECs $C_1, \ldots, C_k$ the operation returns a list $Q = \{C_1, \ldots, C_k\}$ of references to the new non-trivial MECs.*

in total expected update time $\widetilde{O}(m)$. The bound holds against an oblivious adaptive adversary.

Deleting bad vertices. As the decremental MEC-algorithm only allows deletion of player-1 edges, we first modify the original instance $P = (V, E, \langle V_1, V_R \rangle, \delta)$ to a new instance $P' = (V', E', \langle V_1', V_R' \rangle, \delta')$ such that we can remove bad vertices by deleting player-1 edges only. In P' each vertex $v \in V_x$ for $x \in \{1, R\}$ is split into two vertices $v_{in} \in V_1'$ and $v_{out} \in V_x'$ such that $E' = \{(u_{out}, v_{in}) \mid (u, v) \in E\} \cup \{(v_{in}, v_{out}) \mid v \in V\}$ and $L_i' = \{v_{in} \in V' \mid v \in L_i\}$ and $U_i' = \{v_{out} \in V' \mid v \in U_i\}$ for all $1 \leq i \leq k$. The new probability distribution is $\delta'(v_{out})[w_{in}] = \delta(v)[w]$ for $v \in V_R$ and $w \in \mathit{Out}(v)$. Note that for each $v \in V_R$ the corresponding vertex $v_{out} \in V_R'$ has the same probabilities to reach the representation v_{out} of a vertex as v. The described reduction allows us to remove bad vertices from MECs by removing the player-1 edge (v_{in}, v_{out}).

The key idea for the following lemma is that for each original vertex $v \in V$ either both v_{in} and v_{out} are part of a good end-component or none of them. Note that the only way that v_{in} and v_{out} are strongly connected is when the other vertex is also in the strongly connected component because v_{in} (v_{out}) has only one outgoing (incoming) edges to v_{out} (from v_{in}).

▶ **Lemma 20.** *There is a good end-component in the modified instance P' iff there is a good component in the original instance P.*

On the modified instance P' the algorithm for MDPs is identical to Algorithm 1 except that we use a dynamic MEC algorithm instead of a dynamic SCC algorithm.

▶ **Theorem 21.** *In an MDP the winning set for a k-pair Street objectives can be computed in $\widetilde{O}(m + b)$ expected time.*

References

1 R. Alur and T.A. Henzinger. Computer-Aided Verification. unpublished., 2004. URL: https://web.archive.org/web/20041207121830/http://www.cis.upenn.edu/group/cis673/.

2 C. Baier and J.P. Katoen. *Principles of model checking*. MIT Press, 2008.

3 C. Beeri. On the Membership Problem for Functional and Multivalued Dependencies in Relational Databases. *ACM Trans. Database Syst.*, 5(3):241–259, 1980. doi:10.1145/320613.320614.

4 A. Bernstein, M. Probst, and C. Wulff-Nilsen. Decremental Strongly-Connected Components and Single-Source Reachability in Near-Linear Time. In *STOC*, pages 365–376, 2019. doi:10.1145/3313276.3316335.

5 K. Chatterjee, W. Dvořák, M. Henzinger, and V. Loitzenbauer. Model and Objective Separation with Conditional Lower Bounds: Disjunction is Harder than Conjunction. In *LICS*, pages 197–206, 2016. doi:10.1145/2933575.2935304.

6 K. Chatterjee, A. Gaiser, and J. Kretínský. Automata with Generalized Rabin Pairs for Probabilistic Model Checking and LTL Synthesis. In *CAV*, pages 559–575, 2013. doi:10.1007/978-3-642-39799-8_37.

7 K. Chatterjee and M. Henzinger. Faster and Dynamic Algorithms for Maximal End-Component Decomposition and Related Graph Problems in Probabilistic Verification. In *SODA*, pages 1318–1336, 2011. doi:10.1137/1.9781611973082.101.

8 K. Chatterjee and M. Henzinger. An $O(n^2)$ Time Algorithm for Alternating Büchi Games. In *SODA*, pages 1386–1399, 2012. URL: http://portal.acm.org/citation.cfm?id=2095225&CFID=63838676&CFTOKEN=79617016.

9 K. Chatterjee and M. Henzinger. Efficient and Dynamic Algorithms for Alternating Büchi Games and Maximal End-Component Decomposition. *J. ACM*, 61(3):15.1–15.40, 2014. doi:10.1145/2597631.

10 K. Chatterjee, M. Henzinger, and V. Loitzenbauer. Improved Algorithms for One-Pair and k-Pair Streett Objectives. In *LICS*, pages 269–280, 2015. doi:10.1109/LICS.2015.34.

11 K. Chatterjee, M. Henzinger, and V. Loitzenbauer. Improved Algorithms for Parity and Streett objectives. *Logical Methods in Computer Science*, 13(3):1–27, 2017. doi:10.23638/LMCS-13(3:26)2017.

12 S. Chechik, T. D. Hansen, G. F. Italiano, J. Lacki, and N. Parotsidis. Decremental Single-Source Reachability and Strongly Connected Components in $\tilde{O}(m\sqrt{n})$ Total Update Time. In *FOCS*, pages 315–324, 2016. doi:10.1109/FOCS.2016.42.

13 F. Ciesinski and C. Baier. LiQuor: A Tool for Qualitative and Quantitative Linear Time Analysis of Reactive Systems. In *QEST*, pages 131–132, 2006. doi:10.1109/QEST.2006.25.

14 A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NUSMV: A new Symbolic Model Checker. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4):410–425, 2000. doi:10.1007/s100090050046.

15 E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

16 C. Courcoubetis and M. Yannakakis. The Complexity of Probabilistic Verification. *J. ACM*, 42(4):857–907, 1995. doi:10.1145/210332.210339.

17 C. Dehnert, S. Junges, J.P. Katoen, and M. Volk. A Storm is Coming: A Modern Probabilistic Model Checker. In *CAV*, pages 592–600, 2017. doi:10.1007/978-3-319-63390-9_31.

18 J. Esparza and J. Kretínský. From LTL to Deterministic Automata: A Safraless Compositional Approach. In *CAV*, pages 192–208, 2014. doi:10.1007/978-3-319-08867-9_13.

19 M. Rauch Henzinger and J. A. Telle. Faster Algorithms for the Nonemptiness of Streett Automata and for Communication Protocol Pruning. In *SWAT*, pages 16–27, 1996. doi:10.1007/3-540-61422-2_117.

20 G. J. Holzmann. The Model Checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997. doi:10.1109/32.588521.

21 N. Immerman. Number of Quantifiers is Better Than Number of Tape Cells. *J. Comput. Syst. Sci.*, 22(3):384–406, 1981. doi:10.1016/0022-0000(81)90039-8.

22 Z. Komárková and J. Kretínský. Rabinizer 3: Safraless Translation of LTL to Small Deterministic Automata. In *ATVA*, pages 235–241, 2014. doi:10.1007/978-3-319-11936-6_17.

23 M. Z. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of Probabilistic Real-Time Systems. In *CAV*, pages 585–591, 2011. doi:10.1007/978-3-642-22110-1_47.

24 Z. Manna and A. Pnueli. Temporal Verification of Reactive Systems: Progress (Draft), 1996. URL: http://theory.stanford.edu/~zm/tvors3.html.

25 S. Safra. On the Complexity of ω-Automata. In *FOCS*, pages 319–327, 1988. doi:10.1109/SFCS.1988.21948.

26 R. Segala. *Modeling and Verification of Randomized Distributed Real-Time Systems*. PhD thesis, MIT, 1995.

27 R. Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

28 M. Y. Vardi. Automatic Verification of Probabilistic Concurrent Finite-State Programs. In *FOCS*, pages 327–338, 1985. doi:10.1109/SFCS.1985.12.