Efficient and Dynamic Algorithms for Alternating Büchi Games and Maximal End-component Decomposition

Krishnendu Chatterjee, IST Austria (Institute of Science and Technology Austria)
Monika Henzinger, Faculty of Computer Science, University of Vienna, Austria

The computation of the winning set for Büchi objectives in alternating games on graphs is a central problem in computer aided verification with a large number of applications. The long standing best known upper bound for solving the problem is $\widetilde{O}(n \cdot m)$, where n is the number of vertices and m is the number of edges in the graph. We are the first to break the $\widetilde{O}(n \cdot m)$ boundary by presenting a new technique that reduces the running time to $O(n^2)$. This bound also leads to $O(n^2)$-time algorithms for computing the set of almost-sure winning vertices for Büchi objectives (1) in alternating games with probabilistic transitions (improving an earlier bound of $\widetilde{O}(n \cdot m)$), (2) in concurrent graph games with constant actions (improving an earlier bound of $O(n^3)$), and (3) in Markov decision processes (improving for $m > n^{4/3}$ an earlier bound of $O(m \cdot \sqrt{m})$). We then show how to maintain the winning set for Büchi objectives in alternating games under a sequence of edge insertions or a sequence of edge deletions in $O(n)$ amortized time per operation. Our algorithms are the first dynamic algorithms for this problem. We then consider another core graph theoretic problem in verification of probabilistic systems, namely computing the maximal end-component decomposition of a graph. We present two improved static algorithms for the maximal end-component decomposition problem. Our first algorithm is an $O(m \cdot \sqrt{m})$-time algorithm, and our second algorithm is an $O(n^2)$-time algorithm which is obtained using the same technique as for alternating Büchi games. Thus we obtain an $O(\min\{m \cdot \sqrt{m}, n^2\})$-time algorithm improving the long-standing $O(n \cdot m)$ time bound. Finally, we show how to maintain the maximal end-component decomposition of a graph under a sequence of edge insertions or a sequence of edge deletions in $O(n)$ amortized time per edge deletion, and $O(m)$ worst case time per edge insertion. Again, our algorithms are the first dynamic algorithms for this problem.

Categories and Subject Descriptors: Theory of Computation [**Analysis of Algorithms and Problem Complexity**]: Computations on discrete structures

General Terms: Algorithms (Graph Algorithms and data structures); Verification (Computer-aided verification)

Additional Key Words and Phrases: Graph games; Büchi objectives; Graph algorithms; Dynamic graph algorithms; Verification and synthesis.

ACM Reference Format:
J. ACM V, N, Article A (January YYYY), 36 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

We consider two fundamental algorithmic problems that lie in the core of many applications in formal verification and analysis of systems. The two problems are alternating games with Büchi objectives, and maximal end-component decomposition of Markov decision processes (MDPs). We will present graph theoretic description of both the problems. In this work we present faster static algorithms for both the problems improving the long-standing upper bounds, and the first dynamic algorithms for both problems.

Alternating Büchi games. Consider a finite directed graph (V, E) with a partition (V_1, V_2) of V and a set $B \subset V$ of *Büchi* vertices. This graph is called a *game graph*. Let $n = |V|$ and $m = |E|$. Two players play the following *alternating game* on the graph that forms an infinite path. They

This article is a combined and improved version of the conference papers [Chatterjee and Henzinger 2011; 2012].

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

start by placing a token on an initial vertex and then take turns indefinitely in moving the token: At a vertex $v \in V_1$ player 1 moves the token along one of the outedges of v, at a vertex $u \in V_2$ player 2 moves the token along one of the outedges of u. A first question to ask is given a start vertex $x \in V$ can player 1 guarantee that the infinite path visits a vertex in B *at least once*, no matter what choices player 2 makes. If so player 1 can *win* from x and x belongs to the *winning set of player 1*. The question of computing the set of vertices from which player 1 can win (called the *winning set*) is called the *(alternating) reachability game problem*. The problem is PTIME-complete and the winning set of player 1 can be computed in time linear in the size of the graph [Beeri 1980; Immerman 1981]. A second, more central question is whether player 1 can guarantee that the infinite path visits a vertex in B *infinitely often*, no matter what choices player 2 makes. The computation of the winning set of player 1 for this setting is called the *(alternating) Büchi game problem*. The best known algorithms for this problem are algorithms that repeatedly compute the alternating reachability game solution on the graph after the removal of specific vertices. Their running time is $\widetilde{O}(n \cdot m)$, where we denote by $\widetilde{O}(f) = O(f/\log(f))$ (i.e., to omit $\log$-factors). We present in this paper a new algorithmic technique for the alternating Büchi game problem which is inspired by dynamic graph algorithms and reduces the running time to $O(n^2)$.

Büchi games: applications and significance. Two-player games on graphs played by player 1 and the adversary player 2 are central in many problems in computer science, specially in verification and synthesis of systems such as the synthesis of systems from specifications and synthesis of reactive systems [Church 1962; Pnueli and Rosner 1989; Ramadge and Wonham 1987], verification of open systems [Alur et al. 2002], checking interface compatibility [de Alfaro and Henzinger 2001], well-formedness of specifications [Dill 1989], and many others. Besides their application in verification, they have also been studied in artifical intelligence as AND-OR graphs [Mahanti and Bagchi 1985], and in the context of alternating Turing machines [Chandra et al. 1981]. The class of Büchi or repeated reachability objectives was introduced in the seminal works of Büchi [Büchi 1960; 1962; Büchi and Landweber 1969] in the context of automata over infinite words. The alternating Büchi game problem has many applications in relation to synthesis, verification and automata theory. For example, (a) the solution of the synthesis problem for deterministic Büchi automata is achieved through solving the alternating Büchi game problem (see [Kupferman and Vardi 2005] for the importance of deterministic Büchi automata); and (b) the verification of open systems with liveness and weak fairness conditions (two key specifications used in verification) is again solved through the alternating Büchi game problem [Alur et al. 2002]. Vardi [Vardi 2007b; 2007a] discusses further applications of the alternating Büchi game problem and its importance. We mention a few applications of alternating Büchi games, and its relation to logic and automata theory to highlight its significance.

(1) *(Protocol synthesis).* In verification, after safety and reachability conditions, the most widely used condition is liveness (or weak-fairness) that corresponds to Büchi objectives. For example, the progress condition in mutual exclusion protocols (that specifies that it should always hold that if there is a request to the criticial section by a process, then the process eventually enters the critical section) is a liveness condition. The synthesis of mutual exclusion protocols reduces to solving alternating Büchi games [Chatterjee and Henzinger 2007a]. Moreover, recent works for synthesis of fair non-repudiation protocols (a class of security protocols) also only requires liveness objectives and are synthesized through solution of alternating Büchi games [Chatterjee and Raman 2012].

(2) *(Automata and LTL synthesis).* Though deterministic Büchi word automata (DBW) are not ω-regular complete, there are several results of wide interest related to the importance of properties in verification expressed as a DBW. In particular, DBW can express many important and most practically relevant fragments of linear-time temporal logic (LTL), the de-facto logic to specify properties in verification [Kupferman and Vardi 2005; 1998; Alur and Torre 2004; Krishnan et al. 1994]. We mention a few of them below:

(a) It was shown in [Kupferman and Vardi 2005] that a translation from LTL to DBW whenever possible is much simpler than using complicated determinization of ω-regular automata. Thus the fragment of LTL that can be translated to DBW can be translated efficiently. Moreover, it was shown in [Kupferman and Vardi 1998] that an important fragment of linear-time μ-calculus (a formalism to specify properties in verification as fix-point formulas), namely, linear-time AFMC (alternation free μ-calculus) exactly corresponds to DBW.

(b) There are many other relevant fragments of LTL that are used in practice and can be translated to DBW. For example, four subclasses of LTL (with always and eventually operators) were introduced in [Alur and Torre 2004] and it was shown that all of them can be translated to DBW. Another fragment of LTL that can be translated to DBW was presented in [Krishnan et al. 1994]. It was shown in all these works that the fragments proposed cover a large set of properties that are actually used in verification.

(c) Finally, currently the popular fragment of LTL that is used in specifying properties for synthesis is called the GR(1) (generalized reactivity (1)) fragment [Piterman et al. 2006]. A huge fragment of GR(1) properties for synthesis reduces to conjunction of Büchi objectives. GR(1) properties consist of a conjunction of assumptions and a conjunction of guarantees. If the guarantees are safety properties only (or the assumptions are safety properties only), then GR(1) synthesis reduces to solving alternating games with conjunction of Büchi objectives. Morevoer, in several practical examples of synthesis the properties used satisfy that the guarantees are safety properties. The most prominent example of synthesis of GR(1) properties used in industrial example is the synthesis of AMBA AHB protocol [Bloem et al. 2007; Godhal et al. 2011]. In the specifications for AMBA AHB Master and AMBA AHB Slave the guarantees are either safety properties, or safety with next or until upto 3 steps (all of which are safety properties) [Godhal et al. 2011]. The synthesis problem for all these properties are reduced to alternating games with conjunction of Büchi objectives; and alternating games with n vertices, m edges, and conjunction of k Büchi objectives reduces to solving alternating Büchi games with $k \cdot n$ vertices and $k \cdot m$ edges.

In summary, many important properties in verification, most practically relevant subclasses of standard logics (such as LTL and μ-calculus) can be translated to DBW, and practical examples of specifications used in synthesis are Büchi objectives. Thus alternating Büchi games are of wide interest and significance to the verification, synthesis and temporal logic community.

Büchi games: previous results. The alternating Büchi game problem is one of the core problems in verification and synthesis (as highlighted in the above discussion). The classical algorithm for alternating Büchi games follows from the results of [Emerson and Jutla 1991; McNaughton 1993; Zielonka 1998], its complexity is $O(n \cdot m)$. The algorithm was improved in the special case of game graphs with $m = O(n)$ to $O(n^2/\log n)$ time in [Chatterjee et al. 2003]. A generalization of the algorithm from [Chatterjee et al. 2003] was presented in [Chatterjee et al. 2006], and the new algorithm requires $O((n \cdot m \cdot \log \Delta)/\log n)$ time, where Δ is the maximum outdegree. Thus the long standing best known upper bound for solving the alternating Büchi game problem is $\widetilde{O}(n \cdot m)$.

Motivation for dynamic algorithms. In the design and verification of open systems it is natural that the systems under verification are developed incrementally by adding choices or removing choices for the system, which is represented by player 1. However the adversary, modeled by player 2, is the environment, and the system design has no control over the environment actions. Hence there is a clear motivation to obtain dynamic algorithms for the alternating Büchi game problem, when edges leaving player-1 vertices are inserted or deleted, while edges leaving player-2 vertices remain unchanged.

Maximal end-component decomposition problem. The standard mathematical model in the analysis of probabilistic systems are called *Markov decision processes (MDPs)*, that exhibit both nondeterministic and probabilistic behavior [Howard 1960; Courcoubetis and Yannakakis 1995]. We first present a graph problem that lies at the core of many algorithms in the analysis of MDPs and

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 (with $V_P = \emptyset$) and closed recurrent sets for Markov chains (with $V_1 = \emptyset$).

MDPs: applications and previous results. In probabilistic verification, systems are frequently modeled as Markov decision processes. As described below, MDPs are a generalization of graphs. The generalization is needed to model two different kind of "behaviors" at vertices [Howard 1960]. 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 outedge to follow, and the vertices in V_P, that are vertices where the outedge is chosen 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 [Filar and Vrieze 1997], concurrent probabilistic systems [Courcoubetis and Yannakakis 1995], probabilistic systems operating in open environments [Segala 1995], and under-specified probabilistic systems [Bianco and de Alfaro 1995]. For instance, MDPs are the formal model to analyze systems with randomized embedded schedulers [de Alfaro et al. 2005], or analyze correctness of randomized distributed algorithms (see, e.g., [Pogosyants et al. 2000; Kwiatkowska et al. 2000; Stoelinga 2002]). Thus MDPs with ω-regular specifications (that can express all commonly used properties in verification) are at the heart of most problems in probabilistic verification. The maximal-end component decomposition problem is the graph algorithmic problem required to solve MDPs with ω-regular specifications [Courcoubetis and Yannakakis 1995; de Alfaro 1997]. In addition, several algorithms for analysis of MDPs with quantitative objectives such as $\limsup$ and $\liminf$ objectives [Chatterjee and Henzinger 2007b], and combination of mean-payoff and parity objectives [Chatterjee et al. 2010], or multi-objective optimization in MDPs [Etessami et al. 2008; Brázdil et al. 2011] rely crucially on the maximal end-component decomposition problem. The previous best known bound to compute the maximal end-component decomposition of MDPs is $O(n \cdot m)$ [Courcoubetis and Yannakakis 1995; de Alfaro 1997].

Motivation for dynamic algorithms. As in the case of open systems, 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 the maximal end-component decomposition problem 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.

[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 $\delta(u)(v) > 0$.

Our contributions. In this work we present improved static and the first dynamic algorithms for the alternating Büchi game problem and the maximal end-component decomposition problem using graph algorithmic techniques. Our main results are as follows.

(1) *Alternating Büchi games.* Our results for alternating Büchi games are as follows:
   (a) *Improved static algorithm.* We present an $O(n^2)$ time algorithm for the alternating Büchi game problem, and thus break the long standing barrier of $\widetilde{O}(n \cdot m)$ for the problem. It follows that along with the $O(n^2/\log n)$ algorithm for $m = O(n)$ [Chatterjee et al. 2003], the $O(n \cdot m)$ barrier is now broken for all cases.
   (b) *First dynamic algorithms.* We present the first incremental and decremental algorithms for the alternating Büchi game problem for insertion and deletion of player-1 edges. Our algorithm is based on the progress measure algorithm of [Jurdziński 2000] and generalizes the Even-Shiloach algorithm for decremental reachability in undirected graphs [Even and Shiloach 1981]. The total time for all operations is $O(n \cdot m)$, i.e., the amortized time per operation is $O(n)$. Our correctness proof is an elegant fix-point based argument, and to the best of our knowledge such fix-point based arguments for correctness have not been used for dynamic graph algorithms.

(2) *Maximal end-component decomposition.* Our results for maximal end-component decomposition are as follows:
   (a) *Improved static algorithms.* We present two improved static algorithms for the problem. Our first improved algorithm requires $O(m \cdot \sqrt{m})$ time. Using our technique to solve alternating Büchi games, we also present an $O(n^2)$-time algorithm for the problem in MDPs. Thus we obtain an $O(\min\{m \cdot \sqrt{m}, n^2\})$-time algorithm for the problem, and hence the problem can be solved in $O(m \cdot n^{2/3})$ time, improving the $O(m \cdot n)$ bound from 1995 [Courcoubetis and Yannakakis 1995; de Alfaro 1997]. This is the first algorithm that breaks the $O(m \cdot n)$ barrier for the problem.
   (b) *First dynamic algorithms.* We show how to maintain the maximal end-component 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 (amortized $O(n)$ per operation), whereas for the incremental case we give a worst case bound (worst case $O(m)$ per operation). Note that the problem of maintaining a maximal end-component decomposition generalizes the problem of maintaining a scc decomposition, and our results match the best known bounds for incremental and decremental scc decomposition.

Our main results are shown in Table I. Our results for alternating Büchi games and maximal end-decomposition improve the bounds for additional problems that we list next.

(1) The problem of computing the set of almost-sure (or probability 1) winning vertices in alternating games with probabilistic transitions (aka simple stochastic games [Condon 1992]) and Büchi objectives can be solved in $O(n^2)$ time improving the previous known $\widetilde{O}(n \cdot m)$ bound: this follows from the linear reduction of [Chatterjee et al. 2004] from simple stochastic games to alternating Büchi games for almost-sure winning and our Büchi games algorithm.

(2) The problem of computing the set of almost-sure (probability 1) and limit-sure (probability arbitrarily close to 1) winning vertices in concurrent graph games (aka games with simultaneous interaction) with constant actions with Büchi objectives can be solved in $O(n^2)$ time: this follows from the linear reduction from concurrent games to alternating Büchi games [Jurdziński et al. 2002] and our Büchi games algorithm. The best known bound for concurrent graph games with constant actions with Büchi objectives was $O(n \cdot |\delta|)$, where $|\delta|$ is the number of transitions which is $O(n^2)$ in the worst case. Thus, in the worst case the previous best known bound was $O(n^3)$.

(3) As a consequence of our $O(n^2)$ algorithm for Büchi games and the linear reduction of [Chatterjee et al. 2004], we also obtain an $O(n^2)$ algorithm for computing almost-sure winning states for MDPs with Büchi objectives. The best known bound for this problem was $O(m \cdot \sqrt{m})$ [Chat-

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

	Previous Algorithm	Our Algorithm	Incremental	Decremental
Alt. Büchi games	$O(m \cdot n)$ $O(\frac{n \cdot m \cdot \log(\Delta)}{\log(n)})$ Δ is max-degree	$\mathbf{O(n^2)}$	$\mathbf{O(n)}$ (Amortized)	$\mathbf{O(n)}$ (Amortized)
Max. end-component	$O(m \cdot n)$	$\mathbf{O(\min\{m \cdot \sqrt{m}, n^2\})}$	$\mathbf{O(m)}$ (Worst-case)	$\mathbf{O(n)}$ (Amortized)

terjee et al. 2003]. Thus we obtain an $O(\min\{m \cdot \sqrt{m}, n^2\})$-time (hence $O(m \cdot n^{2/3})$-time) algorithm for the problem. Thus, our algorithm is faster than the previous result for $m > n^{4/3}$.

(4) We showed in [Chatterjee and Henzinger 2011] that the almost-sure winning vertices in MDPs with parity objectives (a canonical form to express all ω-regular objectives) can be computed using $\log(d)$ calls to a maximal end-component decomposition algorithm and one call to compute almost-sure winning vertices for reachability objectives (which can be treated a special case of Büchi objectives), where d is the number of priorities (or parities) of the parity objective. Thus it follows from our results that MDPs with parity objectives can be solved in time $O(\log(d) \cdot \min\{m \cdot \sqrt{m}, n^2\})$.

Our main technical contributions for alternating Büchi games are as follows: (1) The classical algorithm for alternating Büchi games repeatedly removes *non-winning* vertices from the game graph and then recomputes the player-1 winning set for the alternating reachability game problem. Similar to the classical algorithm our algorithm repeatedly removes non-winning vertices from the game graph. However, it finds these vertices more efficiently using a hierarchical graph decomposition technique. This technique was used first by Henzinger et al. [Henzinger et al. 1999] for processing repeated edge deletions in undirected graphs. We show how this technique can be extended to work for vertex deletions in (directed) game graphs. As a result we achieve faster algorithms for the alternating Büchi game problem and for computing the maximal end-component decomposition. Moreover, even in sparse graphs, our technique can be useful. If $m = c \cdot n$ and c is a large constant, then our hiercharical decomposition can be used with a small number of levels, such as 2 or 3, to speed up the algorithm in practice. (2) Even and Shiloach [Even and Shiloach 1981] gave a deletions-only algorithm for maintaining reachability in undirected graphs. We show how to extend this algorithm to edge deletions in directed game graphs. A purely graph-theoretic proof of the correctness of the new algorithm would be lengthy. However, by using an elegant argument based on fix-points we give a simple proof of the correctness and an analysis of the running time of the new algorithm. The new algorithm is simple and, like the algorithm in [Even and Shiloach 1981], does not need any sophisticated data structures. We use a "dual" fix-point argument to construct an incremental algorithm for alternating Büchi games.

Our main technical contributions for maximal end-component decompositions 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 [Tarjan 1972] 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. Our second improved static algorithm uses the same hierarchical graph decomposition technique as our algorithm for Büchi games. (2) Our result for dynamic algorithms is obtained by combining results for dynamic algorithms for scc decomposition and the analysis of the previous known static maximal end-component decomposition algorithm.

The paper is organized as follows: In Section 2 we present all the results for alternating Büchi games and in Section 3 we present the results for maximal end-component decomposition.

2. ALGORITHMS FOR BÜCHI GAMES

In this section we will present improved static and the first dynamic algorithms for alternating Büchi games. We start with the basic definitions and preliminaries required.

2.1. Definitions

We consider *alternating graph games* played by two-players with Büchi (liveness or repeated reachability) and the complementary coBüchi objectives for the players, respectively. We define game graphs, plays, strategies, objectives and the notion of winning below.

Alternating game graphs. An *(alternating) game graph* $G = ((V, E), (V_1, V_2))$ consists of a directed graph (V, E) with a set V of n vertices and a set E of m edges, and a partition (V_1, V_2) of V into two sets. The vertices in V_1 are *player 1 vertices*, where player 1 chooses the outgoing edges, and the vertices in V_2 are *player 2 vertices*, where player 2 (the adversary to player 1) chooses the outgoing edges. Intuitively alternating game graphs are the same as AND-OR graphs. For a vertex $u \in V$, we write $\mathsf{O}ut(u) = \{v \in V \mid (u, v) \in E\}$ for the set of successor vertices of u and $\mathsf{I}n(u) = \{v \in V \mid (v, u) \in E\}$ for the set of incoming edges of u. We denote by $outdeg(u) = |\mathsf{O}ut(u)|$ the number of outgoing edges from u, and by $indeg(u) = |\mathsf{I}n(u)|$ the number of incoming edges. We assume that every vertex has at least one outgoing edge. i.e., $\mathsf{O}ut(u)$ is non-empty for all vertices $u \in V$.

Plays. A game is played by two players: player 1 and player 2, who form an infinite path in the game graph by moving a token along edges. They start by placing the token on an initial vertex, and then they take moves indefinitely in the following way. If the token is on a vertex in V_1, then player 1 moves the token along one of the edges going out of the vertex. If the token is on a vertex in V_2, then player 2 does likewise. The result is an infinite path in the game graph, called *plays*. Formally, a *play* is an infinite sequence $\langle v_0, v_1, v_2, \ldots \rangle$ of vertices such that $(v_k, v_{k+1}) \in E$ for all $k \geq 0$. We write Ω for the set of all plays.

Strategies. A strategy for a player is a rule that specifies how to extend plays. Formally, a *strategy* σ for player 1 is a function $\sigma\colon V^* \cdot V_1 \to V$ that, given a finite sequence of vertices (representing the history of the play so far) which ends in a player 1 vertex, chooses the next vertex. The strategy must choose only available successors, i.e., for all $w \in V^*$ and $v \in V_1$ we have $\sigma(w \cdot v) \in \mathsf{O}ut(v)$. The strategies for player 2 are defined analogously. A strategy is *memoryless* if it is independent of the history and only depends on the current vertex. Formally, a memoryless strategy for player 1 is a function $\sigma\colon V_1 \to V$ such that $\sigma(v) \in \mathsf{O}ut(v)$ for all $v \in V_1$, and analogously for player 2 strategies. We write Σ and Π for the sets of all strategies for player 1 and player 2, respectively. Given a starting vertex $v \in V$, a strategy $\sigma \in \Sigma$ for player 1, and a strategy $\pi \in \Pi$ for player 2, there is a unique play, denoted $\omega(v, \sigma, \pi) = \langle v_0, v_1, v_2, \ldots \rangle$, which is defined as follows: $v_0 = v$ and for all $k \geq 0$, if $v_k \in V_1$, then $\sigma(\langle v_0, v_1, \ldots, v_k \rangle) = v_{k+1}$, and if $v_k \in V_2$, then $\pi(\langle v_0, v_1, \ldots, v_k \rangle) = v_{k+1}$.

Objectives. An *objective* $\Phi \subseteq \Omega$ is a subset of plays, i.e., objectives describe the set of winning plays. We consider game graphs with a Büchi objective for player 1 and the complementary coBüchi objective for player 2. For a play $\omega = \langle v_0, v_1, v_2, \ldots \rangle \in \Omega$, we define $\mathrm{Inf}(\omega) = \{v \in V \mid v_k = v \text{ for infinitely many } k \geq 0\}$ to be the set of vertices that occur infinitely often in ω. We also define reachability and safety objectives as they will be useful in the analysis of the algorithms.

(1) *Reachability and safety objectives.* Given a set $T \subseteq V$ of vertices, the reachability objective $\mathrm{Reach}(T)$ requires that some vertex in T be visited, and dually, the safety objective $\mathrm{Safe}(F)$ requires that only vertices in F be visited. Formally, the sets of winning plays are $\mathrm{Reach}(T) = \{\langle v_0, v_1, v_2, \ldots \rangle \in \Omega \mid \exists k \geq 0.\ v_k \in T\}$ and $\mathrm{Safe}(F) = \{\langle v_0, v_1, v_2, \ldots \rangle \in \Omega \mid \forall k \geq 0.\ v_k \in F\}$. The reachability and safety objectives are dual in the sense that $\mathrm{Reach}(T) = \Omega \setminus \mathrm{Safe}(V \setminus T)$.

(2) *Büchi and coBüchi objectives.* Given a set $B \subseteq V$ of vertices, the Büchi objective $\mathrm{Buchi}(B)$ requires that some vertex in B be visited infinitely often, and dually, the coBüchi objective $\mathrm{coBuchi}(C)$ requires that only vertices in C be visited infinitely often. Thus, the sets of winning plays are $\mathrm{Buchi}(B) = \{\omega \in \Omega \mid \mathrm{Inf}(\omega) \cap B \neq \emptyset\}$ and $\mathrm{coBuchi}(C) = \{\omega \in \Omega \mid \mathrm{Inf}(\omega) \subseteq C\}$. The Büchi and coBüchi objectives are dual in the sense that $\mathrm{Buchi}(B) = \Omega \setminus \mathrm{coBuchi}(V \setminus B)$. Observe that Büchi and coBüchi objectives are *tail (or prefix-independent)* objectives, i.e., a play satisfies the objective if and only if the play obtained by adding or deleting a finite prefix also satisfies the objective.

Winning strategies and sets. Given an objective $\Phi \subseteq \Omega$ for player 1, a strategy $\sigma \in \Sigma$ is a *winning strategy* for player 1 from a vertex v if for all player 2 strategies $\pi \in \Pi$ the play $\omega(v, \sigma, \pi)$ is winning, i.e., $\omega(v, \sigma, \pi) \in \Phi$. The winning strategies for player 2 are defined analogously by switching the role of player 1 and player 2 in the above definition. A vertex $v \in V$ is winning for player 1 with respect to the objective Φ if player 1 has a winning strategy from v. Formally, the set of *winning vertices for player 1* with respect to the objective Φ is $W_1(\Phi) = \{v \in V \mid \exists \sigma \in \Sigma.\ \forall \pi \in \Pi.\ \omega(v, \sigma, \pi) \in \Phi\}$ the set of all winning vertices. Analogously, the set of all winning vertices for player 2 with respect to an objective $\Psi \subseteq \Omega$ is $W_2(\Psi) = \{v \in V \mid \exists \pi \in \Pi.\ \forall \sigma \in \Sigma.\ \omega(v, \sigma, \pi) \in \Psi\}$.

THEOREM 2.1 (CLASSICAL MEMORYLESS DETERMINACY [EMERSON AND JUTLA 1991]). *For all game graphs $G = ((V, E), (V_1, V_2))$, all Büchi objectives Φ for player 1, and the complementary coBüchi objective $\Psi = \Omega \setminus \Phi$ for player 2, we have $W_1(\Phi) = V \setminus W_2(\Psi)$. There exists a memoryless winning strategy σ for player 1 for all vertices in $W_1(\Phi)$ for the objective Φ; and there exists a memoryless winning strategy π for player 2 for all vertices in $W_2(\Psi)$ for the objective Ψ.*

Thus the theorem shows that every vertex of V either belongs to the winning set of Büchi objectives of player 1 or to the winning set of coBüchi objectives for player 2. Since we only consider this setting we simply say in the rest of the paper that every vertex either is *winning for player 1* or *winning for player 2*. Observe that for Büchi objective Φ and the coBüchi objective $\Psi = \Omega \setminus \Phi$ by definition we have $V \setminus W_2(\Psi) = \{v \in V \mid \forall \pi \in \Pi.\ \exists \sigma \in \Sigma.\ \omega(v, \sigma, \pi) \in \Phi\}$. Theorem 2.1 states that $V \setminus W_2(\Psi) = \{v \in V \mid \exists \sigma \in \Sigma.\ \forall \pi \in \Pi.\ \omega(v, \sigma, \pi) \in \Phi\}$, i.e., the order of the universal and the existential quantifiers can be exchanged. In other words, if for every strategy π of player 2 there exists a strategy σ for player 1 that wins from vertex v, then there exists a (general) strategy σ for player 1 that wins against *every* strategy π of player 2. For all objectives considered in the paper if there exists a winning strategy *with memory* for a player at a vertex v, then there exists a memoryless winning strategy for the player at v. Thus for simplicity we will only consider the simpler class of memoryless strategies.

The algorithmic question. The algorithmic question in alternating graph games with Büchi objective Φ is to compute the set $W_1(\Phi)$. In the sequel of this section we consider algorithms for Büchi games, and when we mention winning vertices or strategies we mean winning for Büchi objectives, unless explicitly mentioned otherwise.

2.2. Classical algorithm

In this section we present the classical iterative algorithm for Büchi games to compute the winning sets. We then present our new algorithm. We start with the notion of *closed sets*, *attractors*, and *alternating reachability* which are key notions for the analysis of all the algorithms we present. We present the graph theoretic definitions, and then present well-known facts that establish the connection of the graph definitions and strategies in alternating game graphs.

Closed sets. A set $U \subseteq V$ of vertices is a *closed set* for player 1 if the following two conditions hold: (a) For all vertices $u \in (U \cap V_1)$, we have $\mathsf{Out}(u) \subseteq U$, i.e., all successors of player 1 vertices in U are again in U; and (b) for all $u \in (U \cap V_2)$, we have $\mathsf{Out}(u) \cap U \neq \emptyset$, i.e., every player 2 vertex in U has a successor in U. The closed sets for player 2 are defined analogously as above by exchanging the roles of player 1 and player 2 (exchanging V_1 and V_2). Every closed set U for player $\ell \in \{1, 2\}$, induces a sub-game graph, denoted $G \restriction U$. The following proposition establishes connection of closed sets and winning for safety, reachability, and coBüchi objectives. The proof of the proposition is stratight-forward and we present it for sake of completeness.

PROPOSITION 2.2. *Consider a game graph G, and a closed set U for player 1. Then the following assertions hold:*

(1) *Player 2 has a winning strategy for the objective $\mathrm{Safe}(U)$ for all vertices in U, i.e., player 2 can ensure that if the play starts in U, then the play never leaves the set U.*

(*2*) *For all $T \subseteq V \setminus U$, we have $W_1(\mathrm{Reach}(T)) \cap U = \emptyset$, i.e., for any set T of vertices outside U, player 1 does not have a strategy from vertices in U to ensure to reach T.*
(*3*) *If $U \cap B = \emptyset$ (i.e., there is no Büchi vertex in U), then every vertex in U is winning for player 2 for the coBüchi objective.*

PROOF. We first present the proof of the first item, then show the first item implies the second item (we will also remark that the second item also implies the first item, i.e., they are equivalent). We will then argue that the third item is an easy consequence.

(1) We present a witness memoryless strategy π for player 2 to ensure the objective $\mathrm{Safe}(U)$ for all vertices in U. For a vertex $u \in U \cap V_2$, the strategy $\pi(u) = v \in U$ chooses a successor v in U (such a successor exists since by definition of closed set for all $u \in U \cap V_2$ we have $\mathsf{O}ut(u) \cap U \neq \emptyset$). Consider an arbitrary strategy σ for player 1 and a vertex $v \in U$, and the play $\omega(v, \sigma, \pi) = \langle v_0, v_1, v_2, \ldots \rangle$ with $v_0 = v$. We have $v_0 \in U$. For $i \geq 0$, (i) if $v_i \in V_1$ is a player-1 vertex and $v_i \in U$, then since U is closed (i.e., all successors of v_i also lie in U) we have that v_{i+1} also belong to U; and (ii) if $v_i \in V_2$ is a player-2 vertex and $v_i \in U$, then by definition of π, we have $v_{i+1} \in U$. It follows that the play only visits vertices in U and thus satisfy the objective $\mathrm{Safe}(U)$. Thus the strategy π is a winning strategy for player 2 for the objective $\mathrm{Safe}(U)$ for all vertices in U.
(2) Since player 2 can ensure that from all vertices in U the objective $\mathrm{Safe}(U)$ is satisfied (i.e., vertices outside U is never visited) (by the first item), it follows that for all $T \subseteq V \setminus U$ we have $W_1(\mathrm{Reach}(T)) \cap U = \emptyset$. This shows that the first item implies the second item. We also remark that if we consider the second item with $T = V \setminus U$, it implies that for all vertices in U player 2 must ensure $\mathrm{Safe}(U)$. In other words, the first and second item are equivalent.
(3) The third item is an easy consequence of the first item as the safety objective implies the coBüchi objective (i.e., $\mathrm{Safe}(U) \subseteq \mathrm{coBuchi}(U)$). In other words, if B is never visited, then clearly the Büchi objective to visit B infinitely often is violated.

The desired result follows. ■

Attractors. Given a game graph G, a set $U \subseteq V$ of target vertices, and a player $\ell \in \{1, 2\}$, the set $Attr_\ell(U, G)$ (called *attractor*) is the set of vertices from which player ℓ has a strategy to reach a vertex in U against all strategies of the other player; that is, $Attr_\ell(U, G) = W_\ell(\mathrm{Reach}(U))$. The set $Attr_1(U, G)$ can be defined inductively as follows: let $R_0 = U$; and for all $i \geq 0$ let

$$R_{i+1} = R_i \cup \{v \in V_1 \mid \mathsf{O}ut(v) \cap R_i \neq \emptyset\} \cup \{v \in V_2 \mid \mathsf{O}ut(v) \subseteq R_i\}.$$

Then $Attr_1(U, G) = \bigcup_{i \geq 0} R_i$. The fact that $Attr_1(U, G) = W_1(\mathrm{Reach}(U))$ is standard, for example see [Zielonka 1998; Thomas 1997] for details. The inductive definition of $Attr_2(U, G)$ is analogous with V_1 replaced by V_2 and vice-versa. For all vertices $v \in Attr_1(U, G)$, define $rank(v, U) = i$ if $v \in R_i \setminus R_{i-1}$, that is, $rank(v, U)$ denotes the least $i \geq 0$ such that v is included in R_i. Define a *memoryless attractor strategy* $\sigma \in \Sigma$ for player 1 as follows: for each vertex $v \in (Attr_1(U, G) \cap V_1)$ with $rank(v, U) = i$, choose a successor $\sigma(v) \in (R_{i-1} \cap \mathsf{O}ut(v))$ (such a successor exists by the inductive definition). It follows that for all vertex $v \in Attr_1(U, G)$ and all strategies $\pi \in \Pi$ for player 2, the play $\omega(v, \sigma, \pi)$ reaches U in at most $|Attr_1(U, G)|$ steps. The definition of memoryless attractor strategy for player 2 for $Attr_2(U, G)$ is similar. Observe that for $\ell \in \{1, 2\}$, we have $U \subseteq Attr_\ell(U, G)$, i.e., the set U always belongs to the attractor.

Alternating reachability. For $\ell \in \{1, 2\}$, for a vertex $u \in Attr_\ell(U, G)$ we say that u can *alt$_\ell$-reach* the set U. In other words, alt$_\ell$-reach denotes that player ℓ has a strategy to reach the target set, irrespective of the strategy of the other player.

Fact. For all game graphs G, all players $\ell \in \{1, 2\}$, and all sets $U \subseteq V$ of vertices, the following holds:

Algorithm 1 Classical algorithm for Büchi Games

Input : A game graph $G = ((V, E), (V_1, V_2))$ and $B \subseteq V$.
Output: $W \subseteq V$.
1. $G^0 := G$; $V^0 := V$; 2. $W_0 := \emptyset$; 3. $j := 0$
4. **repeat**
4.1 $W_{j+1} :=$ AvoidSetClassical$(G^j, B \cap V^j)$
4.2 $V^{j+1} := V^j \setminus W_{j+1}$; $G^{j+1} = G \restriction V^{j+1}$; $j := j + 1$;
until $W_j = \emptyset$
5. $W := \bigcup_{k=1}^{j} W_k$;
6. **return** W.

Procedure AvoidSetClassical
Input: Game graph G^j and $B^j \subseteq V^j$.
Output: set $W_{j+1} \subseteq V^j$.
1. $R^j := Attr_1(B^j, G^j)$; 2. $Tr^j := V^j \setminus R^j$; 3. $W_{j+1} := Attr_2(Tr^j, G^j)$

(1) The set $V \setminus Attr_\ell(U, G)$ is a closed set for player ℓ, i.e., no player ℓ vertex in $V \setminus Attr_\ell(U, G)$ has an edge to $Attr_\ell(U, G)$ and every vertex of the other player in $V \setminus Attr_\ell(U, G)$ has an edge in $V \setminus Attr_\ell(U, G)$.
(2) The set $Attr_\ell(U, G)$ can be computed in time $O(|\sum_{v \in Attr_\ell(U,G)} In(v)|)$ [Beeri 1980; Immerman 1981].

COROLLARY 2.3. *Every vertex in the set $V \setminus Attr_1(B, G)$ is winning for player 2 and is not winning for player 1.*

We now start with an informal description of the classical algorithm.

Informal description of classical algorithm. The *classical algorithm* (Algorithm 1) repeatedly removes vertices from the graph. We describe an iteration j of the algorithm: the set of vertices at iteration j is denoted by V^j, the game graph by G^j and the set of Büchi vertices $B \cap V^j$ by B^j. At iteration j, the algorithm first finds the set of vertices R^j from which player 1 can alt$_1$-reach the set B^j, i.e., computes $Attr_1(B^j, G^j)$. The rest of the vertices $Tr^j = V^j \setminus R^j$ is a closed subset for player 1, and $Tr^j \cap B^j = \emptyset$. Thus the set Tr^j is winning for player 2 (by Corollary 2.3). Then the set of vertices W_{j+1}, from which player 2 can alt$_2$-reach the set Tr^j, i.e., $Attr_2(Tr^j, G^j)$ is computed. The set W_{j+1} is winning for player 2, and *not* for player 1 in G^j and also in G. Thus, it is removed from the vertex set to obtain game graph G^{j+1}. The algorithm then iterates on the reduced game graph, i.e., proceeds to iteration $j + 1$ on G^{j+1}. In every iteration a linear-time attractor computation is performed with the current Büchi vertices as target to find the set of vertices which can alt$_1$-reach the Büchi set. Each iteration takes $O(m)$ time and the algorithm runs for at most $O(n)$ iterations, giving a total time of $O(n \cdot m)$. The algorithm is formally described as Algorithm 1. The correctness proof of the algorithm shows that when the algorithm terminates, all the remaining vertices are winning for player 1 [McNaughton 1993; Thomas 1997].

THEOREM 2.4 (CORRECTNESS AND RUNNING TIME). *Given a game graph $G = ((V, E), (V_1, V_2))$ and $B \subseteq V$ the following assertions hold:*

(*1*) *$W = W_2(\mathrm{coBuchi}(V \setminus B))$ and $V \setminus W = W_1(\mathrm{Buchi}(B))$, where W is the output of Algorithm 1; and*
(*2*) *the running time of Algorithm 1 is $O(n \cdot m)$.*

We also remark that the analysis of the classical algorithm is optimal, i.e., there exists a family of game graphs where the classical algorithm require $\theta(n \cdot m)$ time (for example see [Chatterjee et al. 2006]).

2.3. New algorithm

In this section we present our new algorithm for computing the winning set for game graphs with Büchi objectives in time $O(n^2)$.

Notations. Given an alternating game graph $G = ((V, E), (V_1, V_2))$ and a set B of Büchi vertices, we label the Büchi vertices as priority 0 vertices, and the set $V \setminus B$ as priority 1 vertices. For every vertex v the inedges have a ***fixed*** order such that all edges from priority 1 player-2 vertices come before all other edges. In other words, we assign priority 1 to edges (u, v) such that $u \in (V \setminus B) \cap V_2$ (player-2 vertices that are not Büchi vertices), and assign priority 0 to all other edges, and priority 1 edges come before priority 0 edges in the fixed order of the edges. Let $\widetilde{G} = (\widetilde{V}, \widetilde{E})$ be a sub-graph of G with $\widetilde{V} \subseteq V$, and $\widetilde{E} \subseteq E \cap (\widetilde{V} \times \widetilde{V})$, such that each vertex has at least one outgoing edge. We define $\log n$ sub-graphs $\widetilde{G}_i$ of $\widetilde{G}$ such that $\widetilde{G}_i = (\widetilde{V}, \widetilde{E}_i)$. The set $\widetilde{E}_i$ contains all edges (u, v) where

(1) $|\mathsf{O}ut(u) \cap \widetilde{E}| \leq 2^i$ (i.e., the outdegree of u in $\widetilde{E}$ is at most 2^i), or
(2) the edge (u, v) belongs to the first 2^i inedges of vertex v in $\widetilde{E}$.

Note that $\widetilde{E}_{i-1} \subseteq \widetilde{E}_i$ since the order of the inedges is fixed. We color every player-1 vertex v in $\widetilde{G}_i$ ***blue*** if $\mathsf{o}utdeg(v) > 2^i$ in $\widetilde{E}$. We color every player-2 vertex v in G_i ***red*** if $\mathsf{o}utdeg(v) > 2^i$ in $\widetilde{E}$. All other vertices have color white. For every vertex v that is white in $\widetilde{G}_i$, all its outedges $\mathsf{O}ut(v)$ are contained in $\widetilde{E}_i$. There are up to $2^i \cdot n$ such edges to $\widetilde{E}_i$. Additionally the first up to 2^i inedges of every vertex belong to $\widetilde{E}_i$, and again there are up to $2^i \cdot n$ such edges to $\widetilde{E}_i$. Thus $|\widetilde{E}_i| \leq 2^{i+1} \cdot n$. Note that $\widetilde{G} = \widetilde{G}_{\log n}$ and thus all vertices in $\widetilde{G}_{\log n}$ are white.

The new algorithm NEWBUCHIALGO**.** The new algorithm consists of two nested loops, an outer loop with loop counter j and an inner loop with loop counter i. The algorithm will iteratively delete vertices from the graph, and we denote by D_j the set of vertices deleted in iteration j, and by U the set of vertices deleted in all iterations upto the current iteration (initially U is empty). For $j \geq 1$, we will denote by G^j the game graph after removal of the set U of vertices at the beginning of iteration j. We denote the vertex set in iteration j as V^j, the edge set as E^j, and the Büchi set as B^j (i.e., $B^j := V^j \cap B$). We denote by $G^j_i = (V^j, E^j_i)$ the sub-graph of $G^j = (V^j, E^j)$ as defined above (i.e., we treat G^j as $\widetilde{G}$ and obtain G^j_i and $\widetilde{G}_i$). For clarification we have the following properties for G^j_i: E^j_i contains all edges (u, v) where (i) $|\mathsf{O}ut(u) \cap E^j| \leq 2^i$ or (ii) the edge (u, v) belongs to the first 2^i inedges of vertex v in E^j; (iii) $E^j_{i-1} \subseteq E^j_i$; (iv) every player-1 vertex v in G^j_i is ***blue*** if $\mathsf{o}utdeg(v) > 2^i$ in E^j; (v) every player-2 vertex v in G^j_i is ***red*** if $\mathsf{o}utdeg(v) > 2^i$ in E^j; and (vi) all other vertices have color white. Note that G^0_i is G_i (the initial graphs). Also note that since vertices are removed over iterations, the graphs G^j_i can include edges that were not included in G^0_i. The intuitive description of the new algorithm is as follows: Starting from $i = 0$ the algorithm searches in each iteration j in each graph G^j_i for a special player-1 closed set S_j with no Büchi vertex and stops at the smallest i at which such a closed set exists. Since $S_j \cap B^j = \emptyset$, Proposition 2.2 implies that all the vertices in S_j are winning for player 2. Thus, by the same arguments as for the classical algorithm the player-2 attractor $Attr_2(S^j, G^j_i)$ are winning for player 2 in G^j_i and, as our correctness proof shows, also winning in G. Thus they are removed from the vertex set and the algorithm iterates on the reduced game graph. Computing S_j takes time $O(2^i \cdot n)$ and, due to the fact that no such set was found in G^j_{i-1} we can show that S_j contains at least 2^{i-1} vertices. Thus, using amortized analysis we charge $O(n)$ to each of the 2^{i-1} vertices in S_j that are removed, giving a total running time of $O(n^2)$. The details of NEWBUCHIALGO follow.

(1) Let $j := 0$; $Y_0 := Attr_1(B, G)$; $X_0 := V \setminus Y_0$; $D_0 := Attr_2(X, G)$.
(2) Remove the vertices of D_j to obtain graph G^j; $j := j + 1$; and $U := U \cup D_j$;
(3) $i := 1$;
(4) repeat
   (a) Construct graph G_i^j. Let Z_i^j be the vertices of V^j that are (i) either red *with no outedges* in G_i^j or (ii) blue in G_i^j.
   (b) $Y_i^j := Attr_1(B^j \cup Z_i^j, G_i^j)$;
   (c) $S_j := V^j \setminus Y_i^j$;
   (d) $i := i + 1$;
(5) until S_j is non-empty or $i = \log n$
(6) if $S_j \neq \emptyset$, then $D_j := Attr_2(S_j, G^j)$ and go to Step 2, else the whole algorithm terminates and outputs $V \setminus U$.

Note that in Step 2 the vertex set D_j is removed to obtain the graph G^j, but we do not immediately construct all sub-graphs G_i^j. Instead we construct G_i^j in the inner loop, i.e., the graph G_i^j is constructed only if $V^j \setminus Y_{i-1}^j$ is empty in iteration $i - 1$ of the inner loop.

Correctness analysis. Let U^* be the set of vertices removed from the graph over all iterations and $Y^* = V \setminus U^*$ be the output of the algorithm. We first show that $Y^* \subseteq W_1(\Phi)$, where Φ is the Büchi objective, i.e., Y^* is winning for player 1. Then we show that $U^* \cap W_1(\Phi) = \emptyset$ (i.e., U^* is not winning for player 1). Together with Theorem 2.1 this shows that $Y^* = W_1(\Phi)$ estabilishing the correctness of the algorithm. Finally we analyze the running time of the algorithm. We first present Lemma 2.5 and Lemma 2.7 and the proofs of these two lemmata are similar (but not identical as our algorithm and the classical algorithm removes different sets in respective iterations) to the correctness proof for the classical algorithm.

LEMMA 2.5. *Let Y^* be the output of* NEWBUCHIALGO*, and let G^* and B^* be the game graph and the Büchi set on termination, respectively (i.e., G^* is the graph induced by Y^* and B^* is $B \cap Y^*$). The following assertions hold:*

(1) $Y^* = Attr_1(B^*, G^*)$, i.e., player 1 can alt$_1$-reach the set B^* in G^* from Y^*.
(2) Y^* is a player-2 closed set in the original game graph G.
(3) $Y^* \subseteq W_1(\Phi)$, where Φ is the Büchi objective.

PROOF. We prove the three part below.

(1) Consider the last iteration j^* of the outer loop of the algorithm. Since it is the last iteration, the set S_{j^*} must be empty. It follows that i must have been $\log n$ in the last iteration of the repeat loop, i.e., the last iteration of the repeat loop considered $G_{\log n}^{j^*} = G^*$. Let $i = \log n$. Note that all vertices are white in G^*, i.e., $Z_i^{j^*}$ was empty. Hence we have $Y_i^{j^*} = Attr_1(B^* \cup Z_i^{j^*}, G^*) = Attr_1(B^*, G^*)$. Note that $Y^* = Y_i^{j^*}$. Hence the fact that S_{j^*} was empty at the end of the iteration implies that $V^{j^*} \setminus Y_i^{j^*}$ was empty, i.e., all vertices of G^* belong to $Attr_1(B^*, G^*)$. Hence $Y^* = Attr_1(B^*, G^*)$.
(2) Whenever a set of vertices is deleted in any iteration, it is an player-2 attractor. Hence if a vertex $u \in Y^* \cap V_2$ would have an edge to a vertex $v \in U^*$, then u would have been included in U^* (where $U^* = V \setminus Y^*$). Similarly for a player 1 vertex $u \in Y^* \cap V_1$ it must have an edge in Y^*, as we assume that it has at least one outedge and if all its outedges pointed to U^* it would have been included in U^*. It follows that Y^* is a player-2 closed set in G.
(3) The result is obtained from the previous two items. Consider a memoryless attractor strategy σ in G^* for player-1 that ensures that for all vertices in Y^* the set B^* is reached within $|Y^*|$ steps against all strategies of player-2. Moreover the strategy only chooses successor in Y^*. Since Y^* is a player-2 closed set, it follows that against all strategies of player-2 the set Y^* is never left,

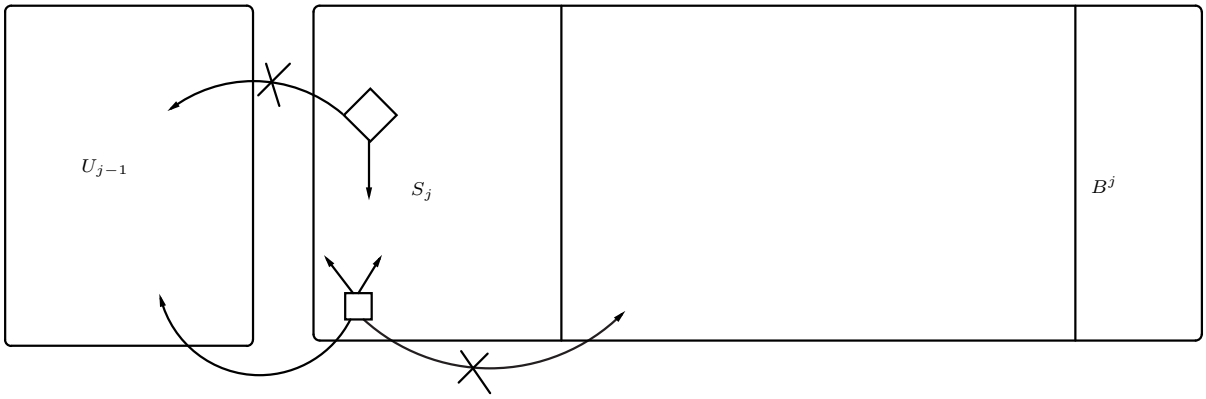

Fig. 1. Pictorial depiction of U_{j-1} and S_j.

thus it is ensured that B^* is visited infinitely often. Hence the strategy σ ensures that for all vertices $v \in Y^*$ and all strategies π we have $\omega(v, \sigma, \pi) \in \Phi$. It follows that $Y^* \subseteq W_1(\Phi)$.

The desired result follows. ■

To complete the correctness proof we need to show that if $U^* = V \setminus Y^*$, then $U^* \cap W_1(\Phi) = \emptyset$, where Φ is the Büchi objective. We will show the result by induction on the number of iterations. Let us denote by U_j the set of vertices removed till iteration j. The base case is trivial as initially $U = \emptyset$. By inductive hypothesis, we assume for $j \geq 1$ we have $U_{j-1} \cap W_1(\Phi) = \emptyset$, and then show that $U_j \cap W_1(\Phi) = \emptyset$. Let G^j be the alternating game graph obtained after removal of the set U_{j-1} of vertices. We will show the following proposition.

PROPOSITION 2.6. *In G^j, let S_j be the non-empty set identified in iteration j, then $Attr_1(B^j, G^j) \cap S_j = \emptyset$.*

In the following lemma we first show how with Proposition 2.6 we establish the correctness of our algorithm and finally prove Proposition 2.6 to complete the correctness proof.

We first depict the situation with the aid of a figure that would help in understanding the following lemma. The situation is shown in Figure 1 where $\Box$ denote player-1 vertices and $\Diamond$ denote player-2 vertices. In the figure U_{j-1} denotes the vertices already removed, and since player-2 attractors are removed it follows that player-2 edges to U_{j-1} are not possible from the remaining vertices. Since S_j is a closed set for player 1, for all player-1 vertices in S_j either the edges point to U_{j-1} or to S_j, but not to vertices outside $U_{j-1} \cup S_j$, and all player-2 vertices in S_j have at least one edge in S_j. Also note that the vertex set B^j of the remaining Büchi vertices (after removal of U_{j-1}) does not intersect with S_j. With this pictorial view we now prove the following lemma.

LEMMA 2.7. *The inductive hypothesis that $U_{j-1} \cap W_1(\Phi) = \emptyset$ and Proposition 2.6 implies that $U_j \cap W_1(\Phi) = \emptyset$.*

PROOF. We first show a claim, and then use it to establish the lemma.

Claim. The inductive hypothesis that $U_{j-1} \cap W_1(\Phi) = \emptyset$ and Proposition 2.6 implies that $S_j \cap W_1(\Phi) = \emptyset$.

We first prove the claim. By Proposition 2.6 we have $Attr_1(B^j, G^j) \cap S_j = \emptyset$, and it follows that if player 1 follows a strategy from any vertex in S_j such that the set $V^j = V \setminus U_{j-1}$ of vertices is never left, then no Büchi vertex is ever reached, as it is not possible to reach B^j from a vertex of S^j in the subgraph G^j. If the set V^j is left after a finite number of steps, then the set U_{j-1} is reached,

and by inductive hypothesis $U_{j-1} \cap W_1(\Phi) = \emptyset$, i.e., player 2 can ensure from U_{j-1} that the set of Büchi vertices is visited finitely often. Since the Büchi objective is independent of finite prefixes, it follows that if V^j is left and U_{j-1} is reached, then player 2 ensures that the Büchi objective is not satisfied. It follows that $S_j \cap W_1(\Phi) = \emptyset$. Thus we have the desired claim.

We now prove the lemma using the above claim. Observe that $U_j \setminus U_{j-1}$ is a player 2 attractor to S_j, and hence player 2 can ensure from $U_j \setminus U_{j-1}$ that S_j is reached in finite number of steps. Since Büchi objectives are independent of finite prefixes, by inductive hypothesis $U_{j-1} \cap W_1(\Phi) = \emptyset$, and by the above claim we have $S_j \cap W_1(\Phi) = \emptyset$, it follows that $U_j \cap W_1(\Phi) = \emptyset$. ■

Hence to complete the proof we need to establish Proposition 2.6. Suppose a non-empty subset S_j is identified at iteration j and let S_j be identified at iteration i of the inner loop. Observe that we have $S_j = V^j \setminus Attr_1(B^j \cup Z_i^j, G_i^j)$ and thus $S_j \cap Attr_1(B^j \cup Z_i^j, G_i^j) = \emptyset$. This implies that $S_j \cap Attr_1(B^j, G_i^j) = \emptyset$. However to establish Proposition 2.6 we need to show that $S_j \cap Attr_1(B^j, G^j) = \emptyset$ (which does not follow from $S_j \cap Attr_1(B^j, G_i^j) = \emptyset$ as G^j may have more edges than G_i^j). While the proofs of Lemma 2.5 and Lemma 2.7 are similar to the correctness proof of the classical algorithm, establishing Proposition 2.6 and the running time analysis is the heart of our proof. The proofs require the notion of a separating cut: separating cuts and the following lemmata of the section are the crux of the proof.

Separating cut. We say a set S of vertices induces a *separating cut* in a graph G_i or G_i^j if (a) the only edges from S to $V \setminus S$ come from player-2 vertices in S, (b) every player-2 vertex in S has an edge to another vertex in S, (c) every player-1 vertex in S is white, and (d) $B \cap S = \emptyset$. Thus S is a player-1 closed set where every player-1 vertex is white and which does not contain a vertex in B.

We will now present some lemmata that will establish Proposition 2.6 and will also be used in the running time analysis.

LEMMA 2.8. *Let $G = ((V, E), (V_1, V_2))$ be a game graph where every vertex has at least outdegree 1, and $G' = ((V, E'), (V_1, V_2))$ be a sub-graph of G with $E' \subseteq E$. Let V_W be the set of vertices that are colored white such that for all vertices v in V_W we have $\mathsf{Out}(v) \cap E = \mathsf{Out}(v) \cap E'$. Let $Z \subseteq V \setminus V_W$ be a set of blue player-1 and red player-2 vertices of G' such that all red vertices in Z have outdegree 0 in G'. If S induces a separating cut in G', then $S \cap Attr_1(B \cup Z, G) = \emptyset$.*

PROOF. We first show that every vertex in S has an edge to another vertex in S in G'. For player-2 vertices this follows from condition (b) of a separating cut. For player-1 vertices this follows since they have outdegree 1 in G, are white in G', and cannot have an edge to a vertex in $V \setminus S$.

Note that $S \cap (B \cup Z) = \emptyset$ since S contains no blue vertex, every red vertex in S has outdegree at least 1 and $B \cap S = \emptyset$ by condition (d) of a separating cut. By condition (a) of a separating cut for all player-1 vertices in S all outgoing edges of G' are in S. It follows that S is a player-1 closed set in G'. By condition (c) of a separating cut all player-1 vertices in S must be white (i.e., $S \subseteq V_W$), and for white vertices in V_W the set of outedges in G' and G coincide. It follows that S is a player-1 closed set in G. Since S is a player-1 closed set in G and $S \cap (B \cup Z) = \emptyset$, the result follows from Proposition 2.2 (second item). ■

LEMMA 2.9. *Let S_j be the non-emptyset computed by* NEWBUCHIALGO *in iteration j. Then (1) S_j is a separating cut in G^j; and (2) $S_j \cap Attr_1(B^j, G^j) = \emptyset$.*

PROOF. We establish both the items of the result.

(1) We establish all the conditions of a separating cut for S_j, where S_j is obtained in iteration i^* of the inner loop for iteration j.
   (a) *Condition (a).* By construction no player-1 vertex in S_j has an edge to $V^j \setminus S_j$, otherwise it would belong to the player-1 attractor of $B^j \cup Z_{i^*}^j$. Since all player-1 vertices in S_j are white in $G_{i^*}^j$, the outedges of the player-1 vertices in S_j are the same in $E_{i^*}^j$ and in E^j. Thus condition (a) of a separating cut holds in G^j.

(b) *Condition (b).* Every player-2 vertex v in S_j must have an edge in $E^j_{i^*}$ to a vertex in S_j, otherwise all its edges in $E^j_{i^*}$ would go to vertices in $V^j \setminus S_j$ and thus it would belong to $Attr_1(B^j \cup Z^j_{i^*}, G^j_{i^*})$. Since $E^j_{i^*} \subseteq E^j$, it follows that every player-2 vertex v in S_j must have an edge in E^j to a vertex in S_j. Hence condition (b) of a separating cut holds in G^j.

(c) *Condition (c).* All vertices are white in G^j. Thus condition (c) holds trivially.

(d) *Condition (d).* The condition (d), $S_j \cap B^j = \emptyset$ holds, since otherwise a vertex of S_j would belong to B^j and, thus, to $Attr_1(B^j \cup Z^j_{i^*}, G^j_{i^*})$.

Thus S_j induces a separating cut in G^j. The desired result follows.

(2) Let v be a vertex in S_j. By construction v cannot alt_1-reach $B^j \cup Z^j_{i^*}$ in $G^j_{i^*}$, where i^* was the last value of i in the repeat loop of iteration j. We will show that v cannot alt_1-reach B^j in G^j. As we showed in the first item of the lemma, S_j induces a separating cut in G^j; and thus we can apply Lemma 2.8 with $G = G^j$, $G' = G^j_{i^*}$, $Z = \emptyset$, and $S = S_j$ to obtain the result of the second item.

The desired result follows. ■

Lemma 2.9 proves Proposition 2.6 and this completes the correctness proof, and gives the following lemma.

LEMMA 2.10. *Let Y^* be the output of* NEWBUCHIALGO. *Then we have $Y^* = W_1(\Phi)$, where Φ is the Büchi objective.*

Running time analysis. We now analyze the running time of the algorithm.

LEMMA 2.11. *Let G^j_i be the game graph in iteration j of the outer loop and iteration i of the inner loop. If S induces a separating cut in G^j_i, then $S \subseteq S_j$.*

PROOF. Let Z^j_i be the set of blue and degree-0 red vertices of G^j_i as defined in iteration j of the outer loop and i of the inner loop of the algorithm. We invoke Lemma 2.8 with $G = G' = G^j_i$, $B = B^j$ and $Z = Z^j_i$, and obtain that none of the vertices in S can alt_1-reach $B^j \cup Z^j_i$ in G^j_i. Hence we have $S \subseteq V^j \setminus Attr_1(B^j \cup Z^j_i, G^j_i)$. Thus $S \subseteq S_j$. ■

LEMMA 2.12. *Consider an iteration j of the outer loop of* NEWBUCHIALGO *such that the algorithm stops the inner loop at value i and identifies a non-empty set S_j. Then S_j is a separating cut in G^j_i.*

PROOF. Consider the non-empty set S_j obtained in the graph G^j_i. First, note that all player-1 vertices in S_j are white, since Z^j_i contains all blue player-1 vertices of V^j and $S_j = V^j \setminus Attr_1(B^j \cup Z^j_i, G^j_i)$. Also note that all red player-2 vertices without an outedge also belong to Z^j_i. Since S_j is a complement of a player-1 attractor in G^j_i it follows that S_j is a player-1 closed set. Moreover as the target set $B^j \cup Z^j_i$ contain all Büchi vertices, there is no Büchi vertex in S_j. Thus S_j is a player-1 closed set where every player-1 vertex is white, and S_j does not contain a Büchi vertex. Thus, S_j induces a separating cut in G^j_i. ■

Note that G^j has possibly more edges than G^j_i and separating cuts are not preserved if we consider more edges (condition (a) maybe violated) or less edges (condition (b) maybe violated). Thus Lemma 2.12 is neither a consequence of nor implies Lemma 2.9 (item 1). By Lemma 2.12 we have that S_j is a separating cut in G^j_i. Lemma 2.11 shows that every separating cut S in G^j_i is a subset of S_j. It follows that S_j is the largest (under set inclusion) separating cut. Recall that for the running time analysis we need to show that computing S_j takes time $O(2^i \cdot n)$ and, due to the fact that no such set was found in G^j_{i-1} we can show that S_j it contains at least 2^{i-1} vertices. Using

Lemma 2.11 we show that NEWBUCHIALGO identifies a separating cut in G_i^j for the smallest i, and also identifies the largest separating cut in G_i^j. Thus Lemma 2.11 is key to the running time analysis, which we present in Lemma 2.13. Before the details of Lemma 2.13 we describe the data structure to ensure that in every iteration j, the construction of the graph G_i^j can be achieved in time $O(n \cdot 2^i)$.

Graph construction. We maintain with each vertex (i) the list of its outedges; (ii) the list of its inedges sorted according to the fixed order of inedges (i.e., edges from priority 1 player-2 vertices or priority 1 edges come before all other edges); (iii) list of pointers to the element of the list of outedges and inedges of other vertices it belongs to. When a vertex is removed in an iteration, then using the list of pointers we update the list of inedges and outedges of the other vertices. Hence over all iterations the data structures are maintained with $O(n^2)$ work. Given the list of outedges and inedges in sorted order, the graph G_i^j is constructed in time $O(n \cdot 2^i)$ as for every vertex we traverse the list of inedges and outedges upto the first 2^i elements.

LEMMA 2.13. *The total time spent by* NEWBUCHIALGO *is* $O(n^2)$.

PROOF. We present the $O(n^2)$ running time analysis and we consider two cases. Recall that vertices in $V \setminus B$ are refered as priority-1 vertices.

All other than the last iteration of the outer loop. Assume in iteration j the algorithm stops the repeat until loop at value i and this is not the last iteration of the algorithm. Then S_j is not empty. By Lemma 2.12 we have that S_j induces a separating cut in G_i^j. Consider the set S_j in G_{i-1}^j. There are 2 cases to consider:

(1) *Case 1:* S_j contains a player-1 vertex x that is blue in G_{i-1}^j. Thus x has outdegree at least 2^{i-1} in G_i^j and none of these edges go to vertices in $V^j \setminus S_j$ in G_i^j. Thus, S_j contains at least 2^{i-1} vertices.

(2) *Case 2:* All player-1 vertices in S_j are white in G_{i-1}^j. Thus, their outedges in G_i^j and G_{i-1}^j are identical.
Consider a player-2 vertex u in S_j. Thus there exists an edge $(u, v) \in E_i^j$ with $v \in S_j$. There are two possibilities.
   — *Case 2a:* For all player-2 vertices $u \in S_j$ there exists a vertex $v \in S_j$ with $(u, v) \in E_{i-1}^j$. But then S_j would be a separating cut in G_{i-1}^j. By Lemma 2.11 it follows that a non-empty subset S with $S_j \subseteq S$ would be non-empty in iteration $i-1$ and thus the repeat loop would have stopped after iteration $i-1$. This is not the case and thus the condition of Case 2a does not hold.
   — *Case 2b:* There exists a player-2 vertex $u \in S_j$ that has an edge $(u, v) \in E_i^j$ to a vertex $v \in S_j$ but this edge is not contained in E_{i-1}^j. This can only happen if v has at least 2^{i-1} other inedges in E_{i-1}^j. Note that u is a priority-1 player-2 vertex, and hence the edge (u, v) has priority 1 and recall that by the fixed inorder of edges, priority-1 edges come before all priority 0 edges. Hence it follows that since the edge (u, v) is not in G_{i-1}^j, all inedges of v that are in G_{i-1}^j must have priority 1 by the fixed order of inedges, i.e., all the inedges of v in G_{i-1}^j are from priority-1 player-2 vertices. We now argue that no priority-1 player-2 vertex in $Y_i^j = V^j \setminus S_j$ has an edge to a vertex S_j in G_i^j: (i) all priority 1 player-2 vertices in Z_i^j are red (and hence have no outgoing edge in G_i^j); and (ii) since $Y_i^j = Attr_1(B^j \cup Z_i^j, G_i^j)$ we have that for all player-2 vertices in $(Y_i^j \setminus (B^j \cup Z_i^j))$ all its outgoing edges in G_i^j must be contained in Y_i^j. Thus no priority-1 player-2 vertex in $V^j \setminus S_j$ has an edge to a vertex of S_j in G_i^j. Since $E_{i-1}^j \subseteq E_i^j$, no priority-1 player-2 vertex in $V^j \setminus S_j$ has an edge to a vertex in S_j in G_{i-1}^j. It follows that none of the inedges of v in G_{i-1}^j are from $V^j \setminus S_j$ and, since

v has at least 2^{i-1} inedges from priority-1 player-2 vertices, the set S_j must contain at least 2^{i-1} player-2 vertices.

Thus in either case S_j contains at least 2^{i-1} vertices and all these vertices are deleted. The time spent for all the executions of the repeat loop in this iteration of the outer loop is the time spent in all graphs G_1, G_2, ..., G_{i^*}, which sums to $O(2^i \cdot n)$ (for the graph construction and the attractor computation). We charge $O(n)$ work to each deleted vertex. This accounts for all but the last iteration of the outer loop. As the algorithm deletes at most n vertices the total time spent over the whole algorithm other than the last iteration is $O(n^2)$.

The last iteration of the outer loop. In the last iteration j^* of the outer loop, when no vertex is deleted, the algorithm works on all $\log n$ graphs, spending time $O(n \cdot 2^i)$ in graph $G_i^{j^*}$. Since each graph $G_i^{j^*}$ has at most $n \cdot 2^{i+1}$ edges and there are $\log n$ graphs, the total number of edges worked in the last iteration is

$$\sum_{i=1}^{\log n} n \cdot 2^{i+1} = 4 \cdot n \cdot \sum_{i=1}^{\log n} 2^{i-1} = 4 \cdot n \cdot (2^{\log n} - 1) = 4 \cdot n \cdot (n-1) = O(n^2).$$

Hence the total time required in the last iteration is $O(n^2)$. An identical argument also shows that the time to build all the graphs $G_i^{j^*}$ is at most $O(n^2)$. Hence the desired result follows. ■

THEOREM 2.14. *Given an alternating game graph G with n vertices, and an Büchi objective Φ, algorithm* NEWBUCHIALGO *correctly computes the winning set $W_1(\Phi)$ in time $O(n^2)$.*

2.4. Decremental and incremental algorithms

In this section we present the decremental and incremental algorithms for computing the winning set in game graphs with Büchi objectives. We will show that the *small progress measure* algorithm of [Jurdziński 2000] works in total time $O(n \cdot m)$ for a sequence of player-1 edge deletions or insertions, and hence the amortized time per operation is $O(n)$.

Motivation. In verification and synthesis of open systems, the systems under verification are developed incrementally by adding choices (or decrementally by removing choices) for the system till the objective is satisfied. The system choices are represented by player 1, whereas the adversary, modeled by player 2, is the adversarial environment, and the system design has no control over the environment choices. Hence dynamic algorithms with player-1 edge deletions or insertions are the relevant decremental and incremental algorithms required for Büchi games, and we will only study these kind of update operations. Moreover, since Büchi objectives generalize reachability objectives, and alternating game graphs generalize directed graphs, our algorithm is a significant generalization of the Even-Shiloach algorithm [Even and Shiloach 1981] for decremental reachability in graphs. However our proof is very different, based on a fix-point argument, and is much simpler. In other words, the second motivation is to present decremental and incremental algorithms for alternating games (that subsume graphs) with simple fix-point based correctness proof.

Summary of previous results. Our decremental and incremental algorithms will be based on the small progress measure algorithm of [Jurdziński 2000], which takes $O(n \cdot m)$ time for alternating Büchi and coBüchi games.[3] In our decremental algorithm we will use the small progress measure algorithm for alternating games with *Büchi* objectives; and in our incremental algorithm we will use the small progress measure algorithm for alternating games with *coBüchi* objectives. The small progress measure algorithm for Büchi games maintains an integer (called *progress measure*) for every vertex and updates it using a monotonic *lift* operation based on the successor vertices, until a fix-point is reached. We will show how to adapt the progress measure algorithm to present decremental and incremental algorithms for alternating Büchi games.

[3] We will specialize the small progress measure algorithm of [Jurdziński 2000] (that also works for parity objectives) for Büchi and coBüchi objectives.

2.4.1. Decremental algorithm for Büchi games. In this section we present the decremental algorithm, and we consider only deletion of player-1 edges, as discussed in the motivation.

Previous results on progress measure [Jurdziński 2000]. Our decremental algorithm is based on the notion of progress measure. We start with the notion of a progress measure and valid progress measure.

Progress measure. Given a game graph with n vertices, a progress measure is a function $\rho : V \to [n] \cup \top$, where $[n] = \{0, 1, 2, \ldots, n\}$, that assigns to every vertex either a number from 0 to n, or the *top element* $\top$. We will follow the conventions that: (a) for all $j \in [n]$ we have $j < \top$; (b) $n + 1 = \top$; (c) $\top + 1 = \top$; (d) $\top \geq \top$. Intuitively, $\top$ will be assigned to a vertex if it does not belong to the winning set. Given a game graph with a set B of Büchi vertices, a progress measure ρ is a *valid* progress measure if the following conditions hold for all $v \in V$:

$$\rho(v) \begin{cases} \geq \min_{(v,w)\in E} \rho(w) + 1 & v \in V_1 \setminus B; \\ \geq \max_{(v,w)\in E} \rho(w) + 1 & v \in V_2 \setminus B; \\ \begin{cases} = \top & v \in V_1 \cap B, \text{for all } (v,w) \in E \text{ we have } \rho(w) = \top; \\ = 0 & v \in V_1 \cap B, \text{there exists } (v,w) \in E \text{ such that } \rho(w) \neq \top; \end{cases} \\ \begin{cases} = \top & v \in V_2 \cap B, \text{there exists } (v,w) \in E \text{ such that } \rho(w) = \top; \\ = 0 & v \in V_2 \cap B, \text{for all } (v,w) \in E \text{ we have } \rho(w) \neq \top; \end{cases} \end{cases}$$

We define the comparison operators $\leq, \geq$ on progress measures with the *pointwise* comparison, i.e., for $\bowtie \in \{\leq, \geq\}$ and progress measures ρ_1 and ρ_2, we write $\rho_1 \bowtie \rho_2$ iff for all $v \in V$ we have $\rho_1(v) \bowtie \rho_2(v)$.

Lift operation on progress measure. Given a game graph G, the function Lift^G takes as input a progress measure and returns a progress measure. For all input progress measures ρ, the output progress measure $\rho' = \mathsf{Lift}^G(\rho)$ is defined as follows: for all $v \in V$, (i) for $v \in V_1 \cap B$, we have $\rho'(v) = \top$ if for all $(v,w) \in E$ we have $\rho(w) = \top$, and 0 otherwise; (ii) for $v \in V_2 \cap B$, we have $\rho'(v) = \top$ if there exists $(v,w) \in E$ with $\rho(w) = \top$, and 0 otherwise; (iii) for $v \in V_1 \setminus B$, we have $\rho'(v) = \min_{(v,w)\in E} \rho(w) + 1$; and (iv) for $v \in V_2 \setminus B$, we have $\rho'(v) = \max_{(v,w)\in E} \rho(w) + 1$. A function f operating on progress measures (that takes as input a progress measure and returns a progress measure) is called *monotonic* if for all progress measures $\rho_1 \leq \rho_2$ we have $f(\rho_1) \leq f(\rho_2)$.

LEMMA 2.15. *For all game graphs G, the function Lift^G is monotonic.*

PROOF. Consider progress measures ρ_1, ρ_2 such that $\rho_1 \leq \rho_2$. For a non-Büchi vertex $v \in (V \setminus B)$ we have

$$\mathsf{Lift}^G(\rho_1)(v) = \begin{cases} \min_{(v,w)\in E} \rho_1(w) + 1 \leq \min_{(v,w)\in E} \rho_2(w) + 1 = \mathsf{Lift}^G(\rho_2)(v) & v \in V_1 \setminus B; \\ \max_{(v,w)\in E} \rho_1(w) + 1 \leq \max_{(v,w)\in E} \rho_2(w) + 1 = \mathsf{Lift}^G(\rho_2)(v) & v \in V_2 \setminus B; \end{cases}$$

where E is the set of edges in G. It follows that for all $v \in (V \setminus B)$ we have $\mathsf{Lift}^G(\rho_1)(v) \leq \mathsf{Lift}^G(\rho_2)(v)$. Note that for vertices in B, progress measures are either 0 or $\top$. For $v \in B$ we have the following cases: (i) $v \in V_1 \cap B$*:* if $\mathsf{Lift}^G(\rho_1)(v) = \top$, then for all $(v,w) \in E$ we have $\rho_1(w) = \top$, and hence for all $(v,w) \in E$ we have $\rho_2(w) = \top$; thus $\mathsf{Lift}^G(\rho_2)(v) = \top$; and (i) $v \in V_2 \cap B$*:* if $\mathsf{Lift}^G(\rho_1)(v) = \top$, then there exists $(v,w) \in E$ with $\rho_1(w) = \top$, and hence we have $\rho_2(w) = \top$; thus $\mathsf{Lift}^G(\rho_2)(v) = \top$. It follows that we have $\mathsf{Lift}^G(\rho_1) \leq \mathsf{Lift}^G(\rho_2)$. The desired result follows. ■

Since Lift^G is a monotonic function on a finite lattice, by the Tarski-Knaster Theorem [Kechris 1995] it has a least fix-point. Given a player-1 attractor $Attr_1(U, G)$, the *minimal alternating distance* of a vertex $v \in Attr_1(U, G)$ is the rank $rank(v, U)$ of the vertex v (in other words it is the

alternating shortest distance to U where player-1 minimizes the distance and player-2 maximizes the distance to U) (recall the definition of rank from Section 2.1). The result of [Jurdziński 2000] established that for all game graphs G, (i) there is a *unique* least fix-point ρ^* of Lift^G, (ii) the least fix-point ρ^* is a valid progress measure, (iii) the least fix-point ρ^* fulfills the following conditions: (a) for all vertices v in the complement of the winning set we have $\rho^*(v) = \top$; (b) for all Büchi vertices v in the winning set we have $\rho^*(v) = 0$; and (c) for all non-Büchi vertices v in the winning set we have $\rho^*(v) = rank(v, B^*)$, where B^* is the set of Büchi vertices in the winning set (i.e., in the winning set the progress measure equals the minimal alternating distance to the set of Büchi vertices in the winning set). The result of [Jurdziński 2000] holds actually for the more general case of parity objectives, and the specialization to Büchi objectives yields the above properties.

THEOREM 2.16 ([JURDZIŃSKI 2000]). *For all game graphs G, let ρ^* be the least fix-point of Lift^G, and let $||\rho^*|| = \{v \in V \mid \rho(v) \in [n]\}$ denote the set of vertices that are not assigned the top element. Then $||\rho^*|| = W_1(\Phi)$, where Φ is the Büchi objective.*

Decremental algorithm. We now present our decremental algorithm. Our algorithm initially computes the least fix-point progress measure ρ^* of the graph and then maintains it after each edge deletion by repeatedly applying the lift operator to the fix-point ρ^* stored *before* the edge deletion. To prove the correctness we will show that the fix-point obtained by repeatedly applying the lift operator on the previous least fix-point converges to the least fix-point of the new game graph. The algorithm maintains the following data structure: (i) For each vertex $x \in V_1 \cap B$ it keeps a list of vertices w such that $(x, w) \in E$ and $\rho^*(w) \neq \top$ and (ii) for each vertex $x \in V_1 \setminus B$ a list of vertices w such that $(x, w) \in E$ and $\rho^*(x) = \rho^*(w) + 1$. (iii) Every edge (x, w) has a pointer to its location in the list of x if it is stored in such a list. During each update operation, the algorithm maintains a queue data structure that contains all player-2 vertices whose progress measure has increased and all player-1 vertices that has an outedge to a vertex whose progress measure has increased. We next describe the algorithm in detail.

Computation of the initial ρ^*. Use the static Büchi algorithm from the previous section to compute the player-1 and player-2 winning sets and assign $\top$ to all vertices in the player-2 winning set. Use the backward search algorithm [Beeri 1980; Immerman 1981] to determine the rank of every vertex in the player-1 winning set and set its initial progress measure equal to its rank. Then compute for each vertex of V_1 its list.

Deletion of the edge (u, v). Maintain a queue of vertices to be processed to update the progress measure until the least fix-point is reached such that a vertex of V_2 is only added to the queue when its progress measure has increased. Initially, enqueue u. Then iteratively process and dequeue the vertices from the queue.

Case 1: A vertex x of V_1 is dequeued. Check whether given the current progress measure, the progress measure of x needs to be increased to satisfy the lift operation for x. To do this first check whether the list of x is empty. If it is not empty, nothing needs to be done. If it is empty, all remaining outedges of x are checked to compute the new progress measure value of x and the new list of x. Then all inedges (u, x) of x are processed using the following steps:

— If u is a player-1 non-Büchi vertex ($u \in V_1 \setminus B$), then it is enqueued (if it is not already in the queue) and x is removed from the list of u if it was there.
— If u is a player-2 non-Büchi vertex ($u \in V_2 \setminus B$), then check whether the change in the progress measure value of x also increases the progress measure value of u. If it does, then u is enqueued (if it is not already in the queue), otherwise u is *not* enqueued.
— If u is a player-1 Büchi vertex ($u \in V_1 \cap B$), then (i) if the progress measure of x is not $\top$, then do nothing; (ii) else remove x from the list of u, and if the list of u is empty, assign progress measure $\top$ to u and u is enqueued (if it is not already in the queue).
— If u is a player-2 Büchi vertex ($u \in V_2 \cap B$), then (i) if the progress measure of x is not $\top$, then do nothing; (ii) else assign progress measure $\top$ to u and u is enqueued (if it is not already in the queue).

Case 2: A vertex x of V_2 is dequeued. In this case the progress measure of x has increased and it has already been updated. Thus all what remains is to process all inedges (u, x) of x. The processing of the inedges is done exactly as in Case 1.

This algorithm is a generalization of the Even-Shiloach algorithm [Even and Shiloach 1981] for maintaining the breadth-first-search tree of a vertex b in an undirected graph. Assume $B = \{b\}$ and that $V = V_1$. Then the progress measure value of a vertex v is exactly v's level in the breadth-first search tree rooted at b (or equivalently its shortest path distance to b). Applying the lift operator to a vertex v is exactly the same as checking whether v has still an edge to an edge at level $level(v) - 1$ and if not, increasing the level of v by 1.

Correctness. Let G be a game graph, and let ρ^* be the least fix-point of Lift^G. Let $\overline{G} = G \setminus \{e\}$, where $e \in E \cap V_1 \times V$, be the game graph obtained by deleting a player-1 edge e. Let $\overline{\rho}^*$ be the least fix-point of $\overline{G}$. Let ρ^*_{new} be the new fix-point obtained by iterating $\mathsf{Lift}^{\overline{G}}$ on ρ^*. We will show that $\overline{\rho}^* = \rho^*_{\mathsf{new}}$.

LEMMA 2.17. *We have $\overline{\rho}^* \leq \rho^*_{\mathsf{new}}$.*

PROOF. Let ρ_0 be the progress measure that assings 0 to all vertices, i.e., the least progress measure. Clearly, $\rho_0 \leq \rho^*$. Let us denote by $(\mathsf{Lift}^{\overline{G}})^i$ the result of applying the lift operator i-times on $\overline{G}$, for some $i \in \mathbb{N}$. From a simple application of Lemma 2.15 it follows that $(\mathsf{Lift}^{\overline{G}})^i$ is monotonic. Hence we have $(\mathsf{Lift}^{\overline{G}})^i(\rho_0) \leq (\mathsf{Lift}^{\overline{G}})^i(\rho^*)$. Since $\overline{\rho}^* = (\mathsf{Lift}^{\overline{G}})^j(\rho_0)$ for some j, and $\rho^*_{\mathsf{new}} \geq (\mathsf{Lift}^{\overline{G}})^i(\rho^*)$ for all i (in particular for the j for which the least fix-point is obtained from ρ_0), it follows that $\overline{\rho}^* \leq \rho^*_{\mathsf{new}}$. ■

LEMMA 2.18. *We have $\rho^*_{\mathsf{new}} \leq \overline{\rho}^*$.*

PROOF. Observe that the graph $\overline{G}$ is obtained by deleting an edge for player-1, and hence the winning set for player 1 can only decrease and the minimal alternating distance to the Büchi set in the winning set can only increase. In other words, we have $\rho^* \leq \overline{\rho}^*$, i.e., the least fix-point of the graph G is smaller than the least fix-point of $\overline{G}$. Since $\rho^*_{\mathsf{new}} = (\mathsf{Lift}^{\overline{G}})^i(\rho^*)$, for some i, we have $\rho^*_{\mathsf{new}} = (\mathsf{Lift}^{\overline{G}})^i(\rho^*) \leq (\mathsf{Lift}^{\overline{G}})^i(\overline{\rho}^*) = \overline{\rho}^*$, where the first inequality is a consequence of Lemma 2.15 that $(\mathsf{Lift}^{\overline{G}})^i$ is monotonic, and the last equality is a consequence of the fact that $\overline{\rho}^*$ is a fix-point. Hence the desired result follows. ■

LEMMA 2.19. *We have $\rho^*_{\mathsf{new}} = \overline{\rho}^*$.*

Lemma 2.19 follows from Lemma 2.17 and Lemma 2.18. Lemma 2.19 and the fact that the algorithm implements the iteration of the lift operator on vertices one by one to compute the fix-point that is obtained by repeatedly applying the lift operator on the least fix-point of the previous game graph, along with Theorem 2.16, establishes the correctness of the algorithm.

Query operation. The query operation of whether a vertex v belongs to the winning set is answered in constant time by checking the progress measure of v. Additionally we can support the operation that requires to output *all* vertices of the winning set, in time proportional to the size of the winning set as follows. We maintain a list of winning vertices, and each vertex has a pointer to itself in the list; and when the progress measure of a vertex is set to $\top$ it is removed from the list. Thus the list of winning vertices can be output in time proportional to the size of the winning set.

Running time. The deletions of player-1 edges only decrease the winning set, and once a vertex is removed from the winning set (i.e., assigned value $\top$ in the progress measure algorithm), then it is never worked upon again. Upon termination, let ρ be the least fix-point in the end. The computation of the initial least fix-point is done in time $O(n^2)$. In the decremental algorithm we check for each dequeued player-1 vertex u whether its progress measure increases in constant time (by checking whether the list of u is empty). If it does not increase no further work is done for u. The constant

amount of work is charged to the edge deletion if an outedge of u was deleted. If no outedge of u was deleted then the progress measure of a vertex w with $(u, w) \in E$ must have increased and we charge the work to w. If the progress measure of u increases we spend time $O(|\mathsf{In}(u)|+|\mathsf{Out}(u)|)$ to determine the new progress measure of u, compute its new list, and process all its inedges, and the work is charged to u. A player-2 vertex u is only enqueued when its progress measure has increased. When it is dequeued we spend time $O(|\mathsf{In}(u)|)$ to process all its inedges, and charge it to u. The number of times the progress measure can increase for a vertex is at most $n + 1$ (as once it is $n + 1$ it is assigned $\top$). For a vertex v, let $\mathsf{Num}(v) = \rho(v)$, if $\rho(v) \neq \top$, and $n + 1$ otherwise. Hence the total work done by the algorithm is

$$O(\sum_{v \in V} \mathsf{Num}(v) \cdot |\mathsf{In}(v)|) + O(\sum_{v \in V} \mathsf{Num}(v) \cdot |\mathsf{Out}(v)|) = O(n \cdot m).$$

THEOREM 2.20. *Given an initial game graph with n vertices and m edges, the winning set partitions can be maintained under the deletion of $O(m)$ edges (u, v) with $u \in V_1$ in total time $O(n \cdot m)$.*

2.4.2. Incremental algorithm for Büchi games. We now present the details of the incremental algorithm for Büchi games, where we consider insertions of player-1 edges. The algorithm is similar to the decremental algorithm, but has several subtle changes (like it is based on the dual progress measure for player 2, and the case analysis of the algorithm is different from the decremental algorithm).

Previous results on dual progress measure [Jurdziński 2000]. The incremental algorithm will be based on the progress measure for coBüchi objectives. The progress measure for coBüchi objectives is simpler but different, and hence the incremental algorithm is simpler but different from the decremental algorithm. We start with the definition of a valid progress measure for player 2.

Valid progress measure for player 2. Consider a game graph with a set B of Büchi vertices. A progress measure ρ is a *valid* progress measure for player 2 if the following conditions hold for all $v \in V$:

$$\rho(v) \geq \begin{cases} \min_{(v,w)\in E} \rho(w) & v \in V_2 \setminus B; \\ \min_{(v,w)\in E} \rho(w) + 1 & v \in V_2 \cap B; \\ \max_{(v,w)\in E} \rho(w) & v \in V_1 \setminus B; \\ \max_{(v,w)\in E} \rho(w) + 1 & v \in V_1 \cap B. \end{cases}$$

We define the comparison operators $\leq, \geq$ on progress measures with the *pointwise* comparison.

Lift operation on progress measure. Given a game graph G, the function coLift^G, like the Lift^G function, takes as input a progress measure and returns a progress measure. For all input progress measures ρ, the output progress measure $\rho' = \mathsf{coLift}^G(\rho)$ is defined as follows: for all $v \in V$,

$$\rho'(v) = \begin{cases} \min_{(v,w)\in E} \rho(w) & v \in V_2 \setminus B; \\ \min_{(v,w)\in E} \rho(w) + 1 & v \in V_2 \cap B; \\ \max_{(v,w)\in E} \rho(w) & v \in V_1 \setminus B; \\ \max_{(v,w)\in E} \rho(w) + 1 & v \in V_1 \cap B. \end{cases}$$

LEMMA 2.21. *For all game graphs G, the function coLift^G is monotonic.*

PROOF. Consider progress measures ρ_1, ρ_2 such that $\rho_1 \leq \rho_2$. For a vertex v we have

$$\mathsf{coLift}^G(\rho_1)(v) = \begin{cases} \min\limits_{(v,w)\in E} \rho_1(w) \leq \min\limits_{(v,w)\in E} \rho_2(w) = \mathsf{coLift}^G(\rho_2)(v) & v \in V_2 \setminus B; \\ \min\limits_{(v,w)\in E} \rho_1(w) + 1 \leq \min\limits_{(v,w)\in E} \rho_2(w) + 1 = \mathsf{coLift}^G(\rho_2)(v) & v \in V_2 \cap B; \\ \max\limits_{(v,w)\in E} \rho_1(w) \leq \max\limits_{(v,w)\in E} \rho_2(w) = \mathsf{coLift}^G(\rho_2)(v) & v \in V_1 \setminus B; \\ \max\limits_{(v,w)\in E} \rho_1(w) + 1 \leq \max\limits_{(v,w)\in E} \rho_2(w) + 1 = \mathsf{coLift}^G(\rho_2)(v) & v \in V_1 \cap B; \end{cases}$$

where E is the set of edges in G. It follows that $\mathsf{coLift}^G(\rho_1) \leq \mathsf{coLift}^G(\rho_2)$. The desired result follows. ■

Since coLift^G is a monotonic function on a finite lattice, by Tarski-Knaster Theorem [Kechris 1995] it has a least fix-point. Before we proceed to the characterization, we present a definition: for a vertex $v \in W_2(\Psi)$, where Ψ is the coBüchi objective $\mathrm{coBuchi}(C)$, where $C = V \setminus B$ is the set of coBüchi vertices, let $\mathsf{maxvisit}(v) = \min_{\pi\in\Pi} \max_{\sigma\in\Sigma} |\{i \mid \omega(v,\sigma,\pi) = \langle v_0, v_1, v_2, \ldots\rangle, v_i \in B\}|$ denote the maximum number of visits to Büchi vertices. Since $v \in W_2(\Psi)$, once a winning strategy for player-2 is fixed, there cannot be a cycle with a Büchi vertex, and hence $\mathsf{maxvisit}(v) \leq n$. The result of [Jurdziński 2000] established that for all game graphs G, (i) there is a *unique* least fix-point ρ^* of coLift^G, (ii) the least fix-point is a valid progress measure, (iii) the least fix-point ρ^* fulfillis the following conditions: (a) all vertices in the winning set for player 1 are assigned the top element $\top$; and (b) for vertices v in the winning set for player 2 the progress measure equals $\mathsf{maxvisit}(v)$ (i.e., $\rho^*(v) = \mathsf{maxvisit}(v)$). The result of [Jurdziński 2000] is for the more general case of parity objectives, and the specialization to coBüchi objectives yields the above properties.

THEOREM 2.22 ([JURDZIŃSKI 2000]). *For all game graphs G, let ρ^* be the least fix-point of coLift^G, and let $||\rho^*|| = \{v \in V \mid \rho(v) \in [n]\}$ denote the set of vertices that are not assigned the top element. Then $||\rho^*|| = W_2(\Psi)$, where Ψ is the coBüchi objective.*

Incremental algorithm. We now present our incremental algorithm for player-1 edges (see discussion on motivation). Our algorithm initially computes the least fix-point progress measure ρ^* of coLift of the graph and then maintains it after each edge insertion by repeatedly applying the lift operator coLift to the fix-point ρ^* stored from *before* the edge insertion. To prove the correctness we will show that the fix-point obtained by repeatedly applying the lift operator on the previous least fix-point converges to the least fix-point of the new game graph. The algorithm maintains the following data structure: (i) For each vertex $x \in V_2 \setminus B$ it keeps a list of vertices w such that $(x,w) \in E$ and $\rho^*(x) = \rho^*(w)$ and (ii) for each vertex $x \in V_2 \cap B$ a list of vertices w such that $(x,w) \in E$ and $\rho^*(x) = \rho^*(w) + 1$. (iii) Every edge (x,w) has a pointer to its location in the list of x if it is stored in such a list. We next describe the algorithm in detail. We first describe the insertion of an edge as the initial fix-point computation is similar.

Insertion of the edge (u,v). Maintain a queue of vertices to be processed to update the progress measure until the least fix-point is reached such that a vertex of V_1 is only added to the queue when its progress measure has increased. Initially, enqueue u. Then iteratively process and dequeue the vertices from the queue.

Case 1: A vertex x of V_2 is dequeued. Check whether given the current progress measure, the progress measure of x needs to be increased to satisfy the lift operation for x. To do this we first check whether the list of x is empty. If it is not empty, nothing needs to be done. If it is empty, all remaining outedges of x are checked to compute the new progress measure value of x and the new list of x. Then all inedges (u,x) of x are processed as follows: If u is a player-2 vertex it is enqueued (if it is not already in the queue) and x is removed from the list of u if it was there. If u is a player-1 vertex then check whether the change in the progress measure value of x also increases

the progress measure value of u. If it does, then u is enqueued (if it is not already in the queue), otherwise u is *not* enqueued.

Case 2: A vertex x of V_1 is dequeued. In this case the progress measure of x has increased and it has already been updated. Thus all what remains is to process all inedges (u, x) of x as follows: If u is a player-2 vertex it is enqueued (if it is not already in the queue) and x is removed from the list of u if it was there. If u is a player-1 vertex then check whether the change in the progress measure value of x also increases the progress measure value of u. If it does, then u is enqueued (if it is not already in the queue), otherwise u is *not* enqueued.

Computation of the initial ρ^*. The computation of the initial ρ^* is similar to the incremental algorithm itself. We initialize the initial progress measure as 0 for all vertices, then enqueue the set of Büchi vertices, and proceed as the incremental algorithm until a fix-point is reached. As we start with the all 0 progress measure and repeatedly apply the lift operator we are guaranteed to reach the least fix-point. Then we compute for each vertex $v \in V_2$ its list.

Correctness. Let G be a game graph, and let ρ^* be the least fix-point of coLift^G. Let $\overline{G} = G \cup \{e\}$, where $e \in E \cap V_1 \times V$, be the game graph obtained by inserting a player-1 edge e. Let $\overline{\rho}^*$ be the least fix-point of $\overline{G}$. Let ρ^*_{new} be the new fix-point obtained by iterating $\mathsf{coLift}^{\overline{G}}$ on ρ^*. We will show that $\overline{\rho}^* = \rho^*_{\mathsf{new}}$.

LEMMA 2.23. *We have $\overline{\rho}^* \leq \rho^*_{\mathsf{new}}$.*

PROOF. Let ρ_0 be the progress measure that assings 0 to all vertices, i.e., the least progress measure. Clearly, $\rho_0 \leq \rho^*$. Let us denote by $(\mathsf{coLift}^{\overline{G}})^i$ the result of applying the lift operator i-times on $\overline{G}$, for some $i \in \mathbb{N}$. From a simple application of Lemma 2.21 it follows that $(\mathsf{coLift}^{\overline{G}})^i$ is monotonic. Hence we have $(\mathsf{coLift}^{\overline{G}})^i(\rho_0) \leq (\mathsf{coLift}^{\overline{G}})^i(\rho^*)$. Since $\overline{\rho}^* = (\mathsf{coLift}^{\overline{G}})^j(\rho_0)$ for some j, and $\rho^*_{\mathsf{new}} \geq (\mathsf{coLift}^{\overline{G}})^i(\rho^*)$ for all i (in particular for the j for which the least fix-point is obtained from ρ_0), it follows that $\overline{\rho}^* \leq \rho^*_{\mathsf{new}}$. ■

LEMMA 2.24. *We have $\rho^*_{\mathsf{new}} \leq \overline{\rho}^*$.*

PROOF. Observe that the graph $\overline{G}$ is obtained by inserting an edge for player-1, and hence the winning set for player 2 can only decrease and $\mathsf{maxvisit}(v)$ can only increase for vertices in the winning set for player 2. In other words, we have $\rho^* \leq \overline{\rho}^*$, i.e., the least fix-point of the graph G is smaller than the least fix-point of $\overline{G}$. Since $\rho^*_{\mathsf{new}} = (\mathsf{coLift}^{\overline{G}})^i(\rho^*)$, for some i, we have

$$\rho^*_{\mathsf{new}} = (\mathsf{coLift}^{\overline{G}})^i(\rho^*) \leq (\mathsf{coLift}^{\overline{G}})^i(\overline{\rho}^*) = \overline{\rho}^*,$$

where the first inequality is a consequence of Lemma 2.21 that $(\mathsf{coLift}^{\overline{G}})^i$ is monotonic, and the last equality is a consequence of the fact that $\overline{\rho}^*$ is a fix-point. Hence the desired result follows. ■

LEMMA 2.25. *We have $\rho^*_{\mathsf{new}} = \overline{\rho}^*$.*

Correctness. The correctness follows from Lemma 2.25, the fact that the algorithm implements the iteration of the lift operator on vertices one by one to compute the fix-point that is obtained by repeatedly applying the lift operator on the least fix-point of the previous game graph, and Theorem 2.22.

Query operation and running time. The query operation that is supported is whether a vertex v belongs to the winning set, and to output the set of winning vertices. The query operations are supported exactly as in the case of the decremental algorithm. The insertions of player-1 edges only decrease the winning set for player 2, and once a vertex is removed from the winning set (i.e., assigned value $\top$ in the progress measure algorithm), then it is never worked upon again. Upon termination, let ρ be the least fix-point in the end. In the incremental algorithm we check for each dequeued player-2 vertex u whether its progress measure increases in constant time (by checking

whether the list of u is empty). If it does not increase no further work is done for u. Since u is processed, the progress measure of a vertex w with $(u, w) \in E$ must have increased and we charge the work to w. If the progress measure of u increases, then we spend time $O(|\mathsf{In}(u)| + |\mathsf{Out}(u)|)$ to determine the new progress measure of u, compute its new list, and process all its inedges, and charge the work to u. A player-1 vertex u is only enqueued when its progress measure has increased, or an edge is inserted at u. If an edge was inserted, the work is charged to the inserted edge. Otherwise u is dequeued and we spend time $O(|\mathsf{In}(u)|)$ to process all its inedges, and charge it to u. The number of times the progress measure can increase for a vertex is at most $n+1$ (as once it is $n+1$ it is assigned $\top$). For a vertex v, let $\mathsf{Num}(v) = \rho(v)$, if $\rho(v) \neq \top$, and $n+1$ otherwise. Hence the total work done by the algorithm is

$$O(\sum_{v \in V} \mathsf{Num}(v) \cdot |\mathsf{In}(v)|) + O(\sum_{v \in V} \mathsf{Num}(v) \cdot |\mathsf{Out}(v)|) = O(n \cdot m).$$

An argument similar to the above also establishes that the initial least fix-point is computed in time $O(n \cdot m)$.

THEOREM 2.26. *Given an initial game graph with n vertices and m edges, the winning set partitions can be maintained under the insertion of $O(m)$ edges (u, v) with $u \in V_1$ in total time $O(n \cdot m)$.*

3. ALGORITHMS FOR MAXIMAL END-COMPONENTS DECOMPOSITION

In this section we present two improved static algorithms for computing the maximal end-component (mec) decomposition of an MDP P, and the first incremental and decremental algorithms to maintain the mec decomposition. We start with the basic definitions and preliminaries.

3.1. Definitions

We present the definitions as familiar in the MDP literature, though the relevant graph definitions are identical to the alternating game graphs defined in the previous section.

MDP graph and mec decomposition. For an MDP P, the MDP graph consists of 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. The vertices in V_1 are called player-1 vertices, and vertices in V_P are called random or probabilistic vertices. An edge $e = (u, v)$ is called a *player-1* edge if $u \in V_1$, and is called a *random* edge if $u \in V_P$. 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[4] for directed graphs ($V_P = \emptyset$) and closed recurrent sets for Markov chains ($V_1 = \emptyset$). A *bottom* scc C of a graph is a scc that has no edge leaving out of C.

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 outgoing edge, i.e. $\mathsf{Out}(v) \neq \emptyset$, because a vertex without outgoing 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

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

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, forms 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.

3.2. Algorithms for mec decomposition

In this subsection we present two improved algorithms for mec decomposition. We first define attractors, random set cuts, 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* $Attr_R(U, \mathsf{P})$ is defined inductively as follows: $U_0 = U$, and for $i \geq 0$, let $U_{i+1} = U_i \cup \{v \in V_P \mid \mathsf{O}ut(v) \cap U_i \neq \emptyset\} \cup \{v \in V_1 \mid \mathsf{O}ut(v) \subseteq U_i\}$. 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 $Attr_R(U, \mathsf{P}) = \bigcup_{i \geq 0} U_i$. The definition of *player-1 attractor* $Attr_1(U, \mathsf{P})$ is obtained by exchanging the role of random vertices and player-1 vertices in the above definition. Note that the definition of attractors are same as defined for alternating game graphs. A (random or player-1) attractor A can be computed in time $O(\sum_{v \in A} |In(v)|)$ [Beeri 1980; Immerman 1981].

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

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. In the Lemma 3.1 we show the following results: (1) In part 1 we show that if C is a scc in a MDP graph, U the set of random vertices in C with edges out of C, and Z the random attractor of U, then no non-trivial mec X intersects with Z and any edge from the mec X to Z must be a player-1 edge; and (2) in part 2 we show that if C is a mec, and Z the random attractor of C minus C, then no non-trivial mec X intersects with Z and all edges from X to Z is a player-1 edge.

LEMMA 3.1. *Let P be an MDP, and let (V, E) with partition (V_1, V_P) be the MDP graph.*

(*1*) *Let C be a scc in (V, E). Let $U = \{v \in C \cap V_P \mid \mathsf{O}ut(v) \cap (V \setminus C) \neq \emptyset\}$ be the random vertices in C with edges out of C. Let $Z = Attr_R(U, \mathsf{P}) \cap C$. 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 $Z = Attr_R(C, \mathsf{P}) \setminus C$. 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 a random set cut, and random vertices in U have edges out of C, we must have $X \cap U = \emptyset$. Thus we have the following two properties:
(1) *(Property 1.)* X is a random set cut (i.e., for all $u \in X \cap V_P$ we have $\mathsf{O}ut(u) \subseteq X$); and
(2) *(Property 2).* X does not contain any vertex in U (i.e., $X \cap U = \emptyset$).
We use the above two properties to show by induction that $X \cap Attr_R(U, \mathsf{P}) = X \cap Z = \emptyset$. 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 property 2 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 $U_{i+1} = U_i \cup \{v \in V_P \mid \mathsf{O}ut(v) \cap U_i \neq \emptyset\} \cup \{v \in V_1 \mid \mathsf{O}ut(v) \subseteq U_i\}$. Consider a vertex $u \in X$:

(1) 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 $\mathsf{Out}(u)$ is not a subset of U_i and hence $u \notin U_{i+1}$.
(2) If $u \in V_P$, then by property 1 we have $\mathsf{Out}(u) \subseteq X$ and by induction hypothesis we have $X \cap U_i = \emptyset$. Thus we have $\mathsf{Out}(u) \cap U_i = \emptyset$, 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 $X \cap Attr_R(U, \mathsf{P}) = X \cap Z = \emptyset$. 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. It follows that we must have $u \in V_1$.

— *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 a random set cut. Since X is a random set cut and X does not contain any vertex in C, it follows from the inductive proof of the previous case that $X \cap Attr_R(C, \mathsf{P}) = X \cap Z = \emptyset$, 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 3.2. *Let* P *be an MDP, and let* (V, E) *with partition* (V_1, V_P) *be the MDP graph. Let* C *be a scc in* (V, E) *such that for all* $v \in C \cap V_P$ *we have* $\mathsf{Out}(v) \subseteq C$*. Then* C *is a mec.*

PROOF. It follows that C is a random set cut, and since C is a scc it follows that C is a mec. ■

It is an easy corollary of Lemma 3.2 that every bottom scc is a mec.

Previous algorithm for maximal end-component decomposition. There were two previous iterative algorithms to compute an mec decomposition of an MDP. The first algorithm is as follows:

(1) Given an MDP P consider the MDP graph (V, E), and compute the scc decomposition and an increasing topological ordering of the scc's of (V, E) in $O(m)$ time.
(2) Consider the scc's C in increasing topological ordering (i.e., starting from the bottom scc's). If there is a random edge leaving C, then let U be the set of random vertices in C with edges out of C. Remove $Attr_R(U, \mathsf{P}) \cap C$ from the graph (by Lemma 3.1 these vertices belong to no mec), and then goto Step 1 with the new graph with the attractor removed.
(3) Output all scc's as mec's.

Observe that in the end all scc's C have no random edges going out of C are mec's (by Lemma 3.2). Each iteration takes $O(m)$ time and removes at least one vertex (by the random attractor). Thus the running time of the algorithm is $O(m \cdot n)$. We will refer this algorithm as the *first simple static* algorithm for mec decomposition. The second iterative algorithm is as follows:

(1) Given an MDP P consider the MDP graph (V, E), and compute the scc decomposition of (V, E) in $O(m)$ time.
(2) Include every bottom scc C to the list mec's (by Lemma 3.2 C is a mec). Remove $Attr_R(C, \mathsf{P})$ from the graph (by Lemma 3.1 these vertices belong to no mec), and then goto Step 1 with the new graph with the attractor removed.
(3) Output the list of mec's.

Note that there is always at least one such scc since every graph has a bottom scc. We remove $Attr_R(C, \mathsf{P})$ 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 *second simple static* algorithm for mec decomposition.

3.2.1. First improved algorithm. Our first improved algorithm for mec decomposition is obtained by combining the second 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 [Tarjan 1972], 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 parallel searches, in one step of 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. 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 second simple static algorithm, and (b) Case 2 is the lock-step exploration of a bottom scc. In Case 2 we start the lock-step exploration from a set of at most $\sqrt{m}$ vertices. At least one of them is in a bottom scc. Thus in time at most $O(\sqrt{m} \cdot |\cup_{u \in C} \mathsf{O}ut(u)|)$ we find a mec C, and amortize the cost over the edges of C. Between two consecutive executions of Case 1 it is ensured that at least $\sqrt{m}$ edges are removed from the graph, and thus Case 1 is executed at most $\sqrt{m}$-times. Thus we achived a $O(m \cdot \sqrt{m})$-time algorithm for the mec decomposition.

The details of the algorithm is as follows. 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, and we refer our algorithm as NEWMECALGO1.

Step 0. Repeat

(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 $Attr_R(U, \mathsf{P}) \cap C$ 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 $Attr_R(C, \mathsf{P})$ 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 Step 0.

(2) *Case 2.* Else ($|J_i| \leq \sqrt{m}$), then
   (a) We do a lock-step search using the scc algorithm of Tarjan [Tarjan 1972] from every vertex v in J_i. 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 identified as a mec and we remove $Attr_R(C, \mathsf{P})$ 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., $L_{i+1} := L_i \cup Attr_R(C, \mathsf{P})$, where C is the bottom scc removed in step 2(a) 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., $J_{i+1} := (J_i \setminus Attr_R(C, \mathsf{P})) \cup Q_i$, where Q_i is the subset of vertices of V_i with an edge to $Attr_R(C, \mathsf{P})$. 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 Step 0.

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

LEMMA 3.3. *The algorithm* NEWMECALGO1 *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 3.2) 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 3.1 and Lemma 3.2. ■

LEMMA 3.4. *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), it follows that 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(a) of iteration j. Hence we have a contradiction. It follows that we always have a vertex in J_i that is in a bottom scc. ■

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 3.5. *The running time of algorithm* NEWMECALGO1 *on an MDP* P *with m edges is $O(m \cdot \sqrt{m})$.*

PROOF. We now analyze the running time of NEWMECALGO1. 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 the scc decomposition can be computed in $O(m)$ time) [Tarjan 1972]. 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. By Lemma 3.4 for every bottom scc C there is a vertex in J_i that belong to C. Let C be the bottom scc discovered in iteration i while executing Case 2. Let $\mathsf{O}ut(C) = \bigcup_{v \in C} \mathsf{O}ut(v)$. The algorithm of [Tarjan 1972] 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 $O(|J_i| \cdot |\mathsf{O}ut(C)|) \leq O(\sqrt{m} \times |\mathsf{O}ut(C)|)$. Since C is identified as a mec and removed from the graph we charge the work of $O(\sqrt{m} \cdot |\mathsf{O}ut(C)|)$ to edges in $\mathsf{O}ut(C)$, 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 3.6. *Given an MDP* P*, the algorithm* NEWMECALGO1 *computes the mec decomposition of* P *in time $O(m \cdot \sqrt{m})$.*

3.2.2. Second improved algorithm. In this section we present an algorithm for the mec decomposition problem that runs in $O(n^2)$ time.

Notations. Given an MDP P, and the MDP graph $G = (V, E)$ with parition $(V_1, V_P))$, we will denote by $Reachable(X, G)$ the set of vertices that can reach a vertex in X in the graph (V, E). Note that $X \subseteq Reachable(X, G)$. Basically the algorithm is similar to NEWBUCHIALGO, and instead of searching for separating cuts, the algorithm for mec decomposition searches for bottom scc's. Specifically as before, we have $\log n$ graphs G_i such that $G_i = (V, E_i)$ and E_i contains all edges (u, v) where $\mathsf{o}utdeg(u) \leq 2^i$. We denote by G the full graph. We color vertices v in G_i *blue* if $\mathsf{o}utdeg(v) > 2^i$, i.e., $\mathsf{Bl}_i = \{v \in V \mid \mathsf{o}utdeg(v) > 2^i\}$ and all other vertices are colored *white*, i.e., $\mathsf{Wh}_i = \{v \in V \mid \mathsf{o}utdeg(v) \leq 2^i\}$. Note that $G = G_{\log n}$ and thus all vertices in $G_{\log n}$ are white. Thus, none of the outedges of the blue vertices of G_i belong to G_i, i.e., all blue vertices have outdegree 0 in G_i.

Second improved algorithm. Our second improved algorithm for mec decomposition of an MDP P consists of two nested loops, an outer loop with loop counter j and an inner loop with loop

counter i. The algorithm will iteratively delete vertices from the graph, and we denote by D_j the set of vertices deleted in iteration j. We will denote by G^j the graph in the beginning of iteration j, and its vertex set and edge set as V^j and E^j, respectively. We will denote by $G^j_i = (V^j, E^j_i)$ the sub-graph of $G^j = (V^j, E^j)$ where E^j_i contains all edges (u, v) where $|\mathsf{O}ut(u) \cap E^j| \leq 2^i$. The set Bl^j_i is the set of vertices in G^j_i with outdegree greater than 2^i in G^j_i. The steps of the algorithm NEWMECALGO2 are as follows. Below we denote by P^j the sub-MDP of P at the beginning of iteration j (in particular $\mathsf{P}^0 = \mathsf{P}$).

(1) Let D_j be the set of vertices deleted in iteration j. For $j := 0$, let $D_0 := Attr_R(X, \mathsf{P}^0)$, where X is the set of vertices that are in the bottom scc's in the initial graph G. Every bottom scc is an mec and included in the mec decomposition.
(2) Remove the vertices of D_j to obtain the graph G^j; $j := j + 1$. If all vertices are removed, then the whole algorithm terminates and outputs the mec decomposition.
(3) $i := 1$;
(4) repeat
  (a) Construct graph G^j_i. Compute the set $Y^j_i = Reachable(\mathsf{Bl}^j_i, G^j_i)$ of vertices in G^j_i that can reach the set Bl^j_i of blue vertices using the standard linear-time algorithm for reachability.
  (b) Let $S_j = V^j \setminus Y^j_i$ be the set of vertices that cannot reach the set Bl^j_i of blue vertices;
  (c) $i := i + 1$
(5) until S_j is non-empty
(6) if $S_j \neq \emptyset$, then let $D_j := Attr_R(X, \mathsf{P}^j)$, where X is the set of vertices that are in the bottom scc's in the sub-graph induced by S_j in G^j_i. Every bottom scc is an mec and included in the mec decomposition. Go to Step 2.

Basic correctness argument. Let us denote G^j to be the remaining game graph after iteration j. Let S_j be the set identified at iteration j, and let the inner iteration stop at i^*. All vertices in S_j are white, since $S_j = V^j \setminus Reachable(\mathsf{Bl}^j_{i^*}, G^j_{i^*})$ and $\mathsf{Bl}^j_{i^*} \subseteq Reachable(\mathsf{Bl}^j_{i^*}, G^j_{i^*})$. For all $v \in S_j$, all outedges from v end in a vertex in S_j: otherwise if there is an edge from v to $Reachable(\mathsf{Bl}^j_{i^*}, G^j_{i^*})$, then v would have been included in $Reachable(\mathsf{Bl}^j_{i^*}, G^j_{i^*})$. Hence any bottom scc in the subgraph induced by S_j in $G^j_{i^*}$ is also a bottom scc of G^j. The correctness of the identification of the bottom scc as an mec and the removal of the attractor follows from Lemma 3.1 and Lemma 3.2. The correctness of the algorithm follows.

LEMMA 3.7. *Algorithm* NEWMECALGO2 *correctly computes the mec decomposition of an MDP* P.

Running time analysis. The crucial result of the running time analysis depends on the following lemma. It shows that in an outer iteration j, if the inner iteration stops at iteration i^* and X is the set of vertices identified as bottom scc, then $X \cap \mathsf{Bl}^j_{i^*-1}$ is non-empty.

LEMMA 3.8. *Consider an outer iteration j of the algorithm, and let the inner iteration stop at iteration i^*. Let X be the set of vertices identified as bottom scc of the graph induced by S in $G^j_{i^*}$. Then $X \cap \mathsf{Bl}^j_{i^*-1} \neq \emptyset$.*

PROOF. Assume towards contradiction that there is a bottom scc C in the induced subgraph of S in $G^j_{i^*}$ such that $C \cap \mathsf{Bl}^j_{i^*-1} = \emptyset$. Now we consider the iteration $i^* - 1$ and then for every vertex in C in $G^j_{i^*-1}$ all outedges end in a vertex in C. Since C does not contain a vertex from $\mathsf{Bl}^j_{i^*-1}$ and C has no outgoing edges, it follows that $C \subseteq V^j \setminus Reachable(\mathsf{Bl}^j_{i^*-1}, G^j_{i^*})$. Since all edges of $G^j_{i^*-1}$ are contained in $G^j_{i^*}$ we have that $C \subseteq V^j \setminus Reachable(\mathsf{Bl}^j_{i^*-1}, G^j_{i^*-1})$. Hence a non-emptyset S_j would have been identified in iteration $i^* - 1$, and this contradicts that the algorithm stops at iteration i^* and not in $i^* - 1$. ∎

LEMMA 3.9. *The total time spent by* NEWMECALGO2 *is* $O(n^2)$.

PROOF. Assume that for an outer iteration j, the inner iteration stops the repeat until loop at value i^*. By the previous lemma, one of the vertices v in X must have belong to $\mathsf{BI}^j_{i^*-1}$ and thus it has outdegree at least 2^{i^*-1}. Since we identify the bottom scc that contain v it must contain all the endpoints of the outedges from v. Hence X contains at least 2^{i^*-1} vertices. The time spent for all the executions of the repeat loop in this iteration of the outer loop is the time spent in all graphs G^j_1, G^j_2, ..., $G^j_{i^*}$, which sums to $O(2^{i^*} \cdot n)$ (the graph construction is similar as in Section 2.3, and the reachability computation is linear time). We charge $O(n)$ to each deleted vertex. As the algorithm deletes at most n vertices the total time spent over the whole algorithm is $O(n^2)$. The removal of all the player-2 attractors over all iterations takes $O(m) = O(n^2)$ time. The result follows. ■

THEOREM 3.10. *Algorithm* NEWMECALGO2 *correctly computes the mec decomposition of an MDP* P *in* $O(n^2)$ *time.*

COROLLARY 3.11. *Given an MDP* P*, the mec decomposition can be computed in time* $O(\min\{m \cdot \sqrt{m}, n^2\})$*; and hence in time* $O(m \cdot n^{2/3})$.

3.3. Incremental and decremental algorithms

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 $u \in V_1$.

Motivation for dynamic algorithms. As in the case verification of open systems, in the verification of probabilistic systems it is natural that the systems under verification are developed incrementally by adding choices (or decrementally by removing choices) of the sytem till the objective is satisfied. The system choices are represented by player 1, whereas the probabilistic environment (or nature) is modeled by the random (or probabilistic) player, and the system design has no control over the environment choices. Hence dynamic algorithms with player-1 edge deletions or insertions are the relevant decremental and incremental algorithms required for MDPs.

3.3.1. Incremental algorithm for mec decomposition. We first present the basic idea of the incremental algorithm.

Basic idea. Since we consider insertions of player-1 edges, the insertions of edges can only merge mec's. Hence we collapse the mec's in a collapsed graph that has no non-trivial mec's. We then show that insertion of one player-1 edge in such a graph adds at most one non-trivial mec. We now present the notion of a collapsed graph.

Collapsed graph. 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^C_1, V^C_P) 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 $V^C_1 = \mathcal{C}_m \cup (V_C \cap V_1)$ and $V^C_P = V_C \setminus V^C_1$.

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

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^C_1, V^C_P) has the following property:

LEMMA 3.12. *The collapsed graph* G_C *with vertex partition* (V^C_1, V^C_P) *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. In a graph with no non-trivial mec the above fact also holds for insertions of random edges. However, in general graphs, the insertion of a random edge (u, v) with $u \in V_P$ can 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 3.13. *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 contain any non-trivial mec's before the insertion. Thus, the insertion can have created at most one new mec, namely the mec containing u and v. Furthermore, the mec decomposition of at most one scc, namely the scc containing u and v 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. In step 5 the algorithm performs a computation similar to random attractor computation, but ignorning self-loops for player-1 vertices (to ensure that trivial mec's are removed by the computation). The steps are as follows:

(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 $i \geq 0$, $Z_{i+1} = Z_i \cup \{v \in V_P \mid \mathsf{O}ut(v) \cap Z_i \neq \emptyset\} \cup \{v \in V_1 \mid \mathsf{O}ut(v) \cap C \subseteq Z_i \cup \{v\}\}$. [5]
(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 $|C'| \geq 2$ and C' is the new unique non-trivial mec. Update the data structures accordingly.

We now prove the crucial lemma that shows that if in step 6 we have $|C'| \geq 2$, then it is the new unique non-trivial mec.

[5] 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}.

LEMMA 3.14. *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 $\mathsf{Out}(u) \cap U \neq \emptyset$ (otherwise u would have been included in Z); (b) for all $u \in U \cap V_P$ we must have $\mathsf{Out}(u) \subseteq U$ (otherwise u would have been included in Z). It follows that every vertex in U has an outedge 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 $|C'| = 1$, then let $C' = \{v'\}$. 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 $\mathsf{Out}(v') \cap C \subseteq Z \cup \{v'\}$, and hence v' must have been 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 3.2 that C' is a non-trivial mec. Since by Lemma 3.13 it follows that there is at most one non-trivial mec, it follows that C' is the unique non-trivial mec. ■

By Lemma 3.1 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 3.14 there exists a unique bottom scc in $C \setminus Z$, which according to Lemma 3.2 is a mec. Since Lemma 3.13 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 Lemma 3.15.

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.

LEMMA 3.15. *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 $O(m)$.*

The collapsed graph, the incremental algorithm on the collapsed graph, and Lemma 3.15 gives us the desired result for the incremental algorithm. Our algorithm outputs the mec decomposition, or equivalently an integer for every vertex such that two vertices in the same mec has the same positive integer and vertices that do not belong to any mec is assigned a negative integer. Thus the query of whether two vertices belong to the same mec is answered in constant time.

THEOREM 3.16. *Given an MDP P and the maximal end-component decomposition of P, the new maximal end-component decomposition after the insertion of an edge (u, v) with $u \in V_1$ can be computed in time $O(m)$.*

3.3.2. Decremental algorithm for mec decomposition. We consider maintaining the mec decomposition of an MDP under edge deletion for player-1 vertices. The basic idea is to show that the decremental scc decomposition algorithm of Lacki [Lacki 2011] combined with the approach of the first simple static algorithm works in amortized time $O(n)$.

Decremental algorithm. We show that the first simple static algorithm can be modified to handle the deletion of an edge (u, v) with $u \in V_1$. 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. Hence at any point of the algorithm we will maintain edges (u, v) such that both u and v belong to the same mec, and all other edges will not be stored in our data structures. Given the mec decomposition of an MDP, we consider an edge deletion e for player 1. The basic idea is if the deletion of edge (u, v) splits the mec containing both u and v, then it also must split the scc containing u and v because (u, v) is a player-1 edge. In this case the decremental scc data structure of [Lacki 2011] will return all the new scc's and the edges between them in total time $O(n \cdot m)$ over all deletions. We spend time proportional to the number of new scc's to topologically

sort them and to check all new scc's in increasing topological order whether they form a mec or whether they have to be split further. The cost of this is charged to all the edges between the new scc's. Note that there are as many such edges as there are new scc's and each edge is charged in this way only once as it is removed for our data structure immediately afterwards. If a scc has to be split further, at least one vertex of the scc does not belong to any mec and will be removed from our data structure. We use the approach of the first simple static algorithm for mec decomposition to determine the new mec's and by repeating this step potentially removing multiple vertices from our data structure (as they do not belong to any mec's anymore). The total work of $O(k \cdot m)$, where k is the number of removed vertices in this way, is charged to the k deleted vertices, leading to a total time of $O(n \cdot m)$ over all deletions.

The details of the algorithm is as follows: we keep as data structures (a) a list of mec's and along with each mec the list of vertices in the mec and (b) the decremental scc data structure of [Lacki 2011] keeping all edges inside mec's and none of the other edges. Thus every scc in the data structure is also a mec. For an edge $e = (u, v)$ with $u \in V_1$, when e is deleted we execute the following steps: (A) If the edge e does not belong to any existing mec, then no action is required (as the edge e is not stored in our data structure); and (B) if the edge e belongs to a mec C, then we execute the following steps:

(1) Compute the scc decomposition of C using the decremental scc decomposition algorithm of [Lacki 2011]. Let L be an empty list.
(2) If C is a still a scc, then no action is required, otherwise remove C from the list of mec's and execute the following steps:
   (a) Compute an increasing topological ordering of the scc's (i.e., starting from bottom scc's) and add to L all the scc's created in this way. Remove all edges that do not belong to any scc from the decremental scc data structure.
   (b) While L is not empty
      i. Consider the next scc C' of L and remove it from L. If C' has a random edge leaving C', then execute the following step.
      ii. Let U be the non-empty set of random vertices with an edge leaving C'. Remove $A = Attr_R(U, G \restriction C)$ (i.e., all incoming and outgoing edges of vertices in A are removed, where $G \restriction C$ is the MDP induced by C). Set L as the empty list; compute the scc decomposition and an increasing topological ordering of the scc's and add to L all the scc's created in this way. Remove all edges that do not belong to any scc from the decremental scc data structure.
   (c) Add all the scc's to the list of mec's.

We now present the correctness and the amortized running time analysis.

Correctness. Like edge deletions in scc's, under player-1 edge deletions, the mec's of an MDP can only be decomposed into smaller mec's[6]. Hence it follows that if the deleted edge does not belong to any mec, then it can be simply removed. We now consider the case when the edge e deleted belongs to a mec C. Note that since C is a mec, before the edge deletion the following property holds: for all edges (u, v) with $u \in C \cap V_P$, we have $v \in C$. If C is a scc after the edge deletion, then it follows that it is still a random set cut, and hence C is an end-component. Since C is a mec before edge deletion, and mec's can only be decomposed into smaller mec's after edge deletion, it follows that C is a mec after edge deletion. This establishes the correctness of the step when C is still a scc. The correctness of the other part follows from correctness of the first simple static algorithm, in particular from the fact that in step 2(c) we output the maximal scc's that do not have any random edge leaving them.

Amortized running time analysis. We first observe that the amortized cost of maintaining the decremental scc decomposition over all iterations can be achieved in time $O(n \cdot m)$ [Lacki 2011]. More-

[6]Note that in case of deletions of random edges, smaller mec's can merge into larger mec's.

over the decremental scc decomposition algorithm of [Lacki 2011] can return the edges leaving a newly created scc (as required in step 2(a)) in total time $O(n \cdot m)$ as each edge is adjacent to a newly created scc at most n times.

Thus detecting the new scc's in step 1 and step 2(b)(ii) can be done in total time $O(n \cdot m)$. The topological sorting in step 2(a), considering the scc's in topological ordering in step 2(b)(i), and the if condition checks of step 2(b)(ii) of the first iteration of the while loop is charged to the edges between the newly created scc's. If no new scc has a random edge leaving, then the while loop stops without any further splitting. Each of the scc's in L is a mec in this case. Thus the work is proportional to the number of edges between the newly created scc's and can be charged to them. As each edge is adjacent to a newly created scc at most n times, this gives a total time of $O(n \cdot m)$.

In case that at least one "if" condition of step 2(b)(ii) is evaluated to true, then a non-empty random attractor (i.e., at least one vertex) is removed and step 2(b) is repeated. We charge the computation of the random attractor, the scc decomposition, the topological sort, the update of L, and all tests of the "if" statement until the next successful one or the termination of the while loop to the non-empty set of removed vertices. Thus $O(m)$ work is charged to the removed vertices. This can be repeated multiple times, each iteration being charged to the vertices removed in this iteration. Note that once a vertex is removed, it never belongs to a list of vertices in a mec again. Thus every vertex is charged at most once with $O(m)$ work. Hence the total time for all the iterations of all the while loops is $O(n \cdot m)$. This completes the proof of Theorem 3.17. Also note that as in the case of the incremental algorithm, the decremental algorithm also outputs the mec decomposition, and hence the query of whether two vertices belong to the same mec is answered in constant time.

THEOREM 3.17. *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 $O(n \cdot m)$.*

4. CONCLUSION

In this work we presented an improved ($O(n^2)$-time) static algorithm for alternating Büchi games improving the long-standing $\widetilde{O}(n \cdot m)$ barrier. Our result is obtained by a hierarchical graph construction technique, an improvement technique for algorithms based on attractors for games, and improves the complexity dependence on edges. The class of Büchi objectives are also special case of parity objectives. The classical algorithm for alternating games with parity objectives with d priorities (i.e., parity index d), is a recursive algorithm based on attractors (alternating reachability), and the base case of the recursive algorithm is alternating Büchi games. Thus our result for Büchi games immediately improves the complexity of the classical algorithm for parity games from $O(n^{d-1} \cdot m)$ to $O(n^d)$. However, the sub-exponential algorithm for parity games (with complexity $n^{O(\sqrt{n})}$) does not depend on the edge parameter, and our technique has no impact on it. The small progress measure algorithm for parity games [Jurdziński 2000] is also not based on attractors and our technique has no immediate impact. We also presented improved ($O(\min\{m \cdot \sqrt{m}, n^2\})$-time) static algorithms for maximal end-component decomposition improving the long-standing $O(n \cdot m)$ barrier. We also present the first incremental and decrmental algorithms for the problems, and for all the algorithms the amortized update time is linear, and match the best known complexity of simpler problems (such as decremental reachability in graphs and decremental scc decomposition in graphs). The most interesting open questions are as follows: (a) does there exist an $O(n \cdot m^{1-\epsilon})$-time or an $O(m \cdot n^{1-\epsilon})$-time algorithm for alternating Büchi games, for $\epsilon > 0$; and (b) can the maximal end-component decomposition problem be solved in time $O(n \cdot \sqrt{m})$.

Acknowledgements. We thank anonymous reviewers for helpful comments that helped us to improve the presentation of the paper. The research was supported by Austrian Science Fund (FWF) Grant No P 23499-N23 on Modern Graph Algorithmic Techniques in Formal Verification, Vienna Science and Technology Fund (WWTF) Grant ICT10-002, FWF NFN Grant No S11407-N23 (RiSE), ERC Start grant (279307: Graph Games), and Microsoft faculty fellows award.

REFERENCES

ALUR, R., HENZINGER, T., AND KUPFERMAN, O. 2002. Alternating-time temporal logic. *Journal of the ACM 49*, 672–713.

ALUR, R. AND TORRE, S. L. 2004. Deterministic generators and games for ltl fragments. *ACM Trans. Comput. Log. 5*, 1, 1–25.

BEERI, C. 1980. On the membership problem for functional and multivalued dependencies in relational databases. *ACM Trans. on Database Systems 5*, 241–259.

BIANCO, A. AND DE ALFARO, L. 1995. Model checking of probabilistic and nondeterministic systems. In *FSTTCS 95: Software Technology and Theoretical Computer Science*. Lecture Notes in Computer Science Series, vol. 1026. Springer-Verlag, 499–513.

BLOEM, R., GALLER, S. J., JOBSTMANN, B., PITERMAN, N., PNUELI, A., AND WEIGLHOFER, M. 2007. Interactive presentation: Automatic hardware synthesis from specifications: a case study. In *DATE*. 1188–1193.

BRÁZDIL, T., BROZEK, V., CHATTERJEE, K., FOREJT, V., AND KUČERA, A. 2011. Two views on multiple mean-payoff objectives in markov decision processes. In *LICS*. 33–42.

BÜCHI, J. 1960. Weak second-order arithmetic and finite automata. *Zeitschrift für mathematische Logik und Grundlagen der Mathematik 6*, 66–92.

BÜCHI, J. 1962. On a decision method in restricted second-order arithmetic. In *Proceedings of the First International Congress on Logic, Methodology, and Philosophy of Science 1960*, E. Nagel, P. Suppes, and A. Tarski, Eds. Stanford University Press, 1–11.

BÜCHI, J. AND LANDWEBER, L. 1969. Solving sequential conditions by finite-state strategies. *Transactions of the AMS 138*, 295–311.

CHANDRA, A. K., KOZEN, D., AND STOCKMEYER, L. J. 1981. Alternation. *J. ACM 28*, 1, 114–133.

CHATTERJEE, K. AND HENZINGER, M. 2011. Faster and dynamic algorithms for maximal end-component decomposition and related graph problems in probabilistic verification. In *SODA'11*. SIAM.

CHATTERJEE, K. AND HENZINGER, M. 2012. An O(n^2) algorithm for alternating Büchi games. In *SODA*. ACM-SIAM.

CHATTERJEE, K., HENZINGER, T., AND PITERMAN, N. 2006. Algorithms for Büchi games. In *Games in Design and Verification (GDV)*.

CHATTERJEE, K. AND HENZINGER, T. A. 2007a. Assume-guarantee synthesis. In *TACAS*. LNCS 4424, Springer. 261–275.

CHATTERJEE, K. AND HENZINGER, T. A. 2007b. Probabilistic systems with limsup and liminf objectives. In *ILC*. 32–45.

CHATTERJEE, K., HENZINGER, T. A., JOBSTMANN, B., AND SINGH, R. 2010. Measuring and synthesizing systems in probabilistic environments. In *CAV 10*. Springer.

CHATTERJEE, K., JURDZIŃSKI, M., AND HENZINGER, T. 2003. Simple stochastic parity games. In *CSL'03*. LNCS Series, vol. 2803. Springer, 100–113.

CHATTERJEE, K., JURDZIŃSKI, M., AND HENZINGER, T. 2004. Quantitative stochastic parity games. In *SODA'04*. SIAM, 121–130.

CHATTERJEE, K. AND RAMAN, V. 2012. Synthesizing protocols for digital contract signing. In *VMCAI*. LNCS 7148, Springer. 152–168.

CHURCH, A. 1962. Logic, arithmetic, and automata. In *Proceedings of the International Congress of Mathematicians*. Institut Mittag-Leffler, 23–35.

CONDON, A. 1992. The complexity of stochastic games. *Information and Computation 96(2)*, 203–224.

COURCOUBETIS, C. AND YANNAKAKIS, M. 1995. The complexity of probabilistic verification. *Journal of the ACM 42*, 4, 857–907.

DE ALFARO, L. 1997. Formal verification of probabilistic systems. Ph.D. thesis, Stanford University.

DE ALFARO, L., FAELLA, M., MAJUMDAR, R., AND RAMAN, V. 2005. Code-aware resource management. In *EMSOFT 05*. ACM.

DE ALFARO, L. AND HENZINGER, T. 2001. Interface automata. In *FSE'01*. ACM Press, 109–120.

DILL, D. 1989. *Trace Theory for Automatic Hierarchical Verification of Speed-independent Circuits*. The MIT Press.

EMERSON, E. AND JUTLA, C. 1991. Tree automata, mu-calculus and determinacy. In *FOCS'91*. IEEE, 368–377.

ETESSAMI, K., KWIATKOWSKA, M. Z., VARDI, M. Y., AND YANNAKAKIS, M. 2008. Multi-objective model checking of markov decision processes. *Logical Methods in Computer Science 4*, 4.

EVEN, S. AND SHILOACH, Y. 1981. An on-line edge-deletion problem. *J. ACM 28*, 1, 1–4.

FILAR, J. AND VRIEZE, K. 1997. *Competitive Markov Decision Processes*. Springer-Verlag.

GODHAL, Y., CHATTERJEE, K., AND HENZINGER, T. A. 2011. Synthesis of AMBA AHB from formal specification: A case study. *Journal of Software Tools Technology Transfer*.

HENZINGER, M. R., KING, V., AND WARNOW, T. 1999. Constructing a tree from homeomorphic subtrees, with applications to computational evolutionary biology. *Algorithmica 24*, 1, 1–13.

HOWARD, H. 1960. *Dynamic Programming and Markov Processes*. MIT Press.

IMMERMAN, N. 1981. Number of quantifiers is better than number of tape cells. *Journal of Computer and System Sciences 22*, 384–406.

JURDZIŃSKI, M. 2000. Small progress measures for solving parity games. In *STACS'00*. LNCS 1770, Springer, 290–301.

JURDZIŃSKI, M., KUPFERMAN, O., AND HENZINGER, T. A. 2002. Trading probability for fairness. In *CSL: Computer Science Logic*. Lecture Notes in Computer Science 2471. Springer, 292–305.

KECHRIS, A. 1995. *Classical Descriptive Set Theory*. Springer.

KRISHNAN, S. C., PURI, A., AND BRAYTON, R. K. 1994. Deterministic w automata vis-a-vis deterministic buchi automata. In *ISAAC*. LNCS 834, Springer. 378–386.

KUPFERMAN, O. AND VARDI, M. 2005. From linear time to branching time. *ACM Transactions on Computational Logic 6*, 2, 273–294.

KUPFERMAN, O. AND VARDI, M. Y. 1998. Freedom, weakness, and determinism: From linear-time to branching-time. In *LICS*. 81–92.

KWIATKOWSKA, M., NORMAN, G., AND PARKER, D. 2000. Verifying randomized distributed algorithms with prism. In *Workshop on Advances in Verification (WAVE'00)*.

LACKI, J. 2011. Improved deterministic algorithms for decremental transitive closure and strongly connected components. In *SODA*. 1438–1445.

MAHANTI, A. AND BAGCHI, A. 1985. AND/OR graph heuristic search methods. *JACM 32*, 1, 28–51.

MCNAUGHTON, R. 1993. Infinite games played on finite graphs. *Annals of Pure and Applied Logic 65*, 149–184.

PITERMAN, N., PNUELI, A., AND SA'AR, Y. 2006. Synthesis of reactive(1) designs. In *VMCAI*. LNCS 3855, Springer. 364–380.

PNUELI, A. AND ROSNER, R. 1989. On the synthesis of a reactive module. In *POPL'89*. ACM Press, 179–190.

POGOSYANTS, A., SEGALA, R., AND LYNCH, N. 2000. Verification of the randomized consensus algorithm of Aspnes and Herlihy: a case study. *Distributed Computing 13*, 3, 155–186.

RAMADGE, P. AND WONHAM, W. 1987. Supervisory control of a class of discrete-event processes. *SIAM Journal of Control and Optimization 25*, 1, 206–230.

SEGALA, R. 1995. Modeling and verification of randomized distributed real-time systems. Ph.D. thesis, MIT. Technical Report MIT/LCS/TR-676.

STOELINGA, M. 2002. Fun with FireWire: Experiments with verifying the IEEE1394 root contention protocol. In *Formal Aspects of Computing*.

TARJAN, R. E. 1972. Depth first search and linear graph algorithms. *SIAM J. Computing 1*, 2, 146–160.

THOMAS, W. 1997. Languages, automata, and logic. In *Handbook of Formal Languages*, G. Rozenberg and A. Salomaa, Eds. Vol. 3, Beyond Words. Springer, Chapter 7, 389–455.

VARDI, M. 2007a. Automata-theoretic model checking revisited. In *Proc. of Verification, Model Checking, and Abstract Interpretation*. Vol. LNCS 4349. Springer, 137–150.

VARDI, M. 2007b. The Büchi complementation saga. In *Proc. of Symp. on Theoretical Aspects of Computer Science*. Vol. LNCS 4393. Springer, 12–22.

ZIELONKA, W. 1998. Infinite games on finitely coloured graphs with applications to automata on infinite trees. In *Theoretical Computer Science*. Vol. 200(1-2). 135–183.