Improved Algorithms for One-Pair and k-Pair Streett Objectives

Krishnendu Chatterjee[*], Monika Henzinger[**], and Veronika Loitzenbauer[**]

[*]IST Austria
krishnendu.chatterjee@ist.ac.at

[**]University of Vienna
{monika.henzinger,veronika.loitzenbauer}@univie.ac.at

Abstract

The computation of the winning set for one-pair Streett objectives and for k-pair Streett objectives in (standard) graphs as well as in game graphs are central problems in computer-aided verification, with application to the verification of open systems, checking interface compatibility, well-formedness of specifications, the synthesis of systems from specifications, and the synthesis of reactive systems. We give faster algorithms for the computation of the winning set for (1) one-pair Streett objectives (aka parity-3 problem) in game graphs and (2) for k-pair Streett objectives in graphs. For both problems this represents the first improvement in asymptotic running time in 15 years.

Keywords: *(1) Graph games; (2) Streett objectives; (3) Graph algorithms; (4) Computer-aided verification.*

1 Introduction

Game graphs and graphs. Consider a directed graph (V, E) with a partition (V_1, V_2) of V, which 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 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. If $V_2 = \emptyset$, then we simply have a standard graph.

Objectives and winning sets. Objectives are subsets of infinite paths that specify the desired set of paths for player 1, and the objective for player 2 is the complement of player-1 objective (i.e., we consider *zero-sum* games). Given an objective Φ, an infinite path *satisfies the objective* if it belongs to Φ. Given a starting vertex $x \in V$ and an objective Φ, if player 1 can guarantee that the infinite path starting at x satisfies Φ, *no matter what choices player 2 makes,* then player 1 can *win from* x and x belongs to the *winning set of player 1*, and since the winning sets partition the game graph [31], the complement of the winning set for player 1 is the winning set for player 2. In case the game graph is a standard graph (i.e., $V_2 = \emptyset$), the *winning set* consists of those vertices x such that there exists an infinite path starting at x that satisfies Φ. The winning set computation for game graphs is more involved than for standard graphs due to the presence of the adversarial player 2.

Relevant objectives. The most basic objective is *reachability* where, given a set $U \subseteq V$ of vertices, an infinite path satisfies the objective if the path visits a vertex in U *at least once*. The next interesting objective is the *Büchi* objective that requires an infinite path to visit a vertex in U *infinitely often*. The next and a very central objective in formal verification and automata theory is the *one-pair Streett objective* that consists of a pair (L_1, U_1) of sets of vertices (i.e., $L_1 \subseteq V$ and $U_1 \subseteq V$), and an infinite path satisfies the objective iff the following condition holds: if some vertex in L_1 is visited infinitely often, then some vertex in U_1 is visited infinitely often (intuitively the objective specifies that if one Büchi objective holds, then

another Büchi objective must also hold). A generalization of one-pair Streett objectives is the *k-pair Streett objective* (aka *general Streett objective*) that consists of k-Streett pairs $(L_1, U_1), (L_2, U_2), \ldots, (L_k, U_k)$, and an infinite path satisfies the objective iff the condition for every Streett pair is satisfied (in other words the objective is the conjunction of k one-pair Streett objectives).

We study (1) game graphs with one-pair Streett objectives and (2) graphs with general Streett objectives.

Significance in verification. Two-player games on graphs are useful 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 [10, 34, 35], verification of open systems [2], checking interface compatibility [13], well-formedness of specifications [14], and many others. General and one-pair Streett objectives are central in verification as most commonly used specifications can be expressed as Streett automata [36, 40].

Game graphs with one-pair Streett objectives arise in many applications in verification. We sketch two of them. (A) Timed automaton games are a model for real-time systems. The analysis of such games with reachability objectives and safety objectives (which are the dual of reachability objectives) reduces to game graphs with one-pair Streett objectives [12, 11, 7, 9]. (B) The synthesis of Generalized Reactivity(1) (aka GR(1)) specifications exactly require the solution of game graphs with one-pair Streett objectives; GR(1) specifications are standard for hardware synthesis [33] and even used in synthesis of industrial protocols [22, 4][1].

General Streett objectives in standard graphs arise, for example, in the verification of closed systems with strong fairness conditions [29, 15, 21]. In program verification, a scheduler is *strongly fair* if every event that is enabled infinitely often is scheduled infinitely often. Thus, verification of systems with strong fairness conditions directly corresponds to checking the non-emptiness of Streett automata, which in turn corresponds to determining the winning set in standard graphs with Streett objectives. Note, however, that a Streett objective can either specify desired behaviors of the system or erroneous ones, and for erroneous specifications, it is useful to have a *certificate* (as defined below) to identify an error trace of the system [16, 29].

Note that *standard graphs* are relevant for testing the non-emptiness of Streett automata and the verification of *closed* systems, while *game graphs* are relevant for the synthesis and verification of *open* systems.

Previous results. We summarize the previous results for game graphs and graphs with Streett objectives.

Game graphs. We consider the computation of the winning set for player 1 in game graphs. For *reachability* objectives, the problem is PTIME-complete, and the computation can be achieved in time linear in the size of the graph [3, 25]. For *Büchi* objectives, the current best known algorithm requires $O(n^2)$ time [5, 6]. For *general Streett* objectives, the problem is coNP-complete [17], and for *one-pair Streett* objectives the current best known algorithm requires $O(nm)$ time [27]. One-pair Streett objectives also corresponds to the well-known parity games problem with three priorities (the parity games problem in general is in UP $\cap$ coUP [26]; it is one of the rare and intriguing combinatorial problems that lie in UP $\cap$ coUP, but not known to be in PTIME). Despite the importance of game graphs with one-pair Streett objectives in numerous applications and several algorithmic ideas to improve the running time for general parity games [41, 28, 37] or Büchi games [8, 5, 6], there has been no algorithmic improvement since 2000 [27] for one-pair Streett games.

Graphs. In standard graphs we study the computation of the winning set for general Streett objectives. If x belongs to the winning set, it is often useful to output a *certificate* for x. Let S be a (not necessarily maximal) strongly connected component (SCC) that is reachable from x such that for all $1 \leq j \leq k$ we have either $S \cap L_j = \emptyset$ or $S \cap U_j \neq \emptyset$ (i.e., if S contains a vertex from L_j then it also contains a vertex from U_j). A certificate is a "lasso-shaped" path that reaches S and then visits all vertices in S infinitely often to satisfy the general Streett objective. The basic algorithm [19, 30] for the winning set problem has

[1] A GR(1) specification expresses that if a conjunction of Büchi objectives holds, then another conjunction of Büchi objectives must also hold, and since conjunction of Büchi objectives can be reduced in linear time to a single Büchi objective, a GR(1) specification reduces to implication between two Büchi objectives, which is an one-pair Streett objective.

an asymptotic running time of $O((m+b)\min(n,k))$ with $b = \sum_{j=1}^{k}(|L_j| + |U_j|) \leq 2nk$. Within the same time bound Latvala and Heljanko [29] additionally compute a certificate of size at most $n\min(n,2k)$. Duret-Lutz et al. [15] presented a space-saving "on-the-fly" algorithm with the same time complexity for the slightly different transition-based Streett automata. The current fastest algorithm for the problem by Henzinger and Telle [24] from 1996 has a running time of $O(m\min(\sqrt{m\log n}, k, n) + b\min(\log n, k))$, however, given a start vertex x, to report the certificate for x it takes time $O(m\min(\sqrt{m\log n}, k, n) + b\min(\log n, k) + n\min(n,k))$.

Our contributions. In this work our contributions are two-fold.

Game graphs. We show that the winning set computation for game graphs with one-pair Streett objectives can be achieved in $O(n^{2.5})$ time. Our algorithm is faster for $m > n^{1.5}$, and breaks the long-standing $O(nm)$ barrier for dense graphs. We also discuss the implications of our algorithm for general parity games in Remark 1.

Graphs. We present an algorithm with $O((n^2+b)\log n)$ running time for the winning set computation in graphs with general Streett objectives, which is faster for $m \geq \max(n^{4/3}, b^{2/3})\log^{1/3} n$ and $k \geq n^2 m^{-1}\log n$. We additionally give an algorithm that computes a certificate for a vertex x in the winning set in time $O(m+n\min(n,k))$. We also provide an example where the certificate has size $\Theta(n\min(n,k))$, showing that no algorithm can compute and *output* a certificate faster. In contrast to [24] the running time of our algorithm for the winning set computation does not change with certificate reporting. Thus when certificates need to be reported and $k = \Omega(n)$, our algorithm is optimal up to a factor of $\log n$ as the size of the input is at least b and the size of the output is $\Omega(n^2)$.

Technical contributions. Both of our algorithms use a *hierarchical (game) graph decomposition* technique that was developed by Henzinger et al. [23] to handle *edge deletions* in *undirected* graphs. In [5, 6] it was extended to deal with *vertex deletions* in *directed* and *game* graphs. We combine and extend this technique in two ways.

Game graphs. The classical algorithm for one-pair Streett objectives repeatedly solves Büchi games such that the union of the winning sets of player 1 in the Büchi games is exactly the winning set for the one-pair Streett objective. Schewe [37] showed that an algorithm for parity games by Jurdziński [27] can be used to compute small subsets of the winning set of player 1, called *dominions*, and thereby improved the running time for general parity games. However his ideas do not improve the running time for one-pair Streett (aka *parity-3*) games. With this algorithm dominions of size h in Büchi games can be found in time $O(mh)$. We extend this approach by using the hierarchical game graph decomposition technique to find small dominions quickly and call the $O(n^2)$ Büchi game algorithm of [5, 6] for large dominions. This extension is possible as we are able to show that, rather surprisingly, it is sufficient to consider game graphs with $O(nh)$ edges to detect dominions of size h (see Lemma 2.5).

Graphs. In prior work that used the hierarchical graph decomposition technique the runtime analysis relied heavily on the fact that identified vertex sets that fulfilled a certain desired condition were *removed* from the (game) graph after their detection. The work for identifying the vertex set was then charged in an amortization argument to the removed vertex set. This is not possible for general Streett objectives on graphs, where SCCs are identified and some but not all of its vertices might be removed. As a consequence a vertex might belong to an identified SCC multiple times. We show how to overcome this difficulty by identifying, when an SCC S splits into multiple SCCs, an SCC $X \subset S$ whose size is at most half of the size of S. We identify X by using Tarjan's SCC algorithm [39] on the graph and its *reverse graph* in lock-step, thereby finding the smallest *top* (i.e. with no incoming edges) or *bottom* (i.e. with no outgoing edges) SCC contained in S. The smallest such SCC X has size at most $|S|/2$ and the algorithm takes time $O(|X|n)$ to find it, which we charge to the vertices in X. In this way, every time a vertex is "charged", the size of the identified vertex set to which it belongs is halved, guaranteeing that each vertex is charged only $O(\log n)$ times.

In Section 2 we present our algorithm for one-pair Streett objectives in game graphs, in Section 3 the algorithm for general Streett objectives in graphs.

2 One-Pair Streett Objectives in Game Graphs

2.1 Preliminaries

Parity games. A *parity game* $P = (G, \alpha)$ consists of a *game graph* $G = ((V, E), (V_{\mathcal{O}}, V_{\mathcal{E}}))$ and a *parity function* $\alpha : V \to \mathbb{Z}$ that assigns an integer value to each vertex. We denote the two players with $\mathcal{O}$ (for odd) and $\mathcal{E}$ (for even). Player $\mathcal{O}$ (resp. player $\mathcal{E}$) wins a play if the *lowest* priority occurring in the play infinitely often is odd (resp. even). We say that the vertices in $V_{\mathcal{O}}$ are $\mathcal{O}$-vertices and the vertices in $V_{\mathcal{E}}$ are $\mathcal{E}$-vertices. We use p to denote one of the players $\{\mathcal{O}, \mathcal{E}\}$ and $\bar{p}$ to denote his opponent. We will specifically consider *parity-3 games* with $\alpha : V \to \{-1, 0, 1\}$ and *Büchi games* with $\alpha : V \to \{0, 1\}$, where the vertices in the set $B = \{v \mid \alpha(v) = 0\}$ are called *Büchi vertices*. Büchi games are denoted as (G, B).

One-pair Streett and parity-3 games. A one-pair Streett objective with pair (L_1, U_1) is equivalent to a parity game with three priorities. Let the vertices in U_1 have priority -1, let the vertices in $L_1 \setminus U_1$ have priority 0 and let the remaining vertices have priority 1. Then player 1 wins the game with the one-pair Streett objective if and only if player $\mathcal{O}$ wins the parity-3 game. As the known algorithms for parity-3 games are special cases of algorithms for general parity games, we will use the notion of parity games (i.e., player $\mathcal{O}$ and player $\mathcal{E}$ instead of player 1 and player 2).

Plays. For technical convenience we consider that every vertex in the game graph G has at least one outgoing edge. A game is initialized by placing a token on a vertex. Then the two players form an infinite path called *play* in the game graph by moving the token along the edges. Whenever the token is on a vertex in V_p, player p moves the token along one of the outgoing edges of the vertex. Formally, a *play* is an infinite sequence $\langle v_0, v_1, v_2, \ldots \rangle$ of vertices such that $(v_j, v_{j+1}) \in E$ for all $j \geq 0$.

For a vertex $u \in V$, we write $Out(u) = \{v \in V \mid (u, v) \in E\}$ for the set of successor vertices of u and $In(u) = \{v \in V \mid (v, u) \in E\}$ for the set of predecessor vertices of u. We denote by $Outdeg(u) = |\, Out(u)|$ the number of outgoing edges from u, and by $Indeg(u) = |\, In(u)|$ the number of incoming edges.

Strategies. A *strategy* of a player $p \in \{\mathcal{O}, \mathcal{E}\}$ is a function that, given a finite prefix of a play ending at $v \in V_p$, selects a vertex from $Out(v)$ to extend the play. *Memoryless strategies* do not depend on the history of a play but only on the current vertex. That is, a memoryless strategy of player p is a function $\tau : V_p \to V$ such that $\tau(v) \in Out(v)$. It is well-known that for parity games it is sufficient to consider memoryless strategies (see Theorem 2.1 below). Therefore we will only consider memoryless strategies from now on. A start vertex v together with a strategy σ for $\mathcal{E}$ and a strategy π for $\mathcal{O}$ defines a unique play $\omega(v, \sigma, \pi) = \langle v_0, v_1, v_2, \ldots \rangle$, which is defined as follows: $v_0 = v$ and for all $j \geq 0$, if $v_j \in V_{\mathcal{E}}$, then $\sigma(v_j) = v_{j+1}$, and if $v_j \in V_{\mathcal{O}}$, then $\pi(v_j) = v_{j+1}$.

Winning strategies and sets. A strategy τ is winning for player p at start vertex v if the resulting play is winning for player p irrespective of the strategy of player $\bar{p}$. A vertex v belongs to the *winning set* W_p of player p if player p has a winning strategy from v. By the following theorem every vertex is winning for exactly one of the two players. When required for explicit reference of a specific game graph G or specific parity game P we use $W_p(G)$ and $W_p(P)$ to refer to the winning sets.

Theorem 2.1 ([18, 32]). *For every parity game the vertices V can be partitioned into the winning set $W_{\mathcal{E}}$ of $\mathcal{E}$ and the winning set $W_{\mathcal{O}}$ of $\mathcal{O}$. There exists a memoryless winning strategy for $\mathcal{E}$ (resp. $\mathcal{O}$) for all vertices in $W_{\mathcal{E}}$ (resp. $W_{\mathcal{O}}$).*

The algorithmic question for parity games is to compute the set $W_{\mathcal{E}}$. We will use the following algorithm for Büchi games as a subroutine in our algorithm.

Theorem 2.2 ([5, 6]). *Let (G, B) be a Büchi game with game graph G and Büchi vertices B. There is an algorithm* BÜCHI(G, B) *that computes $W_{\mathcal{E}}$ in time $O(n^2)$.*

For the analysis of our algorithm we further introduce the notions of *closed sets*, *attractors*, and *dominions*.

Closed sets and attractors. A set $U \subseteq V$ is *p-closed* if for all p-vertices u in U we have $Out(u) \subseteq U$ and for all $\bar{p}$-vertices v in U there exists a vertex $w \in Out(v) \cap U$. Note that player $\bar{p}$ can ensure that a play that currently ends in a p-closed set never leaves the p-closed set against any strategy of player p by choosing an edge (v, w) with $w \in Out(v) \cap U$ whenever the current vertex v is in $U \cap V_{\bar{p}}$ (see also [6, Proposition 2.2]). Given a game graph G and a p-closed set X, we will denote by $G[X]$ the game graph induced by the set X of vertices.

In a game graph G a *p-attractor* $Attr_p(U, G)$ of a set $U \subseteq V$ is the set of vertices from which player p has a strategy to reach U against all strategies of player $\bar{p}$. We have that $U \subseteq Attr_p(U, G)$. A p-attractor can be constructed inductively as follows: Let $R_0 = U$; and for all $i \geq 0$ let

$$R_{i+1} = R_i \cup \{v \in V_p \mid Out(v) \cap R_i \neq \emptyset\} \cup \{v \in V_{\bar{p}} \mid Out(v) \subseteq R_i\}. \quad (\ddagger)$$

Then $Attr_p(U, G) = \bigcup_{i \geq 0} R_i$. The lemma below summarizes some well-known facts about closed sets, attractors, and winning sets.

Lemma 2.3. *Let $p \in \{\mathcal{O}, \mathcal{E}\}$. The following assertions hold for parity games.*

1. *The set $V \setminus Attr_p(U, G)$ is p-closed in G [42, Lemma 4].*
2. *Let $U \subseteq V$ be p-closed. Then $Attr_{\bar{p}}(U, G)$ is p-closed [42, Lemma 5].*
3. *The attractor $Attr_p(U, G)$ can be computed in $O(\sum_{v \in Attr_p(U,G)} |In(v)|)$ time [3, 25].*
4. *Let $U \subseteq W_p(G)$ and let $A = Attr_p(U, G)$. Then $W_p(G) = A \cup W_p(G[V \setminus A])$ and $W_{\bar{p}}(G) = W_{\bar{p}}(G[V \setminus A])$ [28, Lemma 4.5].*

Dominions. A set of vertices $D \subseteq V$ is a *p-dominion* if $D \neq \emptyset$, player p has a winning strategy from every vertex in D that also ensures only vertices in D are visited, and D is a $\bar{p}$-closed set. We will only consider $\mathcal{E}$-dominions in this paper and therefore usually omit the reference to the player. Dominions of size $|D| \leq h$ can be computed by running the small-progress measure algorithm of Jurdziński [27] with a reduced codomain [37]. A description of the small-progress measure algorithm for Büchi games is given in [6, Section 2.4.1]. We will use the following algorithm as a subroutine.

Theorem 2.4 ([27, 37, 6])**.** *Let (G, B) be a Büchi game with game graph G and Büchi vertices B. There is an algorithm* BÜCHIPROGRESSMEASURE(G, B, h) *that returns the set of all dominions of size at most h in time $O(mh)$.*

2.2 Algorithm

In this section we present our new algorithm to compute the winning set of player $\mathcal{E}$ in a parity-3 game $P = (G, \alpha)$ in time $O(n^{2.5})$. Its complement is the winning set of player $\mathcal{O}$.

Initialization (Steps 1–3 of Algorithm PARITY-3*).* First the algorithm constructs the modified game graph $G' = ((V, E'), (V_{\mathcal{O}}, V_{\mathcal{E}}))$ from G. Let Z be the vertices in V with priority -1. In G' the vertices in Z are made *absorbing*, that is, the outgoing edges of the vertices in Z are replaced with self-loops. Otherwise G' contains the same edges as G. We will consider a Büchi game on G' where the vertices in Z have priority 1, and thus in the Büchi game there are only two priorities (priority 0 and 1). The construction of G' ensures that dominions in the Büchi game are also dominions in the parity-3 game P (see Lemma 2.10).

Iterated vertex deletions (Steps 4–9 of Algorithm PARITY-3*).* The algorithm will repeatedly remove vertices from the graphs G and G'. Initially the set V is the set of vertices in the input game graph G. During the algorithm, we denote with V the set of remaining vertices after vertex deletions and we denote with $G[V]$ and $G'[V]$ the subgraphs induced by the vertices remaining in V. The set of Büchi vertices B maintains the set of priority-0 vertices in V. The vertex set removal is achieved by identifying dominions and removing their attractors.

Dominion find and attractor removal. The algorithm repeatedly finds dominions in the Büchi game $(G'[V], B)$. After a dominion in the Büchi game $G'[V]$ is found, its $\mathcal{E}$-attractor in $G[V]$ is removed from V and B. Then the search for dominions is continued on the remaining vertices. If all vertices in the

ALGORITHM PARITY-3: New Algorithm for Parity-3 aka One-Pair Streett Objective

Input : a game graph $G = ((V, E), (V_{\mathcal{O}}, V_{\mathcal{E}}))$ and a priority function $\alpha : V \to \{-1, 0, 1\}$
Output : the winning set $W_{\mathcal{E}}$ of player $\mathcal{E}$

1 $Z = \{v \in V \mid \alpha(v) = -1\}$; $E' = \{(u, u) \mid u \in Z\} \cup \{(u, v) \in E \mid u \in V \setminus Z\}$
2 $G' = (V, E')$ /* vertices with $\alpha = -1$ are absorbing in G' */
3 $W \leftarrow \emptyset$; $B \leftarrow \{v \in V \mid \alpha(v) = 0\}$
4 **repeat**
5 $D \leftarrow$ BÜCHIDOMINION$(G'[V], B, \sqrt{n})$
6 **if** $D = \emptyset$ **then** $D \leftarrow$ BÜCHI$(G'[V], B)$
7 $A \leftarrow Attr_{\mathcal{E}}(D, G[V])$; $W \leftarrow W \cup A$
8 $V \leftarrow V \setminus A$; $B \leftarrow B \setminus A$
9 **until** $D = \emptyset$
10 **return** W
11 **Procedure** BÜCHIDOMINION$(G' = ((V, E'), (V_{\mathcal{O}}, V_{\mathcal{E}})), B, h_{max})$
12 **for** $i \leftarrow 1$ **to** $\lceil \log(2h_{max}) \rceil$ **do**
13 construct G'_i; $Bl_i \leftarrow \{v \in V_{\mathcal{O}} \mid Outdeg(v) > 2^i\}$
14 $Y_i \leftarrow Attr_{\mathcal{O}}(Bl_i, G'_i)$
15 $\mathcal{D}_i \leftarrow$ BÜCHIPROGRESSMEASURE$(G'_i[V \setminus Y_i], B \setminus Y_i, 2^i)$
16 **if** $\mathcal{D}_i \neq \emptyset$ **then return** *union of dominions in* $\mathcal{D}_i$
17 **return** $\emptyset$

Büchi game are winning for $\mathcal{O}$, i.e., no dominion exists in the Büchi game, then Algorithm PARITY-3 terminates. The winning set of player $\mathcal{E}$ is the union of the $\mathcal{E}$-attractors of all found dominions. The remaining vertices are winning for player $\mathcal{O}$. We now describe the steps to find dominions.

Steps of dominion find. For the search for dominions in the Büchi game $(G'[V], B)$ we use two different algorithms, BÜCHI and BÜCHIDOMINION. We first search for "small" dominions with up to $O(h_{\max})$ vertices with $h_{\max} = \sqrt{n}$ with Procedure BÜCHIDOMINION. If no dominion is found, we can conclude that either all dominions contain more than $\sqrt{n}$ vertices or the winning set of $\mathcal{E}$ on the current game graph is empty (in this case the algorithm terminates). The former case occurs at most $\sqrt{n}$ times and in such a case we use the $O(n^2)$ algorithm BÜCHI (Theorem 2.2) to obtain a dominion. Below we describe the details of BÜCHIDOMINION.

Graph decomposition for BÜCHIDOMINION. In the Procedure BÜCHIDOMINION we use the following graph decomposition. For a game graph $G' = ((V, E'), (V_{\mathcal{O}}, V_{\mathcal{E}}))$ we denote its decomposition with $\{G'_i\}$. We consider the incoming edges of each vertex in E' in a fixed order: First the edges from vertices in $V_{\mathcal{E}}$, then the remaining edges. We construct $\log n$ graphs $G'_i = (V, E'_i)$, $1 \leq i \leq \log n$, where the set of edges E'_i contains for each vertex $v \in V$ with $Outdeg(v) \leq 2^i$ all its outgoing edges in E' and in addition for each vertex $v \in V$ its first 2^i incoming edges in E'. Note that (1) $E'_i \subseteq E'_{i+1}$, (2) $|E'_i| \leq 2^{i+1}n$, and (3) $G'_{\log n} = G'$. We color $\mathcal{O}$-vertices v with $Outdeg(v) > 2^i$ *blue* in G'_i and denote the set of blue vertices with Bl_i. We call vertices with $Outdeg(v) \leq 2^i$ *white*.

Procedure BÜCHIDOMINION *(Steps 11–17 of Algorithm* PARITY-3*).* The Procedure BÜCHIDOMINION searches for dominions in the subgraphs G'_i, starting at $i = 1$. The index i is increased one by one up to at most $i = \lceil \log(2h_{\max}) \rceil$ (with $h_{\max} = \sqrt{n}$) as long as no dominion was found. Let Y_i be the $\mathcal{O}$-attractor of blue vertices in G'_i, i.e., of $\mathcal{O}$-vertices that are missing outgoing edges in G'_i. To ensure that dominions found in the subgraph G'_i are also dominions in G', only the vertices in $V \setminus Y_i$ are considered. The BÜCHIPROGRESSMEASURE algorithm (Theorem 2.4) is used to find dominions of size at most $O(2^i)$ in $G'_i[V \setminus Y_i]$.

The following key lemma describes the central connection between dominions of a certain size and our graph decomposition. Namely, if a dominion D is found in G'_i but not in G'_{i-1}, then $Attr_{\mathcal{E}}(D, G')$

contains more than 2^{i-1} vertices. This has the remarkable consequence, detailed in Corollary 2.6, that every dominion of size h can be found by searching for a dominion in G'_i with $i = \lceil \log(2h) \rceil$. This will be crucial for our runtime analysis.

Lemma 2.5. *Let $G' = ((V, E'), (V_{\mathcal{O}}, V_{\mathcal{E}}))$ be a game graph and $\{G'_i\}$ its graph decomposition. For $1 \leq i \leq \log n$ we define the following sets: the set of blue vertices $Bl_i = \{v \in V_{\mathcal{O}} \mid Outdeg(v) > 2^i\}$, the attractor of blue vertices $Y_i = Attr_{\mathcal{O}}(Bl_i, G'_i)$, and the set of dominions $\mathcal{D}_i = \text{BÜCHIPROGRESSMEASURE}(G'_i[V \setminus Y_i], B \setminus Y_i, 2^i)$. If a dominion D is contained in $\mathcal{D}_i$ but not in $\mathcal{D}_{i-1}$, then $Attr_{\mathcal{E}}(D, G')$ contains more than 2^{i-1} vertices.*

Proof. We distinguish three cases:

Case 1: The dominion D contains more than 2^{i-1} vertices. This situation might arise as Procedure $\text{BÜCHIPROGRESSMEASURE}(G'_{i-1}[V \setminus Y_{i-1}], B \setminus Y_{i-1}, 2^{i-1})$ only guarantees to detect dominions of size at most 2^{i-1}. In this case the lemma is satisfied trivially.

Case 2: The dominion D contains a vertex $v \in V_{\mathcal{O}}$ that is blue in G'_{i-1}, i.e., an $\mathcal{O}$-vertex with more than 2^{i-1} outgoing edges. Since D is $\mathcal{O}$-closed, we have $Out(v) \subseteq D$. Thus $|Attr_{\mathcal{E}}(D, G')| \geq |D| > 2^{i-1}$ in this case.

Case 3: All vertices $v \in V_{\mathcal{O}}$ in D are white in G'_{i-1} and thus the outgoing edges of the $\mathcal{O}$-vertices in D are the same in G'_{i-1} and G'_i. There are two subcases.

Case 3a: All edges (u, v) from vertices $u \in V_{\mathcal{E}} \cap D$ to vertices $v \in D$ that are present in G'_i are also present in G'_{i-1}. Let σ be the winning strategy of $\mathcal{E}$ for the vertices in D found in G'_i. This implies that (i) D is $\mathcal{O}$-closed in G'_{i-1} and (ii) all edges (u, v) with $u \in D \cap V_{\mathcal{E}}$ and $v = \sigma(u)$ are contained in G'_{i-1}. Thus σ is also a winning strategy of $\mathcal{E}$ for the vertices in D in G'_{i-1}. Hence the set D is a dominion in G'_{i-1}. Thus either Case 1 applies or the dominion would already have been detected in iteration $i - 1$, a contradiction.

Case 3b: There exists a vertex $u \in V_{\mathcal{E}} \cap D$ that has an outgoing edge (u, v) to a vertex $v \in D$ in G'_i but not in G'_{i-1}. This implies $Indeg(v) > 2^{i-1}$. By the ordering of the incoming edges and the fact that $u \in V_{\mathcal{E}}$, at least 2^{i-1} edges in $In(v)$ emanate from vertices in $V_{\mathcal{E}}$. By the definition of an attractor, all these vertices are contained in $Attr_{\mathcal{E}}(D, G')$. Thus we have $|Attr_{\mathcal{E}}(D, G')| > 2^{i-1}$ as required. ∎

Corollary 2.6. *Let G', $\{G'_i\}$, Bl_i, Y_i, and $\mathcal{D}_i$ be defined as in Lemma 2.5. Let D be a dominion in G' with $D = Attr_{\mathcal{E}}(D, G')$ and $h = |D|$. Then for $i = \lceil \log(2h) \rceil$ the set of dominions $\mathcal{D}_i$ contains D.*

Proof. By the definition of i we have $2^{i-2} < h \leq 2^{i-1}$. Assume by contradiction that $\mathcal{D}_i$ does not contain D. Since $G' = G'_{\log n}$ and $Y_{\log n} = \emptyset$, by Theorem 2.4 there exists some i' with $i < i' \leq \log n$, such that $\mathcal{D}_{i'}$ contains D. Let i^* be the smallest i' such that $\mathcal{D}_{i'}$ contains D. Note that $i^* > i$. We have that $D \notin \mathcal{D}_{i^*-1}$. By Lemma 2.5 this implies $|Attr_{\mathcal{E}}(D, G')| > 2^{i^*-1}$, a contradiction to $h \leq 2^{i-1}$. ∎

Corollary 2.7. *Either the Procedure $\text{BÜCHIDOMINION}(G'[V], B, h_{max})$ returns a dominion or every dominion D in $G'[V]$ with $D = Attr_{\mathcal{E}}(D, G'[V])$ has size greater than h_{max}.*

In the runtime analysis we will additionally use the following lemma, which follows from the inductive construction of attractors.

Lemma 2.8. *Let the game graphs G and G' and the vertex set V be defined as in Algorithm* PARITY-3. *Then for a player $p \in \{\mathcal{E}, \mathcal{O}\}$ and every set $U \subseteq V$ it holds that $Attr_p(U, G'[V]) \subseteq Attr_p(U, G[V])$.*

Proof. Let us consider the attractor computation $Attr_p(U, G'[V])$ and $Attr_p(U, G[V])$ as defined in (‡), and let us call the respective sequences as R'_i and R_i respectively. By the definition of G' for every vertex v in V either (1) $Out'(v) = Out(v)$ or (2) $Out'(v) = \{v\}$. It is straightforward to prove by induction that $R'_i \subseteq R_i$ and the desired result follows. ∎

Lemma 2.9 (Runtime)**.** *Algorithm* PARITY-3 *can be implemented in $O(n^{2.5})$ time.*

Proof. Algorithm PARITY-3 can be initialized in $O(m)$ time as the graph G' and the set B can be constructed from G in linear time. Note that the number of edges m' in G' is at most the number of edges m in G.

For the operations in the repeat-until loop we analyze the *total* running time over all iterations of the loop. The runtime analysis relies heavily on the fact that when a dominion D is identified, the vertices in $Attr_{\mathcal{E}}(D, G)$ and their incident edges are removed from G and G'. In combination with Corollary 2.7, this ensures that BÜCHI is called at most $O(n/h_{\max})$ times. By Theorem 2.2 one call to BÜCHI takes time $O(n^2)$. With $h_{\max} = \sqrt{n}$ we obtain a total time spent in BÜCHI of $O(n^{2.5})$.

To analyze the total time spent in BÜCHIDOMINION, we first show how to efficiently construct the graph decomposition $\{G'_i\}$ of G'. We maintain the following data structure for G' over all iterations of Algorithm PARITY-3. At each vertex v of G' we maintain (a) a sorted list of inedges $In(v)$, and (b) a list of outedges $Out(v)$. Additionally we maintain for each edge (u, v) a pointer to its position in the inlist of v and the outlist of v. This allows us to update the data structure in time proportional to the degree of v when a vertex v is removed. As each vertex can be deleted at most once, the total time to update this data structure is bounded by $O(n + m)$. We next analyze the time needed per iteration i of the for-loop in BÜCHIDOMINION. Given the above data structure, the graph G'_i, the set of blue vertices Bl_i, and the attractor $Y_i = Attr_{\mathcal{O}}(Bl_i, G'_i)$ can be constructed in time $O(n \cdot 2^i)$. By Theorem 2.4 the time for one call of the subroutine BÜCHIPROGRESSMEASURE$(., ., 2^i)$ on graph G'_i, is $O(n \cdot 2^i \cdot 2^i) = O(n \cdot 2^{2i})$.

Let i^* be the iteration at which Procedure BÜCHIDOMINION stops after it is called by Algorithm PARITY-3. The runtime for this call to Procedure BÜCHIDOMINION from $i = 1$ to i^* forms a geometric series that is bound by $O(n \cdot 2^{2i^*})$. By Lemmata 2.5 and 2.8 and Corollary 2.7 either (1) a dominion D with $|Attr_{\mathcal{E}}(D, G)| > 2^{i^*-1}$ vertices was found by BÜCHIDOMINION or (2) all dominions in G' have more than $h_{\max}$ vertices or there are no more dominions in G'. Thus either (2a) a dominion D with more than $h_{\max}$ vertices is detected in the subsequent call to BÜCHI or (2b) there is no dominion in G' and this is the last iteration of Algorithm PARITY-3. Case (2b) can happen at most once and its runtime is bounded by $O(n \cdot 2^{2\log(2h_{\max})}) = O(n^2)$. In the cases (1) and (2a) more than 2^{i^*-2} vertices are removed from the graph in this iteration, as $h_{\max} > 2^{i^*-2}$. We charge each such vertex $O(n \cdot 2^{i^*}) = O(n \cdot h_{\max})$ time. Hence the total runtime for these cases is $O(n^2 \cdot h_{\max}) = O(n^{2.5})$.

It remains to consider the total time needed to compute $A = Attr_{\mathcal{E}}(D, G[V])$. By Lemma 2.3. (3) the attractor A can be computed in time $O(\sum_{v \in A} |In(v)|)$. Since the edges adjacent to vertices in A are removed from G after the iteration in which D was found, this attractor computation can be done in total time $O(m)$. We conclude that the runtime of Algorithm PARITY-3 is $O(n^{2.5})$. ∎

We will show the correctness of Algorithm PARITY-3 by first proving that every dominion found in the Büchi game on G' is indeed a dominion in the parity-3 game on G. Together with Lemma 2.3. (4) this implies that the computed set W is indeed a part of the winning set of player $\mathcal{E}$ in the parity-3 game. We then provide a winning strategy for player $\mathcal{O}$ for all remaining vertices.

Lemma 2.10. *Let the game graphs G and G' and the vertex sets V and B be defined as in Algorithm* PARITY-3. *If D is a dominion in the Büchi game $(G'[V], B)$, then D is a dominion in the parity-3 game $P = (G[V], \alpha)$.*

Proof. Let Z be the vertices in V with priority α equal to -1. The vertices in Z have priority 1 in the Büchi game, i.e., $Z \cap B = \emptyset$. Whenever a play in $G'[V]$ reaches a vertex u in Z, only u will be visited in the subsequent play since $Out(u) = \{u\}$. Thus no vertex in Z is winning for $\mathcal{E}$ in $(G'[V], B)$, i.e., $D \cap Z = \emptyset$. Hence for all vertices in D the outgoing edges are the same in $G[V]$ and $G'[V]$. Thus D is $\mathcal{O}$-closed in $G[V]$ and the winning strategy of player $\mathcal{E}$ for D in the Büchi game $(G'[V], B)$ is also winning for player $\mathcal{E}$ for all vertices in D in the parity-3 game P. ∎

Lemma 2.11 (Correctness)**.** *Given a parity-3 game P, let W be the output of Algorithm* PARITY-3. *We have: (1)* (Soundness). *$W \subseteq W_{\mathcal{E}}(P)$; and (2)* (Completeness). *$W_{\mathcal{E}}(P) \subseteq W$.*

Proof. The first part on soundness follows from Lemmata 2.10 and 2.3. (4). We now prove the completeness result. Given the output W, let $\overline{W}$ denote the complement set. When Algorithm PARITY-3 terminates,

the winning set of player $\mathcal{E}$ in the Büchi game $(G'[\overline{W}], B)$ is empty (otherwise the algorithm would not have terminated). Also note that since the algorithm removes attractors for $\mathcal{E}$, the set $\overline{W}$ is closed for $\mathcal{E}$ (by Lemma 2.3. (1)). Consider the set $Z = \{v \in \overline{W} \mid \alpha(v) = -1\}$, its attractor $X = Attr_{\mathcal{O}}(Z, G[\overline{W}])$, and the subgame induced by $U = \overline{W} \setminus X$. Note that in U the game graphs G and G' coincide. Thus all vertices in U must be winning for player $\mathcal{O}$ in the Büchi game $(G[U], B)$ as otherwise $W_{\mathcal{E}}$ would have been non-empty for $(G'[\overline{W}], B)$. We prove the lemma by describing a winning strategy for player $\mathcal{O}$ in P for all vertices in $\overline{W}$. Since $\overline{W}$ is $\mathcal{E}$-closed, for vertices in $Z \cap V_{\mathcal{O}}$, the winning strategy chooses an edge in $\overline{W}$. For vertices in X player $\mathcal{O}$ follows his attractor strategy to Z. In the subgame induced by $U = \overline{W} \setminus X$ player $\mathcal{O}$ follows his winning strategy in the Büchi game $(G[U], B)$. Then in a play either (i) X is visited infinitely often; or (ii) from some point on only vertices in U are visited. In the former case, the attractor strategy ensures that then some vertex in Z with priority -1 is visited infinitely often; and in the later case, the subgame winning strategy ensures that only vertices with priority 1 and no vertices with priority 0 are visited infinitely often. It follows that $\overline{W} \subseteq W_{\mathcal{O}}(P)$, i.e., $W_{\mathcal{E}}(P) \subseteq W$, and the desired result follows. ∎

Lemmata 2.9 and 2.11 yield the following result.

Theorem 2.12. *Algorithm* PARITY-3 *correctly computes the winning sets in parity-3 games in $O(n^{2.5})$ time.*

Computation of winning strategies. In parity-3 games the previous results for computing winning strategies for the players in their respective winning sets are as follows: The small-progress measure algorithm of [27] requires $O(nm)$ time to compute the winning strategy of the player whose parity is equal to the parity of the lowest priority and $O(n^2m)$ time to compute the respective winning strategies for both players; Schewe [38] shows how to modify the small-progress measure algorithm to compute the respective winning strategies of both players in $O(nm)$ time. We show that our algorithm also computes the respective winning strategies in $O(n^{2.5})$ time. We first observe that the algorithm of [6] that solves Büchi games in $O(n^2)$ time also computes the respective winning strategies of both players (the algorithm is based on identifying traps and attractors, and the corresponding winning strategies are identified immediately with the computation). In Lemma 2.11 we describe the strategy computation for a winning strategy for player $\mathcal{O}$ which involves an attractor strategy and the sub-game strategy for Büchi games, each of which can be computed in $O(n^2)$ time. A winning strategy for player $\mathcal{E}$ is obtained in the iterations of the algorithm, i.e., whenever we obtain a dominion by solving Büchi games we also obtain a corresponding winning strategy, and similarly for the attractor computation. Thus the winning strategy for player $\mathcal{E}$ can be computed in $O(n^{2.5})$ time.

Corollary 2.13. *Winning strategies for player $\mathcal{E}$ and player $\mathcal{O}$ in parity-3 games in their respective winning sets can be computed in $O(n^{2.5})$ time.*

Remark 1. (DISCUSSION ON GENERAL PARITY GAMES). We now discuss the implication of our result for general parity games (we do not discuss general Streett games where the problem is coNP-complete [17]). The current best known algorithm for parity games with dependence on the number of priorities d is from [37], and the (simplified) running time for $d = o(\sqrt{n})$ is $O(n^{\gamma(d)} \cdot m)$, where $\gamma(d)$ is approximately $d/3$ for large d. More precisely, $\gamma(d) = d/3 + 1/2 - 1/(\lceil 0.5d \rceil \lfloor 0.5d \rfloor)$ for odd d, and $\gamma(d) = d/3 + 1/2 - 1/(3d) - 1/(\lceil 0.5d \rceil \lfloor 0.5d \rfloor)$ for even d. Our algorithm for parity-3 games also extends to parity games as a recursive algorithm as follows: we apply our initialization step and iterated vertex deletions, and to find dominions we replace BÜCHI by our recursive algorithm that handles games that have one less priority and replace BÜCHIDOMINION by a procedure to find dominions with small-progress measure of [27] with our graph decomposition and codomain bounded by $h_{\max}$ (where $h_{\max}$ is chosen to balance the running time of the two dominion find procedures). For the sake of simplicity of presentation we consider the case of constantly many priorities and refer for the analysis of the general case to [37]. Using the notation of [37] with $\beta(d) = \gamma(d)/(\lfloor 0.5d \rfloor + 1)$, we obtain with $h_{\max} = n^{\beta(d)}$ a running time of our algorithm of $O(n^{1+\gamma(d+1)}) = O(n^{2+\gamma(d)-\beta(d)})$ for parity games with d priorities, i.e.,

# priorities	3	4	5	6	7
Schewe [37]	$O(mn)$	$O(mn^{3/2})$	$O(mn^2)$	$O(mn^{7/3})$	$O(mn^{11/4})$
[37] if $m = \Theta(n^2)$	$O(n^3)$	$O(n^{7/2})$	$O(n^4)$	$O(n^{13/3})$	$O(n^{19/4})$
this paper	$O(n^{2.5})$	$O(n^3)$	$O(n^{10/3})$	$O(n^{15/4})$	$O(n^{65/16})$

Table 1: Comparison of running times of [37] and our algorithm for small priorities.

it replaces m of [37] by $n^{2-\beta(d)}$. We present the details of the calculation. We show by induction that our algorithm solves parity games with $d-1$ priorities in $O(n^{1+\gamma(d)})$ time. The base case of $d-1=3$ follows from our algorithm for parity-3 games. The inductive case is as follows: To solve a parity game with d priorities, our algorithm calls the progress measure algorithm (on the graph decomposition) for $d-1$ priorities with $h_{\max} = n^{\beta(d)}$ and recursively calls the algorithm for $d-1$ priorities at most $O(n^{1-\beta(d)})$ times. The total time for the progress measure algorithm is bounded by $O(n^2 n^{\beta(d)\lfloor 0.5d \rfloor})$ and the total time for all calls to the algorithm for $d-1$ priorities is bounded by $O(n^{1-\beta(d)} n^{1+\gamma(d)})$. We obtain the recurrence $\gamma(d+1) = 1 + \gamma(d) - \beta(d)$, which yields $\gamma(d)$ as defined above and a running time of $O(n^{1+\gamma(d+1)})$ for d priorities. In the limit $\beta(d)$ approaches $2/3$. For small d we compare our running times with Schewe's in Table 1. We have presented the details for parity-3 games for the following reasons: (1) All the key ideas and conceptual details are easily demonstrated for the simpler case of parity-3 games and (2) while all previous ideas for general parity games (such as [28, 37]) and for Büchi games (such as [8, 5, 6]) fail to improve the running time for parity-3 games, our approach succeeds to break the long-standing $O(nm)$ barrier for dense graphs.

3 K-Pair Streett Objectives in Graphs

3.1 Preliminaries

Let $G[S]$ denote the subgraph of a graph $G = (V, E)$ induced by the set of vertices $S \subseteq V$. $RevG$ denotes the graph with vertices V and all edges of G reversed. Let $Reach(S, G)$ be the set of vertices in G that *can reach* a vertex in $S \subseteq V$. A strongly connected component (SCC) of a directed graph $G = (V, E)$ is a subgraph $G[S]$ induced by a subset of vertices $S \subseteq V$ such that there is a path in $G[S]$ between every pair of vertices in S. We call an SCC *trivial* if it only contains a single vertex and no edges. All other SCCs are *non-trivial*. The set $Reach(S, G)$ and the maximal SCCs of a graph G can be found in linear time [3, 25, 39].

Algorithm STREETT *and good component detection.* The input is a directed graph $G = (V, E)$ and k Streett pairs (L_j, U_j), $j = 1, \ldots, k$. The size of the input is measured in terms of $m = |E|$, $n = |V|$, k, and $b = \sum_{j=1}^{k} (|L_j| + |U_j|)$. Consider a maximal SCC C; the *good component detection problem* asks to (a) output a non-trivial SCC $G[X] \subseteq C$ such that for all $1 \leq j \leq k$ either no vertex in L_j or at least one vertex in U_j is contained in the SCC (i.e., $L_j \cap X = \emptyset$ or $U_j \cap X \neq \emptyset$), or (b) detect that no such SCC exists. In the former case, there exists an infinite path that visits X infinitely often and satisfies the Streett objective, while in the later case there exists no infinite path that visits vertices in C infinitely often and satisfies the Streett objective. It follows from the results of [1] that the following algorithm, called Algorithm STREETT, suffices for the winning set computation: (1) Compute the maximal SCC decomposition of the graph; (2) for each maximal SCC C for which the good component detection returns an SCC, label the maximal SCC C as *satisfying*; (3) output the set of nodes that can reach a satisfying maximal SCCs as the winning set. Since the first and last step are linear time, the runtime of Algorithm STREETT is dominated by the detection of good components in maximal SCCs. In the following we assume that the input graph is strongly connected and focus on good component detection.

Certificate computation. Given a start vertex x that belongs to the winning set, a certificate is an example of an *accepting run*, i.e., an infinite path from x that satisfies the objective. The output of Algorithm STREETT

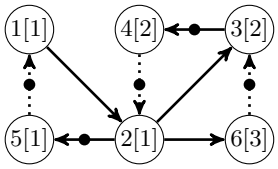

Figure 1: An example for a "jungle" constructed by Tarjan's SCC algorithm for an SCC. Backedges are dotted, spanning tree edges are solid. Backlinks are marked with a dot. The numbers of the vertices represent the order in which the vertices are visited, the numbers in brackets are the lowlinks.

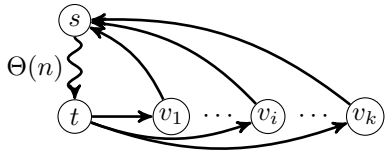

Figure 2: Let the only path between s and t be of length $\Theta(n/2) = \Theta(n)$, not containing any of the vertices v_j for $1 \leq j \leq k$. Let the Streett pairs (L_j, U_j) be given by $L_j = \{s\}$ and $U_j = \{v_j\}$ for $1 \leq j \leq k$. For this example the size of the smallest certificate is $\Theta(nk)$, where k can be of order $\Theta(n)$.

can be used to construct such an accepting run. Given a start vertex x and a good component $G[X]$ reachable from x, we generate the accepting run as follows. A path from x to X can be found in linear time by a depth-first search. Let v be the vertex in X where this path ends. We call v the *root* of the SCC $G[X]$. We show next how to obtain, in $O(m + n \min(n, k))$ time, from the SCC $G[X]$ a cycle starting and ending at the root v such that the resulting certificate is indeed an accepting run. For this it is sufficient that the cycle in $G[X]$ contains for each L_j with $L_j \cap X \neq \emptyset$ a vertex in $U_j \cap X$, i.e., we do not have to include *all* vertices in X.

We can use Tarjan's depth-first search based SCC algorithm [39] to traverse the subgraph $G[X]$ in linear $O(m)$ time, starting from root v. Tarjan's algorithm constructs a graph called *jungle* with $O(|X|)$ edges that for an SCC $G[X]$ consists of a spanning tree and at most one *backedge* per vertex in X. The vertices are assigned pre-order numbers in the order they are traversed. We say an edge of $G[X]$ is a *backedge* if it leads from a vertex with a higher number to a vertex with a lower number. Spanning tree edges always lead from lower numbered vertices to higher numbered vertices. In Tarjan's algorithm a *lowlink* is determined for each vertex u which refers to the lowest numbered vertex w that u can reach by a sequence of tree edges followed by at most one backedge. We additionally store at each vertex $u \neq v$ a *backlink* that is the first edge on the path from u to its lowlink. The backlinks can be determined and stored during the depth-first search without increasing its running time.

With this data structure we can find within $G[X]$ a path from root v to a vertex $u \in X$, $u \neq v$, and back by first searching for u in the spanning tree and then following the backlinks back to v. Since no vertex will appear more than twice on this path, its size and the time to compute it is $O(|X|)$. As it suffices to find such paths for one vertex per nonempty set $U_j \cap X$, we can generate a certificate from $G[X]$ in $O(m + |X| \min(|X|, |\{j \mid U_j \cap X \neq \emptyset\}|))$ time, which can be bounded with $O(m + n \min(n, k))$. This certificate has a size of $O(n \min(n, k))$. As shown in Figure 2, the smallest existing certificate can be as large as $\Theta(n \min(n, k))$.

Next, we introduce the different concepts used in the algorithm for good component detection. First we describe the hierarchical graph decomposition technique for this setting. This decomposition will be crucial for the runtime analysis.

Graph decomposition. In our algorithm we decompose a graph G in the following way. Let $G_i = (V, E_i)$ be a subgraph of G with $E_i = \{(u, v) \mid Outdeg(u) \leq 2^i\}$, i.e., the edges of G_i are the outedges of the vertices with outdegree at most 2^i. Note that for $i = \log n$ we have that $G_i = G$. We say vertices in G with $Outdeg(v) > 2^i$ are *colored blue* in G_i and denote the set of blue vertices in G_i by Bl_i. All other

vertices are *white*. Note that all vertices in $G = G_{\log n}$ are white and that all vertices in Bl_i have outdegree zero in G_i.

Top and bottom strongly connected components. The algorithm will repeatedly find a top or a bottom SCC in the remaining graph G. A bottom SCC $G[S]$ in a directed graph G is an SCC with no edges from vertices in S to vertices in $V \setminus S$, i.e., no *outgoing* edges. A top SCC is a bottom SCC of $RevG$, i.e., an SCC without *incoming* edges. Top and bottom SCCs are by definition maximal SCCs. Note that every graph has at least one bottom and at least one top SCC. If they are not the same, then they are disjoint and thus one of them contains at most half of the vertices of G.

Lock-step search. The lock-step search of Even and Shiloach [20] was already applied (in a different way) to the detection of good components in [24]. Lock-step search simulates parallel execution by alternatingly making one step in each parallel instance. The execution finishes as soon as the first instance terminates. In our algorithm we use this technique to search in parallel for the smallest top or bottom SCC, where the search for a top SCC is performed by searching for a bottom SCC in $RevG$.

Bad vertices. In contrast to *good components* we also define *bad vertices*. The basic idea behind the algorithms for good component detection, described for example in [24], is to repeatedly delete *bad* vertices until either a good component is found or it can be concluded that no such component exists. A vertex is *bad* if for some index j with $1 \leq j \leq k$ the vertex is in L_j but it is not strongly connected to any vertex in U_j. All other vertices are *good*. Note that good vertices can become bad if some vertex deletion disconnects an SCC or a vertex in a set U_j is deleted. A good component is then a non-trivial SCC that only contains good vertices.

Data structure. The algorithm maintains for the current graph $G = (V, E)$ (some vertices of the input graph might have been deleted) a decomposition into vertex sets $S \subseteq V$ such that every SCC of G is completely contained in $G[S]$ for one of the sets S. For all the sets S a data structure $D(S)$ is saved in a list Q. The data structure $D(S)$ supports the following operations: (1) $Construct(S)$ initializes the data structure for the set S, (2) $Remove(S, D(S), B)$ removes a set $B \subseteq V$ from S and updates the data structure of S accordingly, and (3) $Bad(D(S))$ returns the set $\{v \in S \mid \exists j \text{ with } v \in L_j \text{ and } U_j \cap S = \emptyset\}$. In [24] an implementation of this data structure was given that achieves the following running times. For a set of vertices $S \subseteq V$ let $bits(S)$ be defined as $\sum_{j=1}^{k} (|S \cap L_j| + |S \cap U_j|)$.

Lemma 3.1 (Lemma 2.1 in [24])**.** *After a one-time preprocessing of time $O(k)$, the data structure $D(S)$ can be implemented in time $O(bits(S) + |S|)$ for $Construct(S)$, time $O(bits(B) + |B|)$ for $Remove(S, D(S), B)$, and constant running time for $Bad(D(S))$.*

3.2 Algorithm

By abuse of notation we denote by G the *current* graph maintained by the algorithm where some edges and vertices might have been deleted and use *input graph* to denote the unmodified, strongly connected graph for which a good component is searched. Our algorithm for good component detection is given in Algorithm GOODCOMPONENT. It maintains in a list Q a partition of the vertices in G into sets such that every SCC of G is contained in the subgraph induced by one of the vertex sets. The list is initialized with the set of all vertices in the strongly connected input graph. We will show that if a good component exists, it must be fully contained in one of the vertex sets in the partition. The algorithm repeatedly removes a set S from Q and identifies and deletes bad vertices from $G[S]$. If no edge is contained in $G[S]$, the set S is removed as it can only induce trivial components. Otherwise the subgraph $G[S]$ is either determined to be strongly connected and output as a good component or a "small" maximal SCC in $G[S]$ is identified. To find a small maximal SCC the algorithm searches in lock-step in $G[S]$ and in $RevG[S]$ for a bottom SCC and stops as soon as one of the searches stops. (A bottom SCC in $RevG[S]$ is a top SCC in $G[S]$.) We only describe the search in $G[S]$ here, the search in $RevG[S]$ is analogous. The algorithm uses the hierarchical graph decomposition in $G[S]$. The subgraph $G_i[S]$ for any i contains only the outedges of vertices with an outdegree of at most 2^i. The search for a bottom SCC is started at $i = 1$, then i is increased one by one if necessary, up to at most $\log n$. If for some i we can identify a bottom SCC that does not contain any

blue vertex (i.e. a vertex for which some edges are missing in G_i), then the found SCC in $G_i[S]$ must also be a bottom SCC in $G[S]$. If multiple bottom SCCs (without blue vertices) are found in $G_i[S]$, we only consider the smallest one. We then put the newly detected SCC and the "rest" of S back into Q.

ALGORITHM GOODCOMPONENT: Detection of good components for the winning set computation in graphs with k-pair Streett objectives

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

```
1  add Construct(V) to Q
2  while Q ≠ ∅ do
3      pull D(S) from Q
4      while Bad(D(S)) ≠ ∅ do D(S) ← Remove(S, D(S), Bad(D(S)))
5      if G[S] contains at least one edge then
6          in lock-step for H ∈ {G, RevG}
7              for i ← 1 to log n do
8                  construct H_i[S]; Bl_i ← {v ∈ S | Outdeg(v) > 2^i}      /* Outdeg in H */
9                  Z ← S \ Reach(Bl_i, H_i[S])                  /* Z cannot reach Bl_i */
10                 if Z ≠ ∅ then
11                     H[X] ← SmallestBottomSCC(H_i[Z])
12                     if X = S then return H[S]
13                     if |X| ≤ |S|/2 then
14                         break
15         add Remove(S, D(S), X) and Construct(X) to Q
16 return no good component exists
```

The idea of the running time analysis is as follows. We can show that a bottom SCC of $G[S]$ identified in iteration i of the for-loop must contain $\Omega(2^i)$ vertices. In time $O(n2^i)$ a standard SCC algorithm can compute all SCCs of $G_i[S]$ and thus also the smallest bottom SCC. The time needed for the search in all graphs $G_{i'}[S]$ up to i can be bounded with an additional factor of two. Thus the work for the search is $O(n)$ per vertex in the identified SCC.

Given that the subgraph $G[S]$ was split into at least one top and one bottom SCC, the smallest top or bottom SCC contains at most half of the vertices of the subgraph. By searching for a smallest bottom SCC (without blue vertices) in $G_i[S]$ and $RevG_i[S]$ we find one top or bottom SCC with at most half of the vertices of the subgraph. We charge the work for finding such an SCC to the vertices in this SCC. This guarantees that each vertex will be charged at most $O(\log n)$ times over the whole running time of the algorithm. Thus we can bound the total running time for computing SCCs by $O(n^2 \log n)$.

We additionally have to take the time for the maintenance of the data structures into account. Here we use the properties of the data structure $D(S)$ described in Lemma 3.1 to obtain a running time of $O((n + b) \log n)$ for the maintenance of the data structures and the identification of bad vertices over the whole algorithm. Combined these ideas lead to a total running time of $O((n^2 + b) \log n)$.

Lemma 3.2. *Whenever in Algorithm* GOODCOMPONENT *the for-loop stops for $H \in \{G, RevG\}$ and some $i = i^*$ with a nonempty vertex set $Z = S \setminus Reach(Bl_{i^*}, H_{i^*}[S])$ and the smallest bottom* SCC *$H[X]$ in $H_{i^*}[Z]$ returned by $SmallestBottomSCC(H_{i^*}[Z])$ with $|X| \leq |S|/2$, then $H[X]$ contains at least 2^{i^*-1} vertices.*

Proof. As Bl_{i^*-1} is the set of vertices in $H_{i^*-1}[S]$ with outdegree larger than 2^{i^*-1}, any bottom SCC $H[Y]$ that contains a vertex of Bl_{i^*-1}, has $|Y| \geq 2^{i^*-1}$. Hence it suffices to show that $X \cap Bl_{i^*-1} \neq \emptyset$. Assume by contradiction that $X \cap Bl_{i^*-1} = \emptyset$. Since $H[X]$ is a bottom SCC, no vertex in X can reach any vertex in Bl_{i^*-1}, i.e., $X \subseteq S \setminus Reach(Bl_{i^*-1}, H_{i^*}[S])$. As all edges in $H_{i^*-1}[S]$ are contained in $H_{i^*}[S]$, this implies $X \subseteq S \setminus Reach(Bl_{i^*-1}, H_{i^*-1}[S])$. Since $SmallestBottomSCC$ finds the smallest

bottom SCC in graph H_i for each i, the for-loop would thus have terminated in an iteration $i \leq i^* - 1$. Contradiction. ∎

Lemma 3.3 (Runtime)**.** *Algorithm* GOODCOMPONENT *can be implemented in time* $O((n^2 + b) \log n)$.

Proof. The preprocessing and initialization of the data structure and the removal of bad vertices in the whole algorithm take time $O(m + k + b)$ using Lemma 3.1. Additionally we maintain at each vertex a list of its incoming and a list of its outgoing edges including pointers to the lists of its neighbors, which we use to update the lists of its neighbors. Since each vertex is deleted at most once, this data structure can be constructed and maintained in total time $O(n^2)$.

Consider the while loop where a set S is removed from Q. The lock-step search for $G[S]$ and $RevG[S]$ only increases the running time by a factor of two, thus we restrict the analysis of the running time to $G[S]$. The construction of $G_i[S]$, Z, and $G[X]$ can all be done in time $O(n2^i)$ for each i, i.e., in total time $O(n2^{i^*})$ up to level i^*. If $X = S$, then the algorithm terminates and the time for processing S can be bounded by $O(n2^{\log n}) = O(n^2)$. If the processing of S ends when some bottom SCC $G[X] \subset G[S]$ is found, let i^* be the value of i when $G[X]$ is detected and inserted into Q. By Lemma 3.2 the set X contains at least 2^{i^*-1} vertices. We charge $O(n)$ to each vertex in X. Since $|X| \leq |S|/2$, a vertex v is only charged when the size of the set in Q containing v is halved, which can happen at most $\lceil \log n \rceil$ times. Thus the total running time for processing all sets S, except for the work in $Remove$ and $Construct$, can be bounded by $O(n^2 \log n)$. The $Remove$ and $Construct$ are called once per found bottom SCC $G[X]$ with $X \neq S$ and take by Lemma 3.1 time $O(|X| + bits(X))$ time. Hence, by charging $O(1)$ to the vertices in X and, respectively, to $bits(X)$, the total running time for this part can be bounded by $O((n + b) \log n)$ as each vertex and bit will only be charged $O(\log n)$ times. Combining all parts yields the claimed running time bound of $O((n^2 + b) \log n)$. ∎

To prove the correctness of Algorithm GOODCOMPONENT we first show that all candidates for good components are in Q before each iteration of the algorithm.

Lemma 3.4. *Before each iteration of the outer while-loop every good component of the input graph is contained in one of the subgraphs* $G[S]$ *for which the data structure* $D(S)$ *is maintained in the list* Q.

Proof. We will show that the algorithm never removes edges or vertices that belong to a good component, which together with a correct initialization of the list Q will imply the lemma. At the beginning of the algorithm one data structure for the whole strongly connected input graph is added to Q. Thus every good component is contained in this data structure in Q after the initialization. At the beginning of each iteration of the outer while-loop the data structure of one of the subgraphs $G[S]$ is pulled from the list Q. In Line 4 we remove vertices from the subgraph that are in some set L_j but not strongly connected to any vertex in U_j, i.e., bad vertices. In Line 5 we remove trivial SCCs. Observe that a good component is non-trivial and does not contain any bad vertices. Thus the removal of bad vertices and trivial SCCs does not remove any vertices of a good component, i.e., after the removal of these vertices the updated subgraph $G[S]$ still contains the good components it contained before. If no good component is identified in this iteration, i.e., the algorithm does not terminate, we find a bottom or top SCC $G[X]$, which is by definition a maximal SCC. Since a good component has to be strongly connected, every good component in $G[S]$ must either be a subgraph of the newly identified SCC $G[X]$ or does not contain *any* vertex in X. Thus the removed edges between $G[X]$ and the remaining subgraph cannot belong to a good component. Finally, we add the data structures for $G[X]$ as well as for $G[S \setminus X]$ to Q. Thus no vertex or edge of a good component was removed and every good component continues to be completely contained in a subgraph in Q. ∎

As all candidates for good components are maintained in the list Q, it remains to show that the algorithm makes progress in each iteration and correctly outputs a good component if and only if one exists.

Lemma 3.5 (Correctness)**.** *Algorithm* GOODCOMPONENT *outputs a good component if one exists, otherwise the algorithm reports that no such component exists.*

Proof. First we show that whenever Algorithm GOODCOMPONENT outputs a subgraph $G[S]$, then $G[S]$ is a good component. Line 5 ensures only non-trivial SCCs are considered. After the removal of bad vertices from S in Line 4 we know that for all $1 \leq j \leq k$ and all vertices in $S \cap L_j$ there exists a vertex in $S \cap U_j$. Thus if $G[S]$ is strongly connected, then $G[S]$ is a good SCC. The algorithm computes a maximal SCC in $G[S]$. If $G[S]$ is equal to the found maximal SCC, i.e., $G[S]$ remains strongly connected, then $G[S]$ is a good component and is output in Line 12. This is the only case when Algorithm GOODCOMPONENT outputs a subgraph $G[S]$; thus if the algorithm outputs a component, it is a good component.

Algorithm GOODCOMPONENT terminates if a good component is identified or Q is empty. Lemma 3.4 shows that before every iteration of the outer while-loop *every* good component is contained in one of the subgraphs $G[S]$ in Q. That is, if a good component exists in G, the algorithm will not terminate until a good component is identified. Whenever the algorithm does not terminate in an iteration of the outer while-loop, either (a) a trivial SCC is removed from Q (Line 5) or (b) one of the subgraphs from Q is split into two smaller subgraphs (Line 15). Each case can happen at most n times. This implies that the algorithm terminates after a finite number of steps if no good component exists. Next we show that if there exists a good component in G, then the algorithm will output a good component. Let Y be a *maximal* good component in G and let S_Y be the vertex set maintained in Q that currently contains the vertices in Y. By the arguments above after a finite number of steps either (1) another good component is detected or (2) $D(S_Y)$ is pulled from Q. By Lemma 3.4 Y is never split by the algorithm thus after Case (2) happened at most n times, one of the following two cases occurs: either (2a) $D(S_Y)$ is pulled from Q with $G[S_Y] \supset Y$ and after the removal of bad vertices from S_Y, $G[S_Y]$ without the bad vertices is equal to Y or (2b) $G[S_Y] = Y$ is pulled from Q. In both cases the good component Y is output and the algorithm terminates: Since Y is non-trivial, the condition in Line 5 is satisfied. The algorithm searches for a top or bottom SCC in Y. Since Y is strongly connected, the only top or bottom SCC in Y is Y itself. Hence the algorithm outputs Y in Line 12. ∎

Recall Algorithm STREETT that calls Algorithm GOODCOMPONENT for each maximal SCC in the input graph and then computes reachability to the union of the identified good components. Lemmata 3.3 and 3.5 yield the following result.

Theorem 3.6. *Algorithm* STREETT *correctly computes the winning set in graphs with k-pair Streett objectives in $O((n^2 + b) \log n)$ time. Given a vertex x in the winning set, a certificate for x can be output in time $O(m + n \min(n, k))$.*

Acknowledgements

K. C. is supported by the Austrian Science Fund (FWF): P23499-N23 and S11407-N23 (RiSE), an ERC Start Grant (279307: Graph Games), and a Microsoft Faculty Fellows Award. M. H. is supported by the Austrian Science Fund (FWF): P23499-N23 and the Vienna Science and Technology Fund (WWTF) grant ICT10-002. V. L. is supported by the Vienna Science and Technology Fund (WWTF) grant ICT10-002. The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement no. 340506.

References

[1] R. Alur and T. A. Henzinger. Computer-aided verification, 2004. Unpublished, available at `http://www.cis.upenn.edu/cis673/`.

[2] R. Alur, T.A. Henzinger, and O. Kupferman. Alternating-time temporal logic. *Journal of the ACM*, 49:672–713, 2002.

[3] C. Beeri. On the membership problem for functional and multivalued dependencies in relational databases. *ACM Transactions on Database Systems*, pages 241–259, 1980.

[4] R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive(1) designs. *J. Comput. Syst. Sci.*, 78(3):911–938, 2012.

[5] K. Chatterjee and M. Henzinger. An $O(n^2)$ Time Algorithm for Alternating Büchi Games. In *SODA*, pages 1386–1399, 2012.

[6] K. Chatterjee and M. Henzinger. Efficient and Dynamic Algorithms for Alternating Büchi Games and Maximal End-component Decomposition. *Journal of the ACM*, 61(3):15, 2014.

[7] K. Chatterjee, T. A. Henzinger, and V. S. Prabhu. Timed parity games: Complexity and robustness. *Logical Methods in Computer Science*, 7(4), 2011.

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

[9] K. Chatterjee and V. S. Prabhu. Synthesis of memory-efficient, clock-memory free, and non-zeno safety controllers for timed systems. *Inf. Comput.*, 228:83–119, 2013.

[10] A. Church. Logic, arithmetic, and automata. In *Proceedings of the International Congress of Mathematicians*, pages 23–35. Institut Mittag-Leffler, 1962.

[11] L. de Alfaro and M. Faella. An accelerated algorithm for 3-color parity games with an application to timed games. In *CAV*, pages 108–120, 2007.

[12] L. de Alfaro, M. Faella, T. A. Henzinger, R. Majumdar, and M. Stoelinga. The element of surprise in timed games. In *CONCUR*, pages 142–156, 2003.

[13] L. de Alfaro and T.A. Henzinger. Interface automata. In *FSE'01*, pages 109–120. ACM Press, 2001.

[14] D.L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-independent Circuits*. The MIT Press, 1989.

[15] A. Duret-Lutz, D. Poitrenaud, and J.-M. Couvreur. On-the-fly Emptiness Check of Transition-Based Streett Automata. In *ATVA*, pages 213–227, 2009.

[16] R. Ehlers. Short Witnesses and Accepting Lassos in ω-Automata. In *LATA*, pages 261–272. 2010.

[17] E.A. Emerson and C. Jutla. The complexity of tree automata and logics of programs. In *FOCS'88*, pages 328–337. IEEE, 1988.

[18] E.A. Emerson and C.S. Jutla. Tree automata, mu-calculus and determinacy. In *FOCS*, pages 368–377, 1991.

[19] E.A. Emerson and C.-L. Lei. Modalities for Model Checking: Branching Time Logic Strikes Back. *Science of Computer Programming*, 8(3):275–306, 1987.

[20] S. Even and Y. Shiloach. An On-Line Edge-Deletion Problem. *Journal of the ACM*, 28(1):1–4, 1981.

[21] N. Francez. *Fairness*. Springer, New York, 1986.

[22] Y. Godhal, K. Chatterjee, and T. A. Henzinger. Synthesis of AMBA AHB from formal specification: a case study. *STTT*, 15(5-6):585–601, 2013.

[23] M. Henzinger, V. King, and T. Warnow. Constructing a Tree from Homeomorphic Subtrees, with Applications to Computational Evolutionary Biology. *Algorithmica*, 24:1–13, 1999.

[24] M. Henzinger and J.A. Telle. Faster Algorithms for the Nonemptiness of Streett Automata and for Communication Protocol Pruning. In *SWAT*, pages 16–27, 1996.

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

[26] M. Jurdziński. Deciding the winner in parity games is in UP ∩ co-UP. *Information Processing Letters*, 68(3):119–124, 1998.

[27] M. Jurdziński. Small Progress Measures for Solving Parity Games. In *STACS*, pages 290–301, 2000.

[28] M. Jurdziński, M. Paterson, and U. Zwick. A Deterministic Subexponential Algorithm for Solving Parity Games. *SIAM Journal on Computing*, 38(4):1519–1532, 2008.

[29] T. Latvala and K. Heljanko. Coping With Strong Fairness. *Fundamenta Informaticae*, 43(1-4):175–193, 2000.

[30] O. Lichtenstein and A. Pnueli. Checking That Finite State Concurrent Programs Satisfy Their Linear Specification. In *POPL*, pages 97–107, 1985.

[31] D.A. Martin. Borel determinacy. *Annals of Mathematics*, 102(2):363–371, 1975.

[32] R. McNaughton. Infinite games played on finite graphs. *Annals of Pure and Applied Logic*, 65(2):149–184, 1993.

[33] N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive(1) designs. In *VMCAI*, LNCS 3855, Springer, pages 364–380, 2006.

[34] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *POPL'89*, pages 179–190. ACM Press, 1989.

[35] P.J. Ramadge and W.M. Wonham. Supervisory control of a class of discrete-event processes. *SIAM Journal of Control and Optimization*, 25(1):206–230, 1987.

[36] S. Safra. On the complexity of ω-automata. In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science*, pages 319–327. IEEE Computer Society Press, 1988.

[37] S. Schewe. Solving Parity Games in Big Steps. In *FSTTCS*, pages 449–460, 2007.

[38] S. Schewe. *Synthesis of Distributed Systems*. PhD thesis, Universität des Saarlandes, 2008.

[39] R. E. Tarjan. Depth First Search and Linear Graph Algorithms. *SIAM Journal of Computing*, 1(2):146–160, 1972.

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

[41] J. Vöge and M. Jurdziński. A discrete strategy improvement algorithm for solving parity games. In *CAV'00*, pages 202–215. LNCS 1855, Springer, 2000.

[42] W. Zielonka. Infinite games on finitely coloured graphs with applications to automata on infinite trees. *Theoretical Computer Science*, 200(1–2):135–183, 1998.