Logical Methods in Computer Science
Vol. 13(3:26)2017, pp. 1–27
www.lmcs-online.org

Submitted Apr. 21, 2016
Published Sep. 26, 2017

IMPROVED ALGORITHMS FOR PARITY AND STREETT OBJECTIVES *

KRISHNENDU CHATTERJEE [a], MONIKA HENZINGER [b], AND VERONIKA LOITZENBAUER [c]

[a] IST Austria, Klosterneuburg, Austria
e-mail address: krishnendu.chatterjee@ist.ac.at

[b] University of Vienna, Faculty of Computer Science, Vienna, Austria
e-mail address: monika.henzinger@univie.ac.at

[c] Bar-Ilan University, Ramat Gan, Israel
e-mail address: veronika@datalab.cs.biu.ac.il

Abstract. The computation of the winning set for parity objectives and for Streett objectives in graphs as well as in game graphs are central problems in computer-aided verification, with application to the verification of closed systems with strong fairness conditions, the verification of open systems, checking interface compatibility, well-formedness of specifications, and the synthesis of reactive systems. We show how to compute the winning set on n vertices for (1) parity-3 (aka one-pair Streett) objectives in game graphs in time $O(n^{5/2})$ and for (2) k-pair Streett objectives in graphs in time $O(n^2 + nk \log n)$. For both problems this gives faster algorithms for dense graphs and represents the first improvement in asymptotic running time in 15 years.

1. Introduction

In the formal verification and synthesis of systems *graphs* and *game graphs* are fundamental models of systems, where vertices correspond to states of the systems and edges correspond to transitions between states. *ω-regular objectives* are a canonical way to specify desired and undesired behaviors of systems, and the algorithmic questions are to determine whether a model satisfies its specification or to generate a strategy to satisfy the specification. In this work we study *Streett* and *parity* objectives that can express all ω-regular objectives

Key words and phrases: Computer-aided verification; Synthesis; Graph games; Parity games; Streett automata; Graph algorithms.

* A preliminary version of this paper appeared in [14].

[a,b,c] The authors are partially supported by the Vienna Science and Technology Fund (WWTF) grant ICT15-003 and the Austrian Science Fund (FWF): P23499-N23.

[a] Partially supported by the Austrian Science Fund (FWF): S11407-N23 (RiSE/SHiNE), an ERC Start Grant (279307: Graph Games), and a Microsoft Faculty Fellows Award.

[b,c] 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 and the Vienna Science and Technology Fund (WWTF) grant ICT10-002.

[c] Work done while at University of Vienna.

LOGICAL METHODS IN COMPUTER SCIENCE DOI:10.23638/LMCS-13(3:26)2017
© K. Chatterjee, M. Henzinger, and V. Loitzenbauer
Creative Commons

and the algorithmic questions we consider are therefore core questions in formal verification and synthesis. We first define the problem, next discuss its significance and previous work, and then present our contributions.

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 $m = |E|$. On the graph two players play the following *alternating game* 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 the 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.* The winning sets partition the game graph, i.e., 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 of U *at least once*. The next interesting objective is the *Büchi* objective that requires an infinite path to visit some vertex of 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 of L_1 is visited infinitely often, then some vertex of 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). A different generalization of one-pair Streett objectives are *parity objectives.* For a parity objective the input additionally contains a priority function that assigns each vertex a natural number called priority. The parity objective is satisfied if the highest priority visited infinitely often is even. Parity objectives with at most 2 different priorities are equivalent to Büchi objectives and parity objectives with at most 3 different priorities are equivalent to one-pair Streett objectives.

We study (1) game graphs with parity-3 (aka one-pair Streett) objectives and their generalization to parity 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 the verification and synthesis of systems such as the synthesis of reactive systems [18, 48, 49], the verification of open systems [2], and checking interface compatibility [22] and the well-formedness of specifications [23], and many others. General and one-pair Streett objectives are central in verification as most commonly used specifications can be expressed as Streett automata [50, 55]. Parity objectives are also canonical to express

properties in verification, since every Streett automaton can be converted to a parity automaton [51]. Moreover, parity objectives are particularly important as solving parity games is equivalent to μ-calculus model checking [27]. Dense game graphs can emerge from a synchronous product of several components (where each component makes transitions at each step) [42, 11].

Game graphs with parity-3 objectives arise in many applications in verification. We sketch a few of them. (A) Timed automaton games are a model for real-time systems. The analysis of such games with reachability and safety objectives (which are the dual of reachability objectives) reduces to game graphs with parity-3 objectives [21, 20, 15, 17]. (B) The synthesis of Generalized Reactivity(1) (aka GR(1)) specifications exactly require the solution of game graphs with parity-3 objectives [5]; GR(1) specifications are standard for hardware synthesis [47] and even used in the synthesis of industrial protocols [33, 6][1]. (C) Finally, the problem of fair simulation [36] between two systems also reduces to game graphs with parity-3 objectives [29, 8].

General Streett objectives in standard graphs arise, for example, in the verification of closed systems with strong fairness conditions [43, 24, 31]. 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 [25, 43, 24], such as in the counterexample-guided abstraction refinement approach (CEGAR) [19].

Note that *standard graphs* are relevant for the verification of *closed* systems or *open* systems with demonic non-determinism (e.g., all inputs are from the environment that are not controllable); while *game graphs* are relevant for the synthesis and verification of *open* systems with both angelic and demonic non-determinism (e.g., certain inputs are controllable, and certain inputs are not controllable).

Previous results. We summarize the previous results for game graphs and graphs with Streett and Parity 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, 37]. For *Büchi* objectives, the current best known algorithm requires $O(n^2)$ time [13]. For *general Streett* objectives, the problem is coNP-complete [26], and for *one-pair Streett* objectives the current best known algorithm requires $O(mn)$ time [39]. One-pair Streett objectives also corresponds to the well-known parity games problem with three priorities. Despite the importance of game graphs with parity-3 objectives in numerous applications and several algorithmic ideas to improve the running time for general parity games [56, 41, 53] or Büchi games [16, 13], there has been no algorithmic improvement since 2000 [39] for parity-3 games. The parity games problem in general is in UP $\cap$ coUP [38]; it is one of the rare and intriguing combinatorial problems that lie in NP $\cap$ coNP but are not known to be in PTIME. Parity games can be

[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 parity-3 objective.

Table 1: Comparison of Running Times for Few Priorities.

	# priorities				
	3	4	5	6	7
Big-Step [53]	$O(mn)$	$O(mn^{3/2})$	$O(mn^2)$	$O(mn^{7/3})$	$O(mn^{11/4})$
Big-Step [53] with $m = \Theta(n^2)$	$O(n^3)$	$O(n^{7/2})$	$O(n^4)$	$O(n^{13/3})$	$O(n^{19/4})$
Our algorithm	$O(n^{5/2})$	$O(n^3)$	$O(n^{10/3})$	$O(n^{15/4})$	$O(n^{65/16})$

solved by a randomized algorithm in time $n^{O(\sqrt{n/\log n})}$ [4] and by deterministic algorithms in time $n^{O(\sqrt{n})}$ [41] and for $c \geq 3$ priorities in time $O(m \cdot (\kappa n/c^2)^{\gamma(c)})$ [53] for a small constant κ and $c/3 \leq \gamma(c) \leq c/3 + 1/2$. Subsequent to our work, a quasi-polynomial time algorithm for parity games was achieved in a breakthrough result by Calude et. al [7, 32]. In follow-up work different quasi-polynomial time algorithms as well as $O(mn\log(n)^{c-1})$ time algorithms for constant c were shown [40, 30].

Graphs. In graphs the winning set for parity objectives with c priorities can be computed in $O(m \log c)$ time [12]. 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 vertex set reachable from x that induces a strongly connected subgraph 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 consists of a path to S and a (not necessarily simple) cycle between the vertices of S [25]. The basic algorithm [28, 44] for the winning set problem has 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 [43] additionally compute a certificate of size at most $n\min(n, 2k)$. Duret-Lutz et al. [24] 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 [35] 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 adds an additive term of $O(n\min(n, k))$ to the running time bound.

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

Game graphs. We show that the winning set computation for game graphs with parity-3 objectives can be achieved in $O(n^{5/2})$ time. Our algorithm is faster for $m \in \Omega(n^{1.5})$, and breaks the long-standing $O(mn)$ barrier for dense graphs. Our algorithm for parity-3 games also extends to general parity games and improves the running time for dense graphs when the number of priorities is sub-polynomial in n. Let, as in [53], $\gamma(c) = c/3 + 1/2 - 4/(c^2 - 1)$ for odd c and $\gamma(c) = c/3 + 1/2 - 1/(3c) - 4/c^2$ for even c, and let $\beta(c) = \gamma(c)/(\lfloor c/2 \rfloor + 1)$. We obtain that the running time of our algorithm is $O(n^{1+\gamma(c+1)}) = O(n^{2+\gamma(c)-\beta(c)})$ for parity games with c priorities, i.e., for a constant number of priorities we replace m of [53] by $n^{2-\beta(c)}$. Since the value of $\beta(c)$ quickly approaches $2/3$ with increasing c, we have that $n^{2-\beta(c)}$ approaches $n^{4/3}$. For small c we compare our running times with the Big-Step algorithm of [53] in Table 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 $m \in$

$\Omega(\max(n^{4/3}\log^{-1/3} n, b^{2/3}\log^{1/3} n))$ and $k \in \Omega(n^2/m)$. We additionally provide an algorithm that, given the winning set, 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 smallest certificate has size $\Theta(n\min(n, k))$, showing that no algorithm can compute and *output* a certificate faster. In contrast to [35], 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. [34] to handle *edge deletions* in *undirected* graphs. In [13] 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 parity-3 objectives repeatedly solves Büchi games such that the union of the winning sets of player 2 in the Büchi games is exactly the winning set for the parity-3 objective. Schewe [53] showed with his Big-Step algorithm that the small progress measure algorithm for parity games by Jurdziński [39] can be used to compute small subsets of the winning set of player 2, called *dominions*, and thereby improved the running time for general parity games. However, the Big-Step algorithm does not improve the running time for parity-3 games. With Schewe's approach dominions with at most h vertices 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 [13] 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. Our approach extends to general parity games.

Graphs. In prior work that used the hierarchical graph decomposition technique the running time analysis relied 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 a strongly connected subgraph is identified and some but not all of its vertices might be removed. As a consequence a vertex might belong to an identified strongly connected subgraph multiple times. We show how to overcome this difficulty by identifying a strongly connected subgraph with at most half of the vertices whenever a vertex set seizes to be strongly connected. We identify these strongly connected subgraphs by running Tarjan's SCC algorithm [54] on the graph and its *reverse graph*, thereby finding the smallest *top* (i.e. with no incoming edges) or *bottom* (i.e. with no outgoing edges) SCC contained in the formerly strongly connected subgraph. The algorithm takes $O(n)$ time per vertex in the identified set. This will allow us to bound the total running time for this part of the algorithm with $O(n^2)$.

In Section 2 we present our algorithm for game graphs with parity objectives with c priorities, where the special case of $c = 3$ corresponds to one-pair Streett objectives. In Section 3 we present the algorithm for general Streett objectives in graphs.

2. Parity Objectives in Game Graphs

2.1. **Preliminaries.**

Notation. Let for all $c \in \mathbb{N}$ denote the set $\{0, 1, \ldots, c-1\}$ by $[c]$.

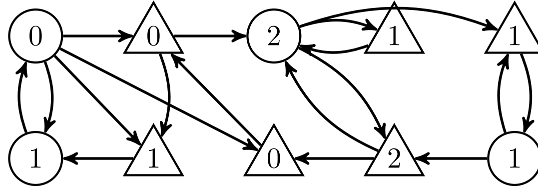

Figure 1: An example of a parity game $\mathcal{P} = (\mathcal{G}, \alpha)$ with three priorities. Circles denote $\mathcal{E}$-vertices, triangles denote $\mathcal{O}$-vertices. The values in the vertices are the priorities.

Parity games. A *parity game* $\mathcal{P} = (\mathcal{G}, \alpha)$ with $c \leq n$ priorities consists of a *game graph* $\mathcal{G} = (G, (V_\mathcal{E}, V_\mathcal{O}))$ with $G = (V, E)$ and a *priority function* $\alpha : V \to [c]$ that assigns an integer from the set $[c]$ to each vertex. The sets $V_\mathcal{E} \subseteq V$ and $V_\mathcal{O} \subseteq V$ form a partition of V. We denote the two players by $\mathcal{E}$ (for even) and $\mathcal{O}$ (for odd). We say that the vertices in $V_\mathcal{E}$ are $\mathcal{E}$-vertices and the vertices in $V_\mathcal{O}$ are $\mathcal{O}$-vertices. Player $\mathcal{E}$ (resp. player $\mathcal{O}$) wins a play if the *highest* priority occurring infinitely often in the play is even (resp. odd). We use z to denote one of the players $\{\mathcal{E}, \mathcal{O}\}$ and $\overline{z}$ to denote her opponent. *Parity-3 games* are parity games with $\alpha : V \to \{0, 1, 2\}$ and *Büchi games* have $\alpha : V \to \{0, 1\}$, where the vertices in the set $B = \{v \mid \alpha(v) = 1\}$ are called *Büchi vertices.*

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 2, let the vertices in $L_1 \setminus U_1$ have priority 1, and let the remaining vertices have priority 0. Then player 1 wins the game with the one-pair Streett objective if and only if player $\mathcal{E}$ wins the parity-3 game. As our algorithm for parity-3 games extends to general parity games, we use the notion of parity games (i.e., player $\mathcal{E}$ and player $\mathcal{O}$ instead of player 1 and player 2).

Plays. For technical convenience we consider that every vertex of the game graph $\mathcal{G}$ 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 of V_z, player z 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 $(v_\ell, v_{\ell+1}) \in E$ for all $\ell \geq 0$.

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

Strategies. A *strategy* of a player $z \in \{\mathcal{E}, \mathcal{O}\}$ is a function that, given a finite prefix of a play ending at $v \in V_z$, selects a vertex from $\mathit{Out}(v)$ 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 z is a function $\sigma : V_z \to V$ such that for all $v \in V_z$ we have $\sigma(v) \in \mathit{Out}(v)$. It is well-known that for parity games it is sufficient to consider memoryless strategies (see Theorem 2.1 below). Therefore we only consider memoryless strategies from now on. A start vertex v, a strategy σ for $\mathcal{E}$, and a strategy π for $\mathcal{O}$ define a unique play $\omega(v, \sigma, \pi) = \langle v_0, v_1, v_2, \ldots \rangle$ 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 σ is winning for player z at start vertex v if the resulting play is winning for player z irrespective of the strategy of player $\overline{z}$. A vertex v belongs to the *winning set* W_z of player z if player z 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 $\mathcal{G}$ or specific parity game $\mathcal{P}$, we use $W_z(\mathcal{G})$ and $W_z(\mathcal{P})$ to refer to the winning sets. The algorithmic question for parity games is to compute the sets $W_\mathcal{E}$ and $W_\mathcal{O}$.

Theorem 2.1 ([27, 46])**.** For every parity game the vertices V can be partitioned into the winning set $W_\mathcal{E}$ of player $\mathcal{E}$ and the winning set $W_\mathcal{O}$ of player $\mathcal{O}$. Each player has a memoryless strategy that is winning for her from all vertices in her winning set.

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

Traps. A set $U \subseteq V$ is a *z-trap* if for all z-vertices u in U we have $Out(u) \subseteq U$ and for all $\overline{z}$-vertices v in U there exists a vertex $w \in Out(v) \cap U$ [57]. Note that player $\overline{z}$ can ensure that a play that currently ends in a z-trap never leaves the z-trap against any strategy of player z by choosing an edge (v, w) with $w \in Out(v) \cap U$ whenever the current vertex v is in $U \cap V_{\overline{z}}$. Given a game graph $\mathcal{G}$ and a z-trap U, we denote by $\mathcal{G}[U]$ the game graph induced by the set of vertices U. Note that given that in $\mathcal{G}$ each vertex has at least one outgoing edge, the same property holds for $\mathcal{G}[U]$. By a slight abuse of notation, we denote the sub-game induced by U by $(\mathcal{G}[U], \alpha)$, where the priority function α is evaluated only on U and we say that the highest priority of $(\mathcal{G}[U], \alpha)$ is $\max_{v \in U} \alpha(v)$.

Attractors. In a game graph $\mathcal{G}$, a *z-attractor* $Attr_z(\mathcal{G}, U)$ of a set $U \subseteq V$ is the set of vertices from which player z has a strategy to reach U against all strategies of player $\overline{z}$. We have that $U \subseteq Attr_z(\mathcal{G}, U)$. A z-attractor can be constructed inductively as follows: Let $Z_0 = U$; and for all $i \geq 0$ let

$$\begin{aligned} Z_{i+1} = Z_i &\cup \{v \in V_z \mid Out(v) \cap Z_i \neq \emptyset\} \\ &\cup \{v \in V_{\overline{z}} \mid Out(v) \subseteq Z_i\}. \end{aligned} \tag{‡}$$

Then $Attr_z(\mathcal{G}, U) = \bigcup_{i \geq 0} Z_i$.

The following lemma summarizes two well-known facts about attractors and traps that we use frequently.

Lemma 2.2. *Let $U \subseteq V$.*

(1) [3, 37] *The attractor $A = Attr_z(\mathcal{G}, U)$ can be computed in $O(\sum_{v \in A} |In(v)|)$ time.*
(2) [57, Lemma 4] *Let $\mathcal{G}$ be a game graph in which each vertex has at least one outgoing edge. Then the set $V \setminus Attr_z(\mathcal{G}, U)$ is a z-trap in $\mathcal{G}$.*

Dominions. A non-empty set of vertices $D \subseteq V$ is a *z-dominion* if player z has a winning strategy from every vertex of D that also ensures only vertices of D are visited. Note that a z-dominion is also a $\overline{z}$-trap and that the z-attractor of a z-dominion is again a z-dominion. In parity games dominions of size $|D| \leq h+1$ can be computed by running the small-progress measure algorithm of Jurdziński [39] with a reduced codomain [53]. We use this algorithm, whose properties are stated in the lemma below, as a subroutine.

Lemma 2.3 ([53])**.** *Let $(\mathcal{G}, \alpha)$ be a parity game with game graph $\mathcal{G}$, n vertices, m edges, a priority function α, and c priorities. There is an algorithm* `ProgressMeasure`$(\mathcal{G}, \alpha, h, z)$ *that returns the union of all z-dominions of size at most $h+1$, including a winning strategy for z, in time $O(c \cdot m \cdot \binom{h + \lceil c/2 \rceil}{h})$.*

The lemma below summarizes some well-known facts about dominions and winning sets.

Lemma 2.4. *The following assertions hold for game graphs $\mathcal{G}$ with at least one outgoing edge per vertex and (in particular) parity objectives. Let $U \subseteq V$.*

(1) [41, Lemma 4.4], [45, Lemma 2.6.11] *Let U be a z-trap in $\mathcal{G}$. Then a $\overline{z}$-dominion in $\mathcal{G}[U]$ is a $\overline{z}$-dominion in $\mathcal{G}$.*
(2) [41, Lemma 4.1] *The set $W_z(\mathcal{G})$ is a z-dominion.*
(3) [41, Lemma 4.5] *Let U be a subset of the winning set $W_z(\mathcal{G})$ of player z and let A be its z-attractor $Attr_z(\mathcal{G}, U)$. Then the winning set $W_z(\mathcal{G})$ of the player z is the union of A and the winning set $W_z(\mathcal{G}[V \setminus A])$, and the winning set $W_{\overline{z}}(\mathcal{G})$ of the opponent $\overline{z}$ is equal to $W_{\overline{z}}(\mathcal{G}[V \setminus A])$.*

2.2. **Algorithm.** In this subsection we present our new algorithm to compute the winning sets of player $\mathcal{E}$ and player $\mathcal{O}$ in a parity game $\mathcal{P} = (\mathcal{G}, \alpha)$ with $c \leq \sqrt{n}$ priorities in time $O(n^{1+\gamma(c+1)})$, where $\gamma(c) = c/3+1/2-4/(c^2-1)$ for odd c, and $\gamma(c) = c/3+1/2-1/(3c)-4/c^2$ for even c. The slightly simpler special case for $c = 3$ is described explicitly in the conference version [14].

High level idea and relation to existing algorithms. The classical algorithm for parity games [46, 57] identifies in each iteration a player-$\overline{z}$ dominion by a recursive call to a parity game with one priority less, then removes its $\overline{z}$-attractor from the game graph and recurses on the remaining game graph until no $\overline{z}$-dominion can be found any more; the remaining vertices form the winning set of player z. The sub-exponential time algorithm of [41] limits the number of recursive calls by computing $\overline{z}$-dominions with up to $\sqrt{n}$ vertices in a brute-force manner before each recursive call. In the Big-Step algorithm [52, 53] this search for dominions before the recursive calls is made more efficient by adapting the small progress measure algorithm for parity games [39] to computing dominions of a bounded size. Our algorithm refines the Big-Step algorithm of [52, 53] for dense graphs; the main new insight is that to find dominions of size at most h, it is sufficient to consider a specific subgame with $O(n \cdot h)$ edges. For the special case of parity games with 2 priorities, i.e., Büchi games, our algorithm is equivalent to the algorithm of [13]. We next describe our algorithm and then prove its correctness and running time.

Initialization (steps 1–3 of Procedure Parity). If all vertices of the parity game have priority zero, player $\mathcal{E}$ wins from all vertices and thus the algorithm terminates and returns the set of all vertices as the winning set of player $\mathcal{E}$ and the empty set as the winning set of player $\mathcal{O}$. Otherwise we set z according to the parity of the highest priority $c - 1$ in the parity game. The procedure then iteratively determines the winning set of player $\overline{z}$ and in the end identifies W_z as the complement of $W_{\overline{z}}$.

Iterated vertex deletions (steps 4–11 of Procedure Parity). The algorithm repeatedly removes vertices from the game graph $\mathcal{G}$. During the algorithm, we denote by $\mathcal{G}$ the remaining game graph after vertex deletions. The vertex removal is achieved by identifying parts of the winning set of player $\overline{z}$, i.e, $\overline{z}$-dominions, and removing their attractors from the maintained game graph.

Dominion find and attractor removal. The algorithm repeatedly finds dominions of player $\overline{z}$ in parity games $\mathcal{P}'$ where the highest priority is at most $c - 2$. The parity game $\mathcal{P}'$ is constructed by removing the z-attractor of vertices with priority $c-1$ from $\mathcal{G}$ (this is implicit

Procedure `Parity`($\mathcal{G}$, α, h)

Input : *game graph* $\mathcal{G} = ((V, E), (V_\mathcal{E}, V_\mathcal{O}))$ with $n = |V|$,
priority function $\alpha : V \to [c]$ with $c \geq 1$, and
parameter $h \in [1, n] \cap \mathbb{N}$

Output : winning sets $(W_\mathcal{E}, W_\mathcal{O})$ of player $\mathcal{E}$ and player $\mathcal{O}$

1 **if** $c = 1$ **then return** $(V, \emptyset)$
2 let z be player $\mathcal{E}$ if c is odd and player $\mathcal{O}$ otherwise
3 $W_{\overline{z}} \leftarrow \emptyset$
4 **repeat**
5 $\quad W'_{\overline{z}} \leftarrow Dominion(\mathcal{G}, \alpha, h, \overline{z})$
6 $\quad$ **if** $W'_{\overline{z}} = \emptyset$ **and** $c \geq 3$ **then**
7 $\quad\quad \mathcal{G}' \leftarrow \mathcal{G} \setminus Attr_z(\mathcal{G}, \alpha^{-1}(c-1))$
8 $\quad\quad (W'_\mathcal{E}, W'_\mathcal{O}) \leftarrow Parity(\mathcal{G}', \alpha)$
9 $\quad A \leftarrow Attr_{\overline{z}}(\mathcal{G}, W'_{\overline{z}})$; $W_{\overline{z}} \leftarrow W_{\overline{z}} \cup A$
10 $\quad \mathcal{G} \leftarrow \mathcal{G} \setminus A$
11 **until** $W'_{\overline{z}} = \emptyset$
12 $W_z \leftarrow V \setminus W_{\overline{z}}$
13 **return** $(W_\mathcal{E}, W_\mathcal{O})$

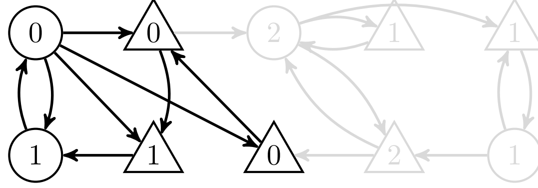

Figure 2: The resulting Büchi game $(\mathcal{G}', \alpha)$ (in black) after removing $Attr_\mathcal{E}(\mathcal{G}, \alpha^{-1}(2))$ from $\mathcal{G}$. The vertex in the top left corner is an $\mathcal{E}$-vertex with out-degree larger than $\sqrt{n}$, i.e., a blue vertex of $\mathcal{G}'_i$ for $i \leq \log_2(\sqrt{n})$. The two Büchi vertices in the bottom left corner are contained in $Attr_\mathcal{E}(\mathcal{G}'_i, Bl_i)$ and we have $\mathcal{D}_i = \emptyset$.

in the Procedure Dominion and explicit before the recursive call to Procedure Parity; more details follow). After a $\overline{z}$-dominion in the parity game $\mathcal{P}'$ is found, its $\overline{z}$-attractor is removed from $\mathcal{G}$. Then the search for $\overline{z}$-dominions is continued on the remaining vertices. If all vertices in the parity game $\mathcal{P}'$ are winning for z, i.e., no $\overline{z}$-dominion exists in $\mathcal{P}'$, then the procedure terminates. The winning set of player $\overline{z}$ is the union of the $\overline{z}$-attractors of all found $\overline{z}$-dominions. The remaining vertices are winning for player z. We now describe the steps to find $\overline{z}$-dominions.

Steps of dominion find. For the search for $\overline{z}$-dominions in the parity game $\mathcal{P}'$ we use two different procedures, Procedure Dominion and a recursive call to Procedure Parity. We first search for "small" $\overline{z}$-dominions with up to h vertices with Procedure Dominion, where h is a parameter that will be set later to balance the running times of the two procedures. If no $\overline{z}$-dominion is found, we can conclude that either all $\overline{z}$-dominions contain more than h vertices or the winning set of $\overline{z}$ on the current game graph is empty. In the latter case the

Procedure `Dominion`($\mathcal{G}$, α, h, $\overline{z}$)

Input : *game graph* $\mathcal{G} = ((V, E), (V_\mathcal{E}, V_\mathcal{O}))$,
priority function $\alpha : V \to [c]$ with $c \geq 2$,
parameter $h \in [1, n] \cap \mathbb{N}$, and
player $\overline{z}$

Output : a $\overline{z}$-dominion that contains all $\overline{z}$-dominions with at most h vertices or possibly the empty set if no such $\overline{z}$-dominion exists

```
1  for i ← 1 to ⌈log_2(h)⌉ do
2      construct G_i
3      Bl_i ← {v ∈ V_z̄ | Outdeg(G_i, v) = 0} ∪ {v ∈ V_z | Outdeg(G, v) > 2^i}
4      G'_i ← G_i \ Attr_z(G_i, α^{-1}(c − 1) ∪ Bl_i)
5      if c = 2 then
6          D_i ← the vertex set of G'_i
7          e
8      else
9          D_i ← ProgressMeasure(G'_i, α, 2^i, z̄)
10     if D_i ≠ ∅ then
11         return D_i
12 return ∅
```

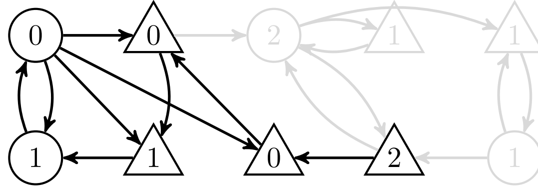

Figure 3: The winning set of player $\mathcal{O}$ in the Büchi game $(\mathcal{G}', \alpha)$ and its $\mathcal{O}$-attractor in $\mathcal{G}$.

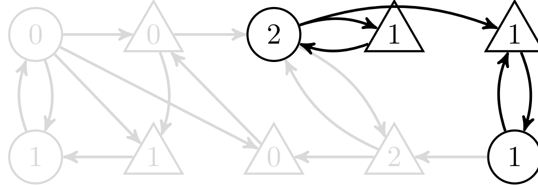

Figure 4: The remaining graph. The two vertices on the right are an $\mathcal{O}$-dominion, the remaining vertices form the winning set of player $\mathcal{E}$.

algorithm terminates. The former case occurs at most n/h times and in such a case we use the recursive call to Procedure Parity on the parity game $\mathcal{P}'$ with one priority less to obtain a $\overline{z}$-dominion. Below we describe the details of Procedure Dominion.

Example 2.5 (Illustration of the Algorithm for Parity-3.)**.** *Figure 1 shows a parity game with priorities* $\{0, 1, 2\}$*, and Figure 2 shows the Büchi game we obtain when we remove the* $\mathcal{E}$*-attractor of the vertices with the highest priority* 2*. Figure 3 shows the winning set of player* $\mathcal{O}$ *in the Büchi game, which is an* $\mathcal{O}$*-dominion in the parity game, and its* $\mathcal{O}$*-attractor in the original game graph. Finally, Figure 4 shows the remaining game graph after the removal of the* $\mathcal{O}$*-dominion and its attractor.*

Graph decomposition for Procedure Dominion. In Procedure Dominion we use the following *hierarchical graph decomposition*. For a game graph $\mathcal{G} = (G, (V_\mathcal{E}, V_\mathcal{O}))$ with $G = (V, E)$ we denote its decomposition with respect to player $\overline{z}$ by $\{\mathcal{G}_i\}$. The decomposition $\{\mathcal{G}_i\}$ consists of $\lceil \log_2 n \rceil$ game graphs $\mathcal{G}_i = (G_i, (V_\mathcal{E}, V_\mathcal{O}))$ with underlying graphs $G_i = (V, E_i)$. We call $1 \le i \le \lceil \log_2 n \rceil$ the *level* of $\mathcal{G}_i$. For the definition of E_i we consider the incoming edges of each vertex in a fixed order: First the edges from vertices of $V_{\overline{z}}$, then the remaining edges. The set of edges E_i contains for each vertex $v \in V$ with $\mathit{Outdeg}(G, v) \le 2^i$ 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) $|E_i| \le 2^{i+1} n$, and (3) $\mathcal{G}_{\lceil \log_2 n \rceil} = \mathcal{G}$. We color z-vertices v with $\mathit{Outdeg}(G, v) > 2^i$ and $\overline{z}$-vertices without outgoing edges in $\mathcal{G}_i$ *blue* in $\mathcal{G}_i$ and denote the set of blue vertices by Bl_i. All other vertices are called *white*.

Procedure Dominion. The Procedure `Dominion` searches in the game graph $\mathcal{G}_i$, starting at $i = 1$. As long as no $\overline{z}$-dominion is found, the level i is increased one by one up to at most $i = \lceil \log_2(h) \rceil$. At each level only a subgame of $\mathcal{G}_i$ that only contains white vertices is considered. This subgame is obtained by removing the z-attractor of the blue vertices Bl_i and the vertices with the maximum priority $c - 1$ from $\mathcal{G}_i$. This (a) reduces reduce the number of priorities in the parity game by one and (b) ensures that $\overline{z}$-dominions found in the subgame are also $\overline{z}$-dominions in $\mathcal{G}$. In the subgame of $\mathcal{G}_i$ for $c \ge 3$ the `ProgressMeasure` algorithm (Lemma 2.3) is used to find $\overline{z}$-dominions of size at most $O(2^i)$. For $c = 2$, i.e., Büchi games, we have that any non-empty set of remaining vertices is a dominion of player $\overline{z}$.

Outline correctness. The correctness of Procedure Parity follows from the correctness of the Big-Step algorithm as soon as we have established the correctness of Procedure Dominion. Let V_i' be the vertices of $\mathcal{G}_i' = \mathcal{G}_i \setminus \mathit{Attr}_z(\mathcal{G}_i, \alpha^{-1}(c-1) \cup \mathit{Bl}_i)$. The correctness of Procedure Dominion follows from the following arguments:

(1) the vertex set V_i' is a z-trap in $\mathcal{G}_i$ and thus a $\overline{z}$-dominion of $\mathcal{G}_i'$ is a $\overline{z}$-dominion of $\mathcal{G}_i$;
(2) the vertices of V_i' are white and hence a $\overline{z}$-dominion of $\mathcal{G}_i$ is a $\overline{z}$-dominion of $\mathcal{G}$; and
(3) thus the correctness of Procedure Dominion follows from the correctness of the progress measure algorithm that is used for determining dominions in $\mathcal{G}_i'$.

Outline running time. We first analyze the running time without the recursive calls and then show a bound of $O(n^{1+\gamma(c+1)})$ for the running time including the recursive calls by induction over c. For the latter we use that we only do recursive calls when no $\overline{z}$-dominion of size at most h exists and we balance the cost of the recursive calls and the other operations by setting the parameter h accordingly, similar to [52]. The running time without the recursive calls is, for $c \ge 3$, dominated by the running time of Procedure Dominion. For Procedure Dominion we show that any $\overline{z}$-dominion that, including its $\overline{z}$-attractor, contains at most 2^i vertices can be identified as a $\overline{z}$-dominion in $\mathcal{G}_i$. Thus if some $\overline{z}$-dominion is found in the search for $\overline{z}$-dominions at level i but was not found at level $i-1$, then it must contain $\Omega(2^i)$ vertices. Since $\mathcal{G}_i$ contains $O(2^i \cdot n)$ edges, we can account for the time that was spent

proportional to the edges of $\mathcal{G}_i$ by charging it, with an additional factor of n, to the vertices in the $\overline{z}$-dominion. This is crucial for the improved running time on dense graphs.

In the remaining part of this section we prove the correctness and running time of Procedure Parity (including the calls to Procedure Dominion). The following lemma captures the essence of why the hierarchical graph decomposition is helpful for graph games. The lemma is a generalization of related lemmata for Büchi games in [13] and for parity-3 games in the conference version of this paper [14] and was also used in subsequent work [9]. The first part of the lemma is essential for the correctness of the hierarchical graph decomposition technique on game graphs: It shows that every z-trap in $\mathcal{G}_i$ that contains only white vertices is also a z-trap in $\mathcal{G}$. The second part is essential for the running time: Every z-trap in $\mathcal{G}$ that, including its $\overline{z}$-attractor, contains at most h vertices, is a z-trap induced by white vertices in $\mathcal{G}_{\lceil \log_2 h \rceil}$. Note that the $\overline{z}$-attractor of a z-trap is itself a z-trap and thus this holds in particular for maximal z-traps of size at most h.

Lemma 2.6. *Let $\mathcal{G} = ((V, E), (V_{\mathcal{E}}, V_{\mathcal{O}}))$ be a game graph and $\{\mathcal{G}_i\}$ its hierarchical graph decomposition w.r.t. to player $\overline{z}$. For $1 \le i \le \lceil \log_2 n \rceil$ let Bl_i be the set consisting of the player-$\overline{z}$ vertices that have no outgoing edge in $\mathcal{G}_i$ and the player-z vertices with more than 2^i outgoing edges in $\mathcal{G}$.*

1. *If a set $S \subseteq V \setminus Bl_i$ is a z-trap in $\mathcal{G}_i$, then S is a z-trap in $\mathcal{G}$.*
2. *If a set $S \subseteq V$ is a z-trap in $\mathcal{G}$ and $|Attr_{\overline{z}}(\mathcal{G}, S)| \le 2^i$, then (i) S is a z-trap in $\mathcal{G}_i$, (ii) the set S is in $V \setminus Bl_i$, and (iii) $\mathcal{G}_i[S] = \mathcal{G}[S]$.*

Proof.

1. By $S \subseteq V \setminus Bl_i$ we have for all $v \in S \cap V_z$ that $Out(G, v) = Out(G_i, v)$. Thus if $Out(\mathcal{G}_i, v) \subseteq S$, then also $Out(G, v) \subseteq S$. Each edge of G_i is contained in G, thus we have for all $v \in S \cap V_{\overline{z}}$ that $Out(G_i, v) \cap S \neq \emptyset$ implies $Out(G, v) \cap S \neq \emptyset$.
2. By the definition of a $\overline{z}$-attractor, all $u \in V_{\overline{z}}$ for which there exists a $w \in S$ such that $(u, w) \in E$ are contained in $Attr_{\overline{z}}(\mathcal{G}, S)$. Thus by $|Attr_{\overline{z}}(\mathcal{G}, S)| \le 2^i$ all vertices of S have less than 2^i incoming edges from vertices of $\overline{z}$. Hence by the ordering of the incoming edges in the construction of $\mathcal{G}_i$
   (a) all edges (u, w) with $u \in V_{\overline{z}}$ and $w \in S$ are contained in E_i and thus in particular
   (b) $(u, w) \in E_i$ for all $(u, w) \in E$ with $u \in S \cap V_{\overline{z}}$ and $w \in S$.
   Note furthermore that since S is a z-trap in $\mathcal{G}$, there exists a $w \in S$ such that $(u, w) \in E$ for all $u \in S \cap V_{\overline{z}}$; by (a) we have $(u, w) \in E_i$ for this w. Together with $E_i \subseteq E$ it follows that (i) S is a z-trap in $\mathcal{G}_i$. The existence of such an edge (u, w) also ensures that every vertex in $V_{\overline{z}} \cap S$ has an outgoing edge in $\mathcal{G}_i$, i.e.,
   (c) $Bl_i \cap V_{\overline{z}} \cap S = \emptyset$.
   Since S is a trap for player z in $\mathcal{G}$ and $|S| \le 2^i$, we have for all $v \in V_z$ that $Outdeg(G, v) < 2^i$. Thus
   (d) $Bl_i \cap V_z \cap S = \emptyset$ and
   (e) $(u, w) \in E_i$ for all $(u, w) \in E$ with $u \in S \cap V_z$ and $w \in S$.
   Combining (c) and (d) yields (ii) $S \cap Bl_i = \emptyset$, and (b) and (e) give (iii) $\mathcal{G}_i[S] = \mathcal{G}[S]$. □

In the next lemma we consider not just z-traps but $\overline{z}$-dominions and thus argue additionally about winning strategies of player $\overline{z}$. The first part of the lemma shows the soundness of Procedure Dominion, while the second part shows completeness and is crucial for the running time analysis of the overall algorithm.

Lemma 2.7. *Let the parity game $(\mathcal{G}, \alpha)$ with c priorities, the parameter h, and the player $\overline{z}$ be the input to Procedure Dominion. Let D_i, $\mathcal{G}_i'$ and Bl_i be as in Procedure Dominion.*

(1) *Every $D_i \neq \emptyset$ is a $\overline{z}$-dominion in the parity game $(\mathcal{G}, \alpha)$ and the priority of the vertices of D_i is at most $c-2$.*
(2) *If there exists a $\overline{z}$-dominion D with $|Attr_{\overline{z}}(\mathcal{G}, D)| \leq 2^i$ in $(\mathcal{G}, \alpha)$ such that the highest priority in D is at most $c-2$, then D is a $\overline{z}$-dominion in the parity game $(\mathcal{G}_i', \alpha)$.*

Proof.

(1) By definition the highest priority in $\mathcal{G}_i'$ and thus in D_i is at most $c-2$. Let V_i' be the vertices of $\mathcal{G}_i'$. By Lemma 2.2 (2) the set V_i' is a z-trap in $\mathcal{G}_i$ and by $V_i' \cap Bl_i = \emptyset$ and Lemma 2.6 (1) also in $\mathcal{G}$. If $c = 2$, we have $V_i' = D_i$ and since D_i only contains vertices with priority 0, the set D_i is a $\overline{z}$-dominion in $(\mathcal{G}, \alpha)$. Suppose now $c \geq 3$. Then D_i is the set returned by `ProgressMeasure`$(\mathcal{G}_i', \alpha, 2^i, \overline{z})$ and we have by Lemma 2.3 that D_i is a $\overline{z}$-dominion in the parity game $(\mathcal{G}_i', \alpha)$. Since V_i' is a z-trap in $\mathcal{G}_i$, the set D_i is also a $\overline{z}$-dominion in $\mathcal{G}_i$ by Lemma 2.4 (1). Since D_i contains only white vertices, we have
   (a) the set D_i is a z-trap in $\mathcal{G}$ by Lemma 2.6 (1) and
   (b) all outgoing edges of vertices of $V_z \cap D_i$ are present in $\mathcal{G}_i$.

   Thus by $E_i \subseteq E$ the winning strategy of player $\overline{z}$ for the vertices of D_i in $\mathcal{G}_i$ is also a winning strategy in $\mathcal{G}$ and hence D_i is a $\overline{z}$-dominion in the parity game $(\mathcal{G}, \alpha)$.
(2) By Lemma 2.6 (2) we have (i) D is a z-trap in $\mathcal{G}_i$, (ii) $D \cap Bl_i = \emptyset$, and (iii) $\mathcal{G}[D] = \mathcal{G}_i[D]$. Thus
   (a) D is contained in $\mathcal{G}_i'$ and
   (b) player $\overline{z}$ can play the same winning strategy in $\mathcal{G}_i[D]$ as in $\mathcal{G}[D]$. □

In the following corollary we state the insights of the previous two lemmata as needed for the running time analysis. The first part shows that when we use the hierarchical graph decomposition with increasing level i to search for a $\overline{z}$-dominion and we have to go up to level i^*, then the found $\overline{z}$-dominion, or at least its $\overline{z}$-attractor (which is again a $\overline{z}$-dominion), contains a number of vertices proportional to 2^{i^*}, which allows us to charge the work done in the search to the vertices in the identified dominion. The second part of the corollary states that no "small" $\overline{z}$-dominions exist in the maintained parity game if Procedure Dominion returns the empty set, where "small" is specified by the parameter h that will be set to balance the running time of Procedure Dominion and the recursive calls. In this case either no $\overline{z}$-dominion exists in the parity game and the algorithm terminates or the subsequent recursive call identifies a $\overline{z}$-dominion with more than h vertices; the latter can happen at most $O(n/h)$ times and allows us to bound the number of iterations in Procedure Parity.

Recall that we set z to $\mathcal{E}$ if the highest priority in the parity game is even and to odd otherwise, i.e., when we search for $\overline{z}$-dominions, we search for dominions of the player that tries to avoid the highest priority. For the proof of the second part we use that in this case every $\overline{z}$-dominion D contains a subset D' that is a $\overline{z}$-dominion itself and does not contain any vertex with the highest priority $c-1$ (Proposition 2 of [39]). Intuitively, D' is the set of vertices that are contained in the cycles of D that are induced by the memoryless winning strategy of player $\overline{z}$ in $\mathcal{G}[D]$.

Corollary 2.8. *Let the parity game $(\mathcal{G}, \alpha)$ with c priorities, the parameter h, and the player $\overline{z}$ be the input to Procedure Dominion. Let D_i, $\mathcal{G}_i'$ and Bl_i be as in Procedure Dominion.*

(1) *If for some $i > 1$ we have $D_i \neq \emptyset$ but $D_{i-1} = \emptyset$, then $|Attr_{\overline{z}}(\mathcal{G}, D_i)| > 2^{i-1}$.*

(2) *If Procedure Dominion returns the empty set, then we have for every $\overline{z}$-dominion D in the given parity game $|Attr_{\overline{z}}(\mathcal{G}, D)| > h$.*

Proof.

(1) By Lemma 2.7 (1) we have that D_i is a $\overline{z}$-dominion in $(\mathcal{G}, \alpha)$ and the vertices of D_i have a priority of at most $c-2$. Assume by contradiction that $|Attr_{\overline{z}}(\mathcal{G}, D_i)| \leq 2^{i-1}$. Then by Lemma 2.7 (2) the set D_{i-1} contains D_i and thus is not empty, a contradiction.

(2) Assume by contradiction that Procedure Dominion returns the empty set and there exists a $\overline{z}$-dominion D with $|Attr_{\overline{z}}(\mathcal{G}, D)| \leq h$ in $(\mathcal{G}, \alpha)$. By Proposition 2 of [39] in this case also a $\overline{z}$-dominion $D' \subseteq D$ exists such that the vertices of D' have priority at most $c-2$. By Lemma 2.7 (2) the set D' is a $\overline{z}$-dominion in the parity game $(\mathcal{G}'_i, \alpha)$ for $i \geq \lceil \log_2(|D'|) \rceil$ and thus in particular for $i = \lceil \log_2(h) \rceil$. Hence by Lemma 2.3 Procedure Dominion returns a non-empty set, a contradiction. □

In the following lemma we use Lemma 2.3 that bounds the running time of the calls to the progress measure algorithm together with the relation between levels in the hierarchical graph decomposition and the size of $\overline{z}$-dominions, provided by the corollary above, to bound the running time of Procedure Parity without recursive calls.

Lemma 2.9. *Let $(\mathcal{G}, \alpha)$ be a parity game with a game graph $\mathcal{G} = ((V, E), (V_\mathcal{E}, V_\mathcal{O}))$ with $n = |V|$, a priority function α, and c priorities and let $h \in [1, n]$ be a parameter that for $c = 2$ is equal to $h = n$. The running time of Procedure Parity($\mathcal{G}$, α)* without the recursive calls*, and without the attractor computation before the recursive calls, is $O\left(c \cdot n^2 \cdot \binom{h + \lfloor c/2 \rfloor}{h}\right)$ for $c \geq 3$, $O(n^2)$ for $c = 2$, and $O(n)$ for $c = 1$.*

Proof. All operations before and after the repeat-until loop can be done in $O(n)$ time, which shows $O(n)$ for $c = 1$. Further the attractor computations and the updates of the maintained sets in lines 9–10 can be done in total time $O(m)$. Thus it remains to bound the total time for the calls to Procedure Dominion.

To efficiently construct the graphs $\mathcal{G}_i$ and the vertex sets Bl_i, we maintain ordered lists of the incoming and outgoing edges of each vertex. These lists can be updated whenever an obsolete entry is encountered in the construction of $\mathcal{G}_i$; as each entry is removed at most once, this takes total time $O(m)$.

We now analyze the time spent in an iteration i of the for-loop in Procedure Dominion. The graph $\mathcal{G}_i$ contains $O(2^i \cdot n)$ edges and both $\mathcal{G}_i$ and Bl_i can be constructed from the maintained lists of in- and outedges in $O(2^i \cdot n)$ time. Also the attractor computation takes time $O(2^i \cdot n)$. Thus for $c = 2$ the time in iteration i is $O(2^i \cdot n)$, while for $c \geq 3$ the time is dominated by the call to `ProgressMeasure`. Note that with the attractor computation the vertices with the highest priority are removed from the parity game, thus the call to `ProgressMeasure` is done for a parity game with $c-1$ priorities and parameter $h = 2^i$. Hence by Lemma 2.3 iteration i for $c \geq 3$ takes time

$$O\left(c \cdot n \cdot 2^i \cdot \binom{2^i + \lceil (c-1)/2 \rceil}{2^i}\right) = O\left(c \cdot n \cdot 2^i \cdot \binom{2^i + \lfloor c/2 \rfloor}{2^i}\right).$$

The time for all iterations up to the i-th iteration forms a geometric series and thus satisfies the same running time bound.

Let i^* be the last iteration of the for-loop in a call to Procedure Dominion. Let z be $\mathcal{E}$ if c is odd and $\mathcal{O}$ otherwise. By Corollary 2.8 either

(1) D_{i^*} is a $\overline{z}$-dominion with $|Attr_{\overline{z}}(\mathcal{G}, D_{i^*})| > 2^{i^*-1}$ vertices or

(2) $i^* = \lceil \log_2(h) \rceil$ and $\mathcal{G}$ does not contain any $\overline{z}$-dominion D with $Attr_{\overline{z}}(\mathcal{G}, D) \leq h$ vertices. In case (1) more than 2^{i^*-1} vertices are removed from the maintained graph in this iteration. We charge each of these vertices $O(c \cdot n \cdot \binom{2^i + \lfloor c/2 \rfloor}{2^i})$ time, which can be bounded by $O(c \cdot n \cdot \binom{h+\lfloor c/2 \rfloor}{h})$ (per vertex). Hence the total running time is bounded by

$$O\left(c \cdot n^2 \cdot \binom{h + \lfloor c/2 \rfloor}{h}\right) .$$

In case (2) either

(a) $c \geq 3$ and a $\overline{z}$-dominion with more than h vertices in its $\overline{z}$-attractor is detected in the subsequent recursive call to Procedure Parity or

(b) there is no $\overline{z}$-dominion in the maintained parity game and this is the last iteration of the repeat-until loop in the Procedure Parity.

In Case (a) we have again that more than 2^{i^*-1} vertices are removed from the maintained graph in this iteration and thus can apply the same argument as for Case (1). Case (b) can happen at most once and its running time is bounded by $O(n^2)$ for $c = 2$ and by

$$O\left(c \cdot n \cdot 2^{\log_2(h)} \cdot \binom{2^{\log_2(h)} + \lfloor c/2 \rfloor}{2^{\log_2(h)}}\right)$$

for $c \geq 3$, which can be bounded by $O(c \cdot n^2 \cdot \binom{h+\lfloor c/2 \rfloor}{h})$. □

To bound the running time including the recursive calls, we use a similar analysis and similar parameters as for the Big-Step algorithm in [52, 53]. Let $\gamma(c) = c/3 + 1/2 - 4/(c^2-1)$ for odd c and $\gamma(c) = c/3 + 1/2 - 1/(3c) - 4/c^2$ for even c. Further let $\beta(c) = \gamma(c)/(\lfloor c/2 \rfloor + 1)$. It can easily be verified that $\gamma(c+1) = 1 + \gamma(c) - \beta(c)$ holds. We set $h = n^{\beta(c)}$ for $c \geq 3$ and additionally $h = n$ for $c = 2$. We show by induction over c a running time bound of $O(n^{1+\gamma(c+1)}) = O(n^{2+\gamma(c)-\beta(c)})$ for parity games with c priorities. The running time of the Big-Step algorithm for parity games with c priorities is $O(m \cdot (6e^{5/3}n/c^2)^{\gamma(c)})$, i.e., for a constant number of priorities we replace m by $n^{2-\beta(c)}$.

Lemma 2.10 (Running time)**.** *For parity games with $c \leq \sqrt{n}$ priorities Procedure Parity takes time $O(n^{1+\gamma(c+1)})$, where $\gamma(c) = c/3 + 1/2 - 4/(c^2-1)$ for odd c and $\gamma(c) = c/3 + 1/2 - 1/(3c) - 4/c^2$ for even c.*

Proof. For the base case of $c = 2$ we have $\gamma(c+1) = 1$ and no recursive calls. Thus the running time of Procedure Parity for $c = 2$ is $O(n^2)$ by Lemma 2.9 (in this case Procedure Parity is equivalent to the algorithm of [13]). Suppose Procedure Parity runs in time $O(n^{1+\gamma(c)})$ for a parity game with $c - 1 \geq 2$ priorities. We show that this implies that Procedure Parity runs in time $O(n^{1+\gamma(c+1)})$ for a parity game with c priorities for $3 \leq c \leq \sqrt{n}$. Let $h = n^{\beta(c)}$ for $\beta(c) = \gamma(c)/(\lfloor c/2 \rfloor + 1)$. We have $\beta(c) \geq 1/2$ for all $c \geq 3$ and thus $h \geq \sqrt{n}$. By Lemma 2.9 the time spent in Procedure Parity without the recursive calls is $O(c \cdot n^2 \cdot \binom{h+\lfloor c/2 \rfloor}{h})$. With Stirling's approximation of $(x/e)^x \leq x!$ we have

$$\binom{h + \lfloor c/2 \rfloor}{h} \leq \frac{(h + \lfloor c/2 \rfloor)^{\lfloor c/2 \rfloor}}{\lfloor c/2 \rfloor !} ,$$

$$\leq \left(\frac{(h + \lfloor c/2 \rfloor) \cdot e}{\lfloor c/2 \rfloor}\right)^{\lfloor c/2 \rfloor} .$$

Using $3 \leq c \leq \sqrt{n} \leq h$, we obtain

$$\left(\frac{(h + \lfloor c/2 \rfloor) \cdot e}{\lfloor c/2 \rfloor}\right)^{\lfloor c/2 \rfloor} \leq \left(\frac{2eh + ec}{c-1}\right)^{\lfloor c/2 \rfloor}, \leq \left(\frac{5eh}{c}\right)^{\lfloor c/2 \rfloor}.$$

Thus we have

$$c \cdot \binom{h + \lfloor c/2 \rfloor}{h} \leq \frac{(5e)^{\lfloor c/2 \rfloor}}{c^{\lfloor c/2 \rfloor - 1}} h^{\lfloor c/2 \rfloor},$$

which is $\leq h^{\lfloor c/2 \rfloor}$ for $c \geq (5e)^{3/2}$ and $\leq \kappa h^{\lfloor c/2 \rfloor}$ for some constant κ for $c < (5e)^{3/2}$. Hence the time without the recursive calls is bounded by $O(n^2 \cdot h^{\lfloor c/2 \rfloor})$, which is equal to $O(n^{2+\beta(c)\lfloor c/2 \rfloor})$ for $h = n^{\beta(c)}$. By the choice of $\beta(c)$ we have $\beta(c)\lfloor c/2 \rfloor = \gamma(c) - \beta(c)$ and thus we can write this bound as $O(n^{2+\gamma(c)-\beta(c)})$. By Corollary 2.8 there are at most $O(n/h) = O(n^{1-\beta(c)})$ recursive calls to Procedure Parity. Each recursive call is for a parity game with one priority less and thus takes time $O(n^{1+\gamma(c)})$. Hence the total time for all recursive calls is bounded by $O(n^{1-\beta(c)+1+\gamma(c)})$. For $\gamma(c)$ as defined above we have $\gamma(c+1) = 1 + \gamma(c) - \beta(c)$, which completes the proof. □

The correctness proof for Procedure Parity follows mostly the correctness proof of the classical algorithm for parity games and is folklore; additionally Lemma 2.7 (1) shows the correctness of Procedure Dominion.

Lemma 2.11 (Correctness)**.** *Given a parity game $\mathcal{P} = (\mathcal{G}, \alpha)$, let z be player $\mathcal{E}$ if c is odd and player $\mathcal{O}$ otherwise and let W_z and $W_{\overline{z}}$ be the output of Procedure Parity. We have: (1)* (Soundness). *$W_{\overline{z}} \subseteq W_{\overline{z}}(P)$; and (2)* (Completeness). *$W_{\overline{z}}(P) \subseteq W_{\overline{z}}$.*

Proof. The proof is by induction over the number of priorities c. The induction base is $c = 1$, where the algorithm correctly returns $W_{\mathcal{E}} = V$ and $W_{\mathcal{O}} = \emptyset$. Suppose the algorithm is correct for parity games with $c - 1$ priorities. We show the correctness for parity games with c priorities by proving (1) by induction on the iterations of the repeat-until loop that all vertices of $W_{\overline{z}}$ are indeed winning for player $\overline{z}$ (soundness) and then (2) construct a winning strategy for player z on the remaining vertices $W_z = V \setminus W_{\overline{z}}$ (completeness).

Soundness will follow from showing that whenever the set $W'_{\overline{z}}$ determined in an iteration of the repeat-until loop is not empty, then it is a $\overline{z}$-dominion. This is sufficient because by Lemma 2.4 (3) it is valid to determine a $\overline{z}$-dominion, remove its $\overline{z}$-attractor, and recurse on the remaining game graph; and hence $W'_{\overline{z}}$ being a $\overline{z}$-dominion implies soundness by induction over the repeat-until loop. If $W'_{\overline{z}} \neq \emptyset$ is returned by Procedure Dominion, then it is a $\overline{z}$-dominion by Lemma 2.7 (1). If $W'_{\overline{z}} \neq \emptyset$ is returned by the recursive call to Procedure Parity for the game graph $\mathcal{G}' = \mathcal{G} \setminus Attr_z(\mathcal{G}, \alpha^{-1}(c-1))$, then $W'_{\overline{z}}$ is a $\overline{z}$-dominion by the following argument: By the induction assumption the set $W'_{\overline{z}}$ is a $\overline{z}$-dominion in $\mathcal{G}'$. By Lemma 2.2 (2) the vertices of $\mathcal{G}'$ form a z-trap in $\mathcal{G}$ and by Lemma 2.4 (1) a $\overline{z}$-dominion in a subgame induced by a z-trap is a $\overline{z}$-dominion in the full game.

We now prove the completeness result. When Procedure Parity terminates, the winning set $W'_{\overline{z}}$ of player $\overline{z}$ in the parity game $(\mathcal{G}', \alpha)$ is empty. Also note that since the algorithm removes attractors of $\overline{z}$, the set W_z is a trap for $\overline{z}$ by Lemma 2.2 (2). Consider the set

$Z = \{v \in W_z \mid \alpha(v) = c-1\}$, its attractor $X = Attr_z(\mathcal{G}, Z)$, and the subgame induced by $U = W_z \setminus X$. Note that the game graphs $\mathcal{G}[U]$ and $\mathcal{G}'[U]$ coincide. Thus all vertices of U must be winning for player z in the parity game $(\mathcal{G}', \alpha)$ as otherwise $W'_{\overline{z}}$ would have been non-empty for $(\mathcal{G}', \alpha)$. We prove the lemma by describing a winning strategy for player z in $\mathcal{P}$ for all vertices in W_z. For vertices of $Z \cap V_z$ the winning strategy chooses an edge in W_z, which exists since W_z is a $\overline{z}$-trap. For vertices in $X \setminus Z$ player z follows her attractor strategy to Z. In the subgame induced by $U = W_z \setminus X$ player z follows her winning strategy in the parity game $(\mathcal{G}', \alpha)$. Then in a play either (i) X is visited infinitely often; or (ii) from some point on only vertices of U are visited. In the former case, the attractor strategy ensures that then some vertex of Z with priority $c-1$ is visited infinitely often; and in the later case, the subgame winning strategy ensures that the highest priority visited infinitely often has the same parity as $c-1$. It follows that $W_z \subseteq W_z(P)$, i.e., $W_{\overline{z}}(P) \subseteq W_{\overline{z}}$, and the desired result follows. □

Lemmata 2.10 and 2.11 yield the following result.

Theorem 2.12. Procedure Parity correctly computes the winning sets in parity games with n vertices and $c \leq \sqrt{n}$ priorities in $O(n^{1+\gamma(c+1)})$ time, where $\gamma(c) = c/3 + 1/2 - 4/(c^2-1)$ for odd c and $\gamma(c) = c/3 + 1/2 - 1/(3c) - 4/c^2$ for even c.

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 [39] requires $O(mn)$ time to compute the winning strategy of player $\mathcal{E}$ and $O(mn^2)$ time to compute the winning strategy for player $\mathcal{O}$; Schewe [53] shows how to modify the small-progress measure algorithm to compute the respective winning strategies of both players in $O(mn)$ time. Schewe's running time bound for general parity games also holds when both winning strategies are requested [53]. We show that our algorithm also computes the respective winning strategies without increasing the running time, i.e., in $O(n^{5/2})$ time for parity-3 games and in $O(n^{1+\gamma(c+1)})$ time for parity games with $c \leq \sqrt{n}$.

For *parity-3* we first observe that for Büchi games [13] we can construct in $O(n^2)$ time also the respective winning strategies of both players since the algorithm is based on identifying traps and attractors, and the corresponding winning strategies are identified immediately with the computation. The proof of Lemma 2.11 describes the strategy computation for a winning strategy of 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 $O(n^{5/2})$ time. For *general parity games* the winning strategies for both players are constructed in a similar way; the argument uses parity-3 games as base case and then induction over the recursive calls. Let z be player $\mathcal{E}$ if c is odd and player $\mathcal{O}$ otherwise. First note that the time bound in Lemma 2.3, and therefore the time bound of Procedure Dominion, includes the computation of a winning strategy for player $\overline{z}$ within a $\overline{z}$-dominion determined by Procedure Dominion. The winning strategy of player $\overline{z}$ is a combination of his winning strategies for the dominions identified in Procedure Dominion and the dominions identified in the recursive calls for parity games with one priority less and the corresponding attractor strategies. The winning strategy of player z, as described in Lemma 2.11, is identified in

the last iteration of the repeat-until loop and consists of her winning strategy for the parity game for which the last recursive call is made and her attractor strategy to vertices with priority $c-1$.

Corollary 2.13. *Winning strategies for player $\mathcal{E}$ and player $\mathcal{O}$ in their respective winning sets in parity games with n vertices and $c \leq \sqrt{n}$ priorities can be computed in $O(n^{1+\gamma(c+1)})$ time.*

3. Streett Objectives in Graphs

In this section we present our algorithm for graphs with Streett objectives, and an upper and a lower bound for reporting a certificate for a vertex in the winning set. 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 $b = \sum_{j=1}^{k}(|L_j| + |U_j|) \leq 2nk$. Our algorithm runs in time $O(n^2 + b \log n)$.

3.1. Preliminaries.

Let *Outdeg*(G, u) be the number of outgoing edges of vertex u in the graph G; we omit G if clear from the context. 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 `GraphReach`(G, S) be the set of vertices in G that *can reach* a vertex of $S \subseteq V$. A strongly connected component (SCC) of a directed graph $G = (V, E)$ is a subgraph $G[S]$ induced by a maximal subset of vertices $S \subseteq V$ such that there is a path in $G[S]$ between every pair of vertices in S. We use the abbreviation SCS to denote a strongly connected subgraph that is not necessarily an SCC (i.e., is not necessarily maximal w.r.t. strong connectivity). We call an SCS (resp. SCC) *trivial* if it only contains a single vertex and no edges. All other SCSs (resp. SCCs) are *non-trivial*. The set `GraphReach`(G, S) and the SCCs of a graph G can be found in linear time by, e.g., depth-first search [54].

Algorithm Streett *and good component detection.* Consider an SCC C; the *good component detection problem* asks to (a) output a non-trivial SCS $G[X] \subseteq C$ induced by some set of vertices X such that for all $1 \leq j \leq k$ either no vertex of L_j or at least one vertex of U_j is contained in the SCS (i.e., $L_j \cap X = \emptyset$ or $U_j \cap X \neq \emptyset$), or (b) detect that no such SCS 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 of the SCC 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 SCC decomposition of the graph;
(2) for each SCC C for which the good component detection returns an SCS, label the SCC C as *satisfying*;
(3) output the set of vertices that can reach a satisfying SCC as the winning set.

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

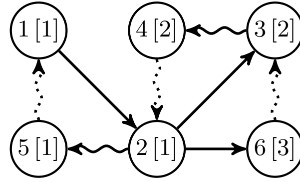

Figure 5: An example for a "jungle" constructed by Tarjan's SCC algorithm for an SCC. Backedges are dotted, spanning tree edges are solid. Backlinks are curved. The numbers of the vertices represent the order in which the vertices are visited, the numbers in brackets are the lowlinks.

3.2. **Certificate computation.** In this subsection we present our results for the certificate computation. Given a start vertex x that belongs to the winning set, a *certificate* or *lasso* [25] is a path from x that consists of a simple path and a not necessarily simple cycle such that the a play that traverses the cycle infinitely often satisfies the objective. From the certificate an example of an *accepting run*, i.e., an infinite path from x that satisfies the objective, can be constructed. The output of Algorithm Streett can be used to obtain a certificate. Let X be a set of vertices that induces a good component $G[X]$ and let x be a start vertex that can reach X. We generate a certificate for x being in the winning set as follows. A simple path from x to X can be found in linear time by a depth-first search. Let v be the vertex of X where this path ends. We call v the *root* of the good component $G[X]$. We show next how to obtain, in $O(m + n\min(n, k))$ time, from the good component $G[X]$ a cycle starting and ending at the root v such that the path resulting from the simple path and the cycle is indeed a certificate. 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 of $U_j \cap X$, i.e., we do not have to include *all* vertices of X.

We can use Tarjan's depth-first search based SCC algorithm [54] 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 the strongly connected (sub)graph $G[X]$ consists of a spanning tree and at most one *backedge* per vertex of 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.

Example 3.1 (Illustration of Tarjan's jungle graph.)**.** *Figure 5 shows the types of edges and the values at the vertices as assigned by Tarjan's SCC algorithm for a small example graph.*

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 appears 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 non-empty 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 by $O(m + n\min(n, k))$. This certificate has a size of $O(n\min(n, k))$.

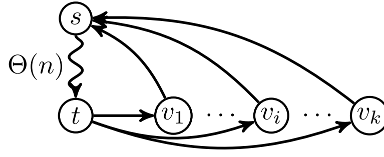

Figure 6: Let the only path between s and t be of length $\Theta(n/2) = \Theta(n)$, not containing any of the vertices v_j for $1 \le j \le k$. Each v_j has only an edge from t and an edge to s. Let the Streett pairs (L_j, U_j) be given by $L_j = \{s\}$ and $U_j = \{v_j\}$ for $1 \le j \le k$. Then the size of the smallest certificate is $\Theta(nk)$, where k can be of order n.

Example 3.2 (Lower bound.)**.** *Figure 6 shows that the smallest existing certificate can be as large as* $\Theta(n \min(n, k))$.

3.3. **Good Component Detection.** In this subsection we present the algorithm for good component detection. First we introduce the different concepts used in the algorithm for good component detection. We start with describing the hierarchical graph decomposition technique for this setting, which is crucial for the running time analysis.

Graph decomposition. In our algorithm we decompose a graph G in the following way. For $i \in \{1, \ldots, \lceil \log n \rceil\}$, let $G_i = (V, E_i)$ be a subgraph of G with $E_i = \{(u, v) \mid \mathit{Outdeg}(u) \le 2^i\}$, i.e., the edges of G_i are the outedges of the vertices with outdegree at most 2^i. Note that for $i = \lceil \log n \rceil$ we have that $G_i = G$. We say vertices in G with $\mathit{Outdeg}(v) > 2^i$ 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 $G = G_{\lceil \log n \rceil}$ are white and that all vertices in Bl_i have outdegree zero in G_i.

Top and bottom strongly connected components. The algorithm repeatedly finds a top or a bottom SCC in the remaining graph G. A bottom SCC $G[S]$ in a directed graph G, induced by some set of vertices S, 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. Note that every graph has at least one bottom and at least one top SCC. If the graph is not strongly connected, then there exist a top and a bottom SCC that are disjoint and thus one of them contains at most half of the vertices of G.

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 [35], 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 $1 \le j \le k$ the vertex is in L_j but it is not strongly connected to any vertex of U_j. All other vertices are *good.* Note that good vertices can become bad if some vertex deletion disconnects an SCS or a vertex of a set U_j is deleted. A good component is a non-trivial SCS 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 [35] an implementation of this data structure is described that achieves the following running times. For a set of vertices $S \subseteq V$ let $bits(S)$ be defined as $\sum_{j=1}^{k} (|S \cap L_j| + |S \cap U_j|)$.

Lemma 3.3 (Lemma 2.1 in [35]). *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`$(D(S))$*.*

Structure of algorithm. 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 GoodComp. 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 show that if a good component exists, its vertices 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 SCCs. Otherwise the subgraph $G[S]$ is either determined to be strongly connected and output as a good component or a "small" SCC in $G[S]$ is identified.

Search for small SCCs. To find a small SCC, the algorithm searches alternatingly 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 of $G[S]$. 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 $\lceil \log(|S|) \rceil$. 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. The Procedure `SmallestBSCC`(H') returns the set of vertices that induces the smallest bottom SCC in the graph H'. We then put the newly detected SCC and the "rest" of S back into Q.

Outline running time. 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 outer for-loop must contain $\Omega(2^i)$ vertices. In time $O(n \cdot 2^i)$ 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 $G_{i'}[S]$ for $1 \leq i' < i$ can be bounded by 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. We show that this yields a total running time of $O(n^2)$ for computing SCCs.

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 3.3 to obtain a running time of $O((n + b) \log n)$ for the maintenance of the data structures and the

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

Input : *strongly connected graph* $G = (V, E)$ and *Streett pairs* (L_j, U_j) for $j = 1, \dots, k$
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
5          D(S) ← Remove(S, D(S), Bad(D(S)))
6      if G[S] contains at least one edge then
7          for i ← 1 to ⌈log(|S|)⌉ do
8              foreach H ∈ {G, RevG} do
9                  construct H_i[S]
10                 Bl_i ← {v ∈ S | Outdeg(H, v) > 2^i}
11                 Z ← S \ GraphReach(H_i[S], Bl_i)
                   /* Z cannot reach Bl_i                                  */
12                 if Z ≠ ∅ then
13                     X ← SmallestBSCC(H_i[Z])
14                     if X = S then
                           /* good component found                         */
15                         return G[S]
16                     if |X| ≤ |S|/2 then
17                         add Remove(S, D(S), X) to Q
18                         add Construct(X) to Q
19                         continue with pull of next D(S) from Q (Line 3)
20 return no good component exists
```

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 3.4. *Let* $H \in \{G, RevG\}$ *be the graph and let* i^* *be the iteration for which in Algorithm* GOODCOMP *the outer for-loop stops. Let* Z *be the non-empty set* $S \setminus \texttt{GraphReach}(H_{i^*}[S], Bl_{i^*})$ *and let* X *be the set of vertices that induces the smallest bottom SCC* $H[X]$ *in* $H_{i^*}[Z]$ *returned by* $\texttt{SmallestBSCC}(H_{i^*}[Z])$*. Assume we have* $|X| \leq |S|/2$*. 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 of X can reach any vertex of Bl_{i^*-1}, i.e., $X \subseteq S \setminus \texttt{GraphReach}(H_{i^*}[S], Bl_{i^*-1})$. As all edges in $H_{i^*-1}[S]$ are contained in $H_{i^*}[S]$, this implies $X \subseteq S \setminus \texttt{GraphReach}(H_{i^*-1}[S], Bl_{i^*-1})$.

Since `SmallestBSCC` finds the smallest bottom SCC in graph H_i for each i, the outer for-loop would thus have terminated in an iteration $i \leq i^* - 1$. Contradiction. □

Lemma 3.5 (Running time)**.** *Algorithm* GoodComp *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 3.3. 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 $O(m)$.

Consider the while loop where a set S is removed from Q. If $G[S]$ does not contain any edge after the removal of bad vertices, then S is not considered further by the algorithm. Otherwise $G[S]$ and $RevG[S]$ are search for bottom SCCs. The search in $G[S]$ and $RevG[S]$ only increases the running time by a factor of two, thus we restrict the analysis of the running time to $G[S]$. Let $n' \leq n$ be the number of vertices in S. The construction of $G_i[S]$, Z, and $G[X]$ can all be done in time $O(n' \cdot 2^i)$ for each i, i.e., in total time $O(n' \cdot 2^{i^*})$ up to level i^*. If $X = S$, then the algorithm terminates and the time for processing S can be bounded by $O(n' \cdot 2^{\log n'}) = O((n')^2)$. If the processing of S ends when some bottom SCC $G[X] \subseteq G[S]$ induced by some set of vertices X is found, let i^* be the value of i when $G[X]$ is detected and inserted into Q, and let c be some constant such that the time spent in this search for X is bounded by $c \cdot n' \cdot 2^{i^*-1}$. By Lemma 3.4 the set X contains at least 2^{i^*-1} vertices. Let $|X| = n_1$. The algorithm ensures $n_1 \leq n'/2$. We claim that the total running time for processing all sets S, except for the work in `Remove` and `Construct`, can be bounded by $f(n) = 2cn^2$. Whenever the algorithm does not terminate, we have by induction, and in particular for $n' = n$,

$$\begin{aligned} f(n') &\leq f(n_1) + f(n' - n_1) + cn'n_1 \,, \\ &\leq 2cn_1^2 + 2c(n' - n_1)^2 + cn'n_1 \,, \\ &= 2cn_1^2 + 2c(n')^2 - 4cn'n_1 + 2cn_1^2 + cn'n_1 \,, \\ &= 2c(n')^2 + 4cn_1^2 - 3cn'n_1 \,, \\ &\leq 2c(n')^2 \,, \end{aligned}$$

where the last inequality follows from $n_1 \leq n'/2$.

The operations `Remove` and `Construct` are called once per found bottom SCC $G[X]$ with $X \neq S$ and take by Lemma 3.3 $O(|X| + \mathit{bits}(X))$ time. By $n_1 \leq n'/2$ we have that whenever a vertex v is in X, the size of the set in Q containing v is halved; this can happen at most $\lceil \log n \rceil$ times. Hence, by charging $O(1)$ to the vertices in X and, respectively, to $\mathit{bits}(X)$, the total running time for this part can be bounded by $O((n + b) \log n)$, as each vertex and bit is only charged $O(\log n)$ times. Combining all parts yields the claimed running time bound of $O(n^2 + b \log n)$. □

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

Lemma 3.6. *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 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 implies 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 Lines 4–5 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 6 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 $G[X]$, induced by some set of vertices X. Since a good component is strongly connected, every good component in $G[S]$ either is a subgraph of the newly identified SCC $G[X]$ or does not contain *any* vertex of 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 correctly outputs a good component if and only if one exists.

Lemma 3.7 (Correctness)**.** *Algorithm* GoodComp *outputs a good component if one exists, otherwise the algorithm reports that no such component exists.*

Proof. First we show that whenever Algorithm GoodComp outputs a subgraph $G[S]$ induced by some set of vertices S, then $G[S]$ is a good component. Line 6 ensures only non-trivial SCSs are considered. After the removal of bad vertices from S in Lines 4–5, 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 component, and the only SCC in $G[S]$ is $G[S]$ itself. Only in this case $G[S]$ is output (in Line 15).

Algorithm GoodComp terminates if a good component is identified or Q is empty; in the latter case it reports that no good component exists. Lemma 3.6 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 and no good component was identified yet by the algorithm, then Q is not empty and thus the algorithm does not terminate until a good component is identified. Hence if the algorithm terminates because Q is empty, then no good component exists. By Lemma 3.5 the algorithm terminates after a finite number of steps.

Next we show that if there exists a good component in G, then the algorithm outputs 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. In Case (1) we are done, the argument for Case (2) is as follows. By Lemma 3.6 the component Y is never split by the algorithm thus after Case (2) happened at most n times, one of the following two cases occurs: either (a) $D(S_Y)$ 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 (b) $G[S_Y] = Y$ 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 6 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 15. □

Recall Algorithm Streett that calls Algorithm GoodComp for each SCC in the input graph and then computes reachability to the union of the identified good components. Lemmata 3.5 and 3.7 yield the following result.

Theorem 3.8. 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 and a good component reachable by x, a certificate for x can be output in time $O(m + n \min(n, k))$.

Remark 3.9 (Optimality). *We have shown that in a graph with k-pair Streett objectives the winning set and a certificate can be computed and output in time $O(n^2 + b \log n)$. Example 3.2 shows a lower bound of $\Omega(n \min(n, k))$ for outputting a certificate. Note that the size of the input is at least b. Hence the presented algorithm is optimal up to a log factor when $k = \Omega(n)$ and a certificate is required.*

4. Conclusion

In this work we have considered two classical algorithmic questions for parity and Streett objectives on graphs and game graphs.

We have presented an algorithm for parity games with n vertices and c priorities with a running time of roughly $O(n^{4/3+c/3})$, which improves the running time over previous results for game graphs with $\Omega(n^{4/3})$ edges when the number of priorities is sub-polynomial in n. In particular we improved the long standing running time for 3 priorities from $O(mn)$ to $O(n^{5/2})$.

For graphs with Streett objectives we have presented an $O(n^2 + b \log n)$-time algorithm (where b is the total number of elements in the Streett pairs), and a lower bound that shows that this running time is tight up to a log factor when a certificate has to be reported. The algorithm improves upon the known running time bounds when the number of edges m is at least of order $n^{4/3} \log^{-1/3} n + b^{2/3} \log^{1/3} n$ and the number of Streett pairs is at least of order n^2/m. This algorithm was extended to Markov Decision Processes (MDPs) in subsequent work [10, 45].

In particular in the light of the quasi-polynomial time algorithm for parity games in a breakthrough result subsequent to our work [7], showing new upper and (conditional) lower bounds for parity games remains a very interesting challenge. The techniques of [7] as well as the techniques presented here are inherently non-symbolic. An interesting open question is thus to find improved symbolic algorithms for these classical problems.

Acknowledgement

We would like to thank Tom Henzinger and the anonymous referees for their useful comments.

References

[1] R. Alur and T. A. Henzinger. Computer-aided verification. 2004.

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

[3] C. Beeri. On the membership problem for functional and multivalued dependencies in relational databases. *ACM Trans. Datab. Sys.*, 5(3):241–259, 1980.

[4] H. Björklund, S. Sandberg, and S. Vorobyov. A discrete subexponential algorithms for parity games. In *STACS*, pages 663–674, 2003.

[5] R. Bloem, K. Chatterjee, K. Greimel, T. A. Henzinger, and B. Jobstmann. Robustness in the presence of liveness. In *CAV*, pages 410–424, 2010.

[6] R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive(1) designs. *Journal of Computer and System Sciences*, 78(3):911–938, 2012.

[7] C. S. Calude, S. Jain, B. Khoussainov, W. Li, and F. Stephan. Deciding parity games in quasipolynomial time. In *STOC*, pages 252–263, 2017.

[8] K. Chatterjee, S. Chaubal, and P. Kamath. Faster algorithms for alternating refinement relations. In *Conference on Computer Science Logic (CSL)*, pages 167–182, 2012.

[9] K. Chatterjee, W. Dvořák, M. Henzinger, and V. Loitzenbauer. Conditionally Optimal Algorithms for Generalized Büchi Games. In *MFCS*, pages 25:1–25:15, 2016.

[10] K. Chatterjee, W. Dvořák, M. Henzinger, and V. Loitzenbauer. Model and objective separation with conditional lower bounds: Disjunction is harder than conjunction. In *LICS*, pages 197–206, 2016.

[11] K. Chatterjee, A. K. Goharshady, R. Ibsen-Jensen, and A. Pavlogiannis. Algorithms for algebraic path properties in concurrent systems of constant treewidth components. In *POPL*, pages 733–747, 2016.

[12] K. Chatterjee and M. Henzinger. Faster and Dynamic Algorithms For Maximal End-Component Decomposition And Related Graph Problems In Probabilistic Verification. In *SODA*, pages 1318–1336, 2011.

[13] 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. Announced at SODA'11 and SODA'12.

[14] K. Chatterjee, M. Henzinger, and V. Loitzenbauer. Improved Algorithms for One-Pair and k-Pair Streett Objectives. In *LICS*, pages 269–280, 2015.

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

[16] K. Chatterjee, M. Jurdziński, and T. A. Henzinger. Simple stochastic parity games. In *CSL*, pages 100–113, 2003.

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

[18] A. Church. Logic, arithmetic, and automata. In *ICM*, pages 23–35, 1962.

[19] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, September 2003.

[20] 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.

[21] 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.

[22] L. de Alfaro and T. A. Henzinger. Interface automata. In *International Symposium on Foundations of Software Engineering (FSE)*, pages 109–120, 2001.

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

[24] 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.

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

[26] E. A. Emerson and C. S. Jutla. The complexity of tree automata and logics of programs (extended abstract). In *FOCS*, pages 328–337, 1988.

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

[28] 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.

[29] K. Etessami, T. Wilke, and R. A. Schuller. Fair simulation relations, parity games, and state space reduction for büchi automata. *SIAM J. Comput.*, 34(5):1159–1175, 2005.
[30] J. Fearnley, S. Jain, S. Schewe, F. Stephan, and D. Wojtczak. An Ordered Approach to Solving Parity Games in Quasi Polynomial Time and Quasi Linear Space. In *SPIN*, pages 112–121, 2017.
[31] N. Francez. *Fairness*. Springer, New York, 1986.
[32] H. Gimbert and R. Ibsen-Jensen. A short proof of correctness of the quasi-polynomial time algorithm for parity games. 2017.
[33] Y. Godhal, K. Chatterjee, and T. A. Henzinger. Synthesis of AMBA AHB from formal specification: A case study. *Journal of Software Tools Technology Transfer*, 15(5-6):585–601, 2013.
[34] M. Henzinger, V. King, and T. Warnow. Constructing a Tree from Homeomorphic Subtrees, with Applications to Computational Evolutionary Biology. *Algorithmica*, 24(1):1–13, 1999. Announced at SODA'96.
[35] 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.
[36] T. A. Henzinger, O. Kupferman, and S. K. Rajamani. Fair simulation. *Information and Computation*, 173(1):64–81, 2002.
[37] N. Immerman. Number of quantifiers is better than number of tape cells. *Journal of Computer and System Sciences*, pages 384–406, 1981.
[38] M. Jurdziński. Deciding the winner in parity games is in UP $\cap$ co-UP. *Information Processing Letters*, 68(3):119–124, 1998.
[39] M. Jurdziński. Small Progress Measures for Solving Parity Games. In *STACS*, pages 290–301, 2000.
[40] M. Jurdziński and R. Lazić. Succinct progress measures for solving parity games. In *LICS*, pages 1–9, 2017.
[41] M. Jurdziński, M. Paterson, and U. Zwick. A Deterministic Subexponential Algorithm for Solving Parity Games. *SIAM J. Comput.*, 38(4):1519–1532, 2008.
[42] W. Kuijper and J. van de Pol. Computing weakest strategies for safety games of imperfect information. In *TACAS*, pages 92–106, 2009.
[43] T. Latvala and K. Heljanko. Coping With Strong Fairness. *Fundamenta Informaticae*, 43(1-4):175–193, 2000.
[44] O. Lichtenstein and A. Pnueli. Checking That Finite State Concurrent Programs Satisfy Their Linear Specification. In *POPL*, pages 97–107, 1985.
[45] V. Loitzenbauer. *Improved Algorithms and Conditional Lower Bounds for Problems in Formal Verification and Reactive Synthesis*. PhD thesis, 2016.
[46] R. McNaughton. Infinite games played on finite graphs. *Annals of Pure and Applied Logic*, 65(2):149–184, 1993.
[47] N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive(1) designs. In *VMCAI*, pages 364–380, 2006.
[48] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *POPL*, pages 179–190, 1989.
[49] P.J. Ramadge and W. Murray Wonham. Supervisory control of a class of discrete-event processes. *SIAM J. Control Optim.*, 25(1):206–230, 1987.
[50] S. Safra. On the complexity of ω-automata. In *FOCS*, pages 319–327, 1988.
[51] S. Safra. Exponential Determinization for omega-Automata with Strong-Fairness Acceptance Condition (Extended Abstract). In *STOC*, pages 275–282, 1992.
[52] S. Schewe. Solving Parity Games in Big Steps. In *FSTTCS*, pages 449–460, 2007.
[53] S. Schewe. Solving Parity Games in Big Steps. *Journal of Computer and Systems Science*, 84:243–262, 2017. Announced at FSTTCS'07.
[54] R. E. Tarjan. Depth First Search and Linear Graph Algorithms. *SIAM Journal of Computing*, 1(2):146–160, 1972.
[55] W. Thomas. Languages, automata, and logic. In *Handbook of Formal Languages: Volume 3 Beyond Words*, pages 389–455. Springer, 1997.
[56] J. Vöge and M. Jurdziński. A discrete strategy improvement algorithm for solving parity games. In *CAV*, pages 202–215, 2000.
[57] W. Zielonka. Infinite games on finitely coloured graphs with applications to automata on infinite trees. *Theoretical Computer Science*, 200(1–2):135–183, 1998.

This work is licensed under the Creative Commons Attribution-NoDerivs License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nd/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 9404, USA