Solving Parity Games Using An Automata-Based Algorithm

Conference Paper · July 2016

CITATIONS
0

READS
118

4 authors, including:

Antonio Di Stasio
University of Naples Federico II
4 PUBLICATIONS **2** CITATIONS
SEE PROFILE

Aniello Murano
University of Naples Federico II
139 PUBLICATIONS **750** CITATIONS
SEE PROFILE

Moshe Vardi
Rice University
658 PUBLICATIONS **32,137** CITATIONS
SEE PROFILE

Solving Parity Games Using An Automata-Based Algorithm*

Antonio Di Stasio[1], Aniello Murano[1], Giuseppe Perelli[2] **, Moshe Y. Vardi[3]

[1]Università di Napoli "Federico II", [2]University of Oxford [3]Rice University

Abstract. *Parity games* are abstract infinite-round games that take an important role in formal verification. In the basic setting, these games are two-player, turn-based, and played under perfect information on directed graphs, whose nodes are labeled with priorities. The winner of a play is determined according to the parities (even or odd) of the minimal priority occurring infinitely often in that play. The problem of finding a winning strategy in parity games is known to be in UPTime ∩ CoUPTime and deciding whether a polynomial time solution exists is a long-standing open question. In the last two decades, a variety of algorithms have been proposed. Many of them have been also implemented in a platform named `PGSolver`. This has enabled an empirical evaluation of these algorithms and a better understanding of their relative merits.
In this paper, we further contribute to this subject by implementing, for the first time, an algorithm based on alternating automata. More precisely, we consider an algorithm introduced by Kupferman and Vardi that solves a parity game by solving the emptiness problem of a corresponding alternating parity automaton. Our empirical evaluation demonstrates that this algorithm outperforms other algorithms when the game has a a small number of priorities relative to the size of the game. In many concrete applications, we do indeed end up with parity games where the number of priorities is relatively small. This makes the new algorithm quite useful in practice.

1 Introduction

Parity games [11,31] are abstract infinite-duration games that represent a powerful mathematical framework to address fundamental questions in computer science. They are intimately related to other infinite-round games, such as *mean* and *discounted* payoff, *stochastic*, and *multi-agent* games [3, 4, 6, 7].

In the basic setting, parity games are two-player, turn-based, played on directed graphs whose nodes are labeled with priorities (also called, *colors*) and players have perfect information about the adversary moves. The two players, Player 0 and Player 1, take turns moving a token along the edges of the graph

* Work supported by NSF grants CCF-1319459 and IIS-1527668, NSF Expeditions in Computing project "ExCAPE: Expeditions in Computer Augmented Program Engineering", BSF grant 9800096, ERC Advanced Investigator Grant 291528 ("Race") at Oxford and GNCS 2016: Logica, Automi e Giochi per Sistemi Auto-adattivi.

** Part of the work has been done while visiting Rice University.

starting from a designated initial node. Thus, a play induces an infinite path and Player 0 wins the play if the smallest priority visited infinitely often is even; otherwise, Player 1 wins the play. The problem of deciding if Player 1 has a winning strategy (i.e., can induce a winning play) in a given parity game is known to be in UPTIME ∩ COUPTIME [15]; whether a polynomial time solution exists is a long-standing open question [30].

Several algorithms for solving parity games have been proposed in the last two decades, aiming to tighten the known complexity bounds for the problem, as well as come out with solutions that work well in practice. Among the latter, we recall the recursive algorithm (`RE`) proposed by Zielonka [31], the Jurdziński's small-progress measures algorithm [16] (`SP`), the strategy-improvement algorithm by Jurdziński and Vöge [28], the (subexponential) algorithm by Jurdziński, Paterson and Zwick [17], and the big-step algorithm by Schewe [25]. These algorithms have been implemented in the platform `PGSolver`, and extensively investigated experimentally [12,13]. This study has also led to a few key optimizations, such as the decomposition into strongly connected components, the removal of self-cycles on nodes, and the application of a priority compression [2, 16]. Specifically, the latter allows to reduce a game to an equivalent game where the priorities are replaced in such a way they form a dense sequence of natural numbers, $1, 2, \ldots, d$, for a minimal possible d. Table 1 summarizes the mentioned algorithms along with their known worst-case complexity, where the parameters n, e, and d denote the number of nodes, edges, and priorities, respectively (see [12, 13], for more).

Algorithm	Computational Complexity
Recursive (RE) [31]	$O(e \cdot n^d)$
Small Progress Measures (SP) [16]	$O(d \cdot e \cdot (\frac{n}{d})^{\frac{d}{2}})$
Strategy Improvement (SI) [28]	$O(2^e \cdot n \cdot e)$
Dominion Decomposition (DD) [17]	$O(n^{\sqrt{n}})$
Big Step (BS) [25]	$O(e \cdot n^{\frac{1}{3}d})$

Table 1. Parity algorithms along with their computational complexities.

In formal system design [8,9,21,24], parity games arise as a natural evaluation machinery for the automatic synthesis and verification of distributed and reactive systems [1,19,27], as they allow to express liveness and safety properties in a very elegant and powerful way [22]. Specifically, in model-checking, one can check the correctness of a system with respect to a desired behavior, by checking whether a model of the system, that is, a *Kripke structure*, is correct with respect to a formal specification of its behavior, usually described in terms of a modal logic formula. In case the specification is given as a μ-calculus formula [18], the model checking question can be rephrased, in linear-time, as a parity game [11]. So, a parity game solver can be used as a model checker for a μ-calculus specification (and vice-versa), as well as for fragments such as CTL, CTL*, and the like.

In the automata-theoretic approach to μ-calculus model checking, under a linear-time translation, one can also reduce the verification problem to a question about automata. More precisely, one can take the product of the model and an alternating tree automaton accepting all tree models of the specification. This

product can be defined as an alternating word parity automaton over a singleton alphabet, and the system is correct with respect to the specification iff this automaton is nonempty [21]. It has been proved there that the nonemptiness problems for nondeterministic tree parity automata and alternating word parity automata over a singleton alphabet are equivalent and that their complexities coincide. For this reason, in the sequel we refer to these two kinds of automata just as parity automata. Hence, algorithms for the solution of the μ-calculus model checking problem, parity games, and the emptiness problem for parity automata can be interchangeably used to solve any of these problems, as they are linear-time equivalent. Several algorithms have been proposed in the literature to solve the non-emptiness problem of parity automata, but none of them has been ever implemented under the purpose of solving parity games.

In this paper, we study and implement an algorithm, which we call `APT`, introduced by Kupferman and Vardi in [20], for solving parity games via emptiness checking of alternating parity automata, and evaluate its performance over the `PGSolver` platform. This algorithm has been sketched in [20], but not spelled out in detail and without a correctness proof, two major gaps that we fill here. The core idea of the `APT` algorithm is an efficient translation to *weak alternating automata* [23]. These are a special case of Büchi automata in which the set of states is partitioned into partially ordered sets. Each set is classified as accepting or rejecting. The transition function is restricted so that the automaton either stays at the same set or moves to a smaller set in the partial order. Thus, each run of a weak automaton eventually gets trapped in some set in the partition. The special structure of weak automata is reflected in their attractive computational properties. In particular, the nonemptiness problem for weak automata can be solved in linear time [21], while the best known upper bound for the nonemptiness problem for Büchi automata is quadratic [5]. Given an alternating parity word automaton with n states and d colors, the `APT` algorithm checks the emptiness of an equivalent weak alternating word automaton with $\mathrm{O}(n^d)$ states. The construction goes through a sequence of d intermediate automata. Each automaton in the sequence refines the state space of its predecessor and has one less color to check in its parity condition. Since one can check in linear time the emptiness of such an automaton, we get an $O(n^d)$ overall complexity for the addressed problem. `APT` does not construct the equivalent weak automaton directly, but applies the emptiness test directly, constructing the equivalent weak automaton on the fly.

We evaluated our implementation of the `APT` algorithm over several random game instances, comparing it with `RE` and `SP` algorithms. Our main finding is that when the number of the priority in a game is significantly smaller (specifically, logarithmically) than the number of nodes in the game graph, the `APT` algorithm significantly outperform the other algorithms. We take this as an important development since in many real applications of parity games we do get game instances where the number of priorities is indeed very small compared to the size of the game graph. For example, coming back to the automata-theoretic approach to μ-calculus model checking [21], the translation usually results in a parity automaton (and thus in a parity game) with few priorities, but with a huge

number of nodes. This is due to the fact that usually specification formulas are small, while the system is big. A similar phenomenon occurs in the application of parity games to reactive synthesis [27].

Outline The sequel of the paper is as follows. Section 2 gives preliminary concepts on parity games. Section 3 introduces extended parity games and describes the `APT` algorithm in detail, including a proof of correctness. Section 4 describes the implementation of the `APT` algorithm in the tool `PGSolver`. Section 5 contains the experimental results on runtime for `APT` over random benchmarks. Finally, Section 6 gives some conclusions.

2 Preliminaries

In this section, we briefly recall some basic concepts regarding parity games. A *Parity Game* (PG, for short) is a tuple $\mathcal{G} \triangleq \langle \mathrm{Ps}_0, \mathrm{Ps}_1, Mv, \mathsf{p} \rangle$, where Ps_0 and Ps_1 are two finite disjoint sets of nodes for Player 0 and Player 1, respectively, with $\mathrm{Ps} = \mathrm{Ps}_0 \cup \mathrm{Ps}_1$, $Mv \subseteq \mathrm{Ps} \times \mathrm{Ps}$, is the left-total binary relation of moves, and $\mathsf{p} : \mathrm{Ps} \to \mathbb{N}$ is the priority function [1]. Each player moves a token along nodes by means of the relation Mv. By $Mv(q) \triangleq \{q' \in \mathrm{Ps} : (q, q') \in Mv\}$ we denote the set of nodes to which the token can be moved, starting from node q.

As a running example, consider the PG depicted in Figure 1. The set of players's nodes is $\mathrm{Ps}_0 = \{\mathsf{q}_0, \mathsf{q}_3, \mathsf{q}_4, \mathsf{q}_5\}$ and $\mathrm{Ps}_1 = \{\mathsf{q}_1, \mathsf{q}_2, \mathsf{q}_6\}$; we use circles to denote nodes belonging to Player 0 and squares for those belonging to Player 1. Mv is described by arrows. Finally, the priority function p is given by $\mathsf{p}(\mathsf{q}_1) = 1$, $\mathsf{p}(\mathsf{q}_3) = \mathsf{p}(\mathsf{q}_4) = \mathsf{p}(\mathsf{q}_6) = 2$, $\mathsf{p}(\mathsf{q}_0) = 3$, and $\mathsf{p}(\mathsf{q}_2) = \mathsf{p}(\mathsf{q}_5) = 5$.

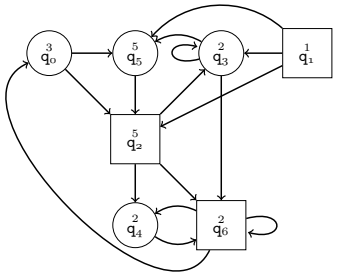

Fig. 1. A parity game.

A *play* (*resp., history*) over $\mathcal{G}$ is an infinite (*resp.,* finite) sequence $\pi = q_1 \cdot q_2 \cdot \ldots \in \mathrm{Pth} \subseteq \mathrm{Ps}^\omega$ (*resp.,* $\rho = q_1 \cdot \ldots \cdot q_n \in \mathrm{Hst} \subseteq \mathrm{Ps}^*$) of nodes that agree with Mv, *i.e.*, $(\pi_i, \pi_{i+1}) \in Mv$, for each natural number $i \in \mathbb{N}$ (*resp.,* $i \in [1, n-1]$). In the PG in Figure 1, a possible play is $\overline{\pi} = \mathsf{q}_1 \cdot \mathsf{q}_5 \cdot \mathsf{q}_2 \cdot (\mathsf{q}_3)^\omega$, while a possible history is given by $\overline{\rho} = \mathsf{q}_1 \cdot \mathsf{q}_5 \cdot \mathsf{q}_2 \cdot \mathsf{q}_3$.

For a given play $\pi = q_1 \cdot q_2 \cdot \ldots$, by $\mathsf{p}(\pi) = \mathsf{p}(q_1) \cdot \mathsf{p}(q_2) \cdot \ldots \in \mathbb{N}^\omega$ we denote the associated priority sequence. As an example, the associated priority sequence to $\overline{\pi}$ is given by $\mathsf{p}(\overline{\pi}) = 1 \cdot 5 \cdot 5 \cdot (2)^\omega$.

For a given history $\rho = q_1 \cdot \ldots \cdot q_n$, by $\mathsf{fst}(\rho) \triangleq q_1$ and $\mathsf{lst}(\rho) \triangleq q_n$ we denote the first and last node occurring in ρ, respectively. For the example history, we have that $\mathsf{fst}(\overline{\rho}) = \mathsf{q}_1$ and $\mathsf{lst}(\overline{\rho}) = \mathsf{q}_3$. By Hst_0 (*resp.,* Hst_1) we denote the set of histories ρ such that $\mathsf{lst}(\rho) \in \mathrm{Ps}_0$ (*resp.,* $\mathsf{lst}(\rho) \in \mathrm{Ps}_1$). Moreover, by $\mathsf{Inf}(\pi)$ and $\mathsf{Inf}(\mathsf{p}(\pi))$ we denote the set of nodes and priorities that occur infinitely often in π and $\mathsf{p}(\pi)$, respectively. Finally, a play π is winning for Player 0 (resp., Player 1) if $\min(\mathsf{Inf}(\mathsf{p}(\pi)))$ is even (*resp.,* odd). In the running example, we have that $\mathsf{Inf}(\overline{\pi}) = \{\mathsf{q}_3\}$ and $\mathsf{Inf}(\mathsf{p}(\overline{\pi})) = \{2\}$ and so, π is winning for Player 0.

[1] Here, we mean the set of non-negative integers, excluding zero.

A Player 0 (*resp.,* Player 1) strategy is a function $\mathsf{str}_0 : \mathrm{Hst}_0 \to \mathrm{Ps}$ (*resp.,* $\mathsf{str}_1 : \mathrm{Hst}_1 \to \mathrm{Ps}$) such that, for all $\rho \in \mathrm{Hst}_0$ (*resp.,* $\rho \in \mathrm{Hst}_1$), it holds that $(\mathsf{lst}(\rho), \mathsf{str}_0(\rho)) \in Mv$ (*resp.,* $\mathsf{lst}(\rho), \mathsf{str}_1(\rho)) \in Mv$).

Given a node q, Player 0 and a Player 1 strategies str_0 and str_1, the play of these two strategies, denoted by $\mathsf{play}(q, \mathsf{str}_0, \mathsf{str}_1)$, is the only play π in the game that starts in q and agrees with both Player 0 and Player 1 strategies, *i.e.*, for all $i \in \mathbb{N}$, if $\pi_i \in \mathrm{Ps}_0$, then $\pi_{i+1} = \mathsf{str}_0(\pi_i)$, and $\pi_{i+1} = \mathsf{str}_1(\pi_i)$, otherwise.

A strategy str_0 (*resp.,* str_1) is *memoryless* if, for all $\rho_1, \rho_2 \in \mathrm{Hst}_0$ (*resp.,* $\rho_1, \rho_2 \in \mathrm{Hst}_1$), with $\mathsf{lst}(\rho_1) = \mathsf{lst}(\rho_2)$, it holds that $\mathsf{str}_0(\rho_1) = \mathsf{str}_0(\rho_2)$ (*resp.,* $\mathsf{str}_0(\rho_1) = \mathsf{str}_1(\rho_2)$). Note that a memoryless strategy can be defined on the set of nodes, instead of the set of histories. Thus we have that they are of the form $\mathsf{str}_0 : \mathrm{Ps}_0 \to \mathrm{Ps}$ and $\mathsf{str}_1 : \mathrm{Ps}_1 \to \mathrm{Ps}$.

We say that Player 0 (*resp.,* Player 1) *wins* the game $\mathcal{G}$ from node q if there exists a Player 0 (*resp.,* Player 1) strategy str_0 (*resp.,* str_1) such that, for all Player 1 (*resp.,* Player 0) strategies str_1 (*resp.,* str_0) it holds that $\mathsf{play}(q, \mathsf{str}_0, \mathsf{str}_1)$ is winning for Player 0 (*resp.,* Player 1).

A node q is *winning* for Player 0 (*resp.,* Player 1) if Player 0 (*resp.,* Player 1) wins the game from q. By $\mathrm{Win}_0(\mathcal{G})$ (*resp.,* $\mathrm{Win}_1(\mathcal{G})$) we denote the set of winning nodes in $\mathcal{G}$ for Player 0 (*resp.,* Player 1). Parity games enjoy determinacy, meaning that, for every node q, either $q \in \mathrm{Win}_0(\mathcal{G})$ or $q \in \mathrm{Win}_1(\mathcal{G})$ [11]. Moreover, it can be proved that, if Player 0 (resp., Player 1) has a winning strategy from node q, then it has a memoryless winning strategy from the same node [31].

3 Extended Parity Games

In this section we recall the `APT` algorithm, introduced by Kupferman and Vardi in [20], to solve parity games via emptiness checking of parity automata. More important, we fill two major gaps from [20] which is to spell out in details the definition of the `APT` algorithm as well as to give a correctness proof. The `APT` algorithm makes use of two special (incomparable) sets of nodes, denoted by V and A, and called set of *Visiting* and *Avoiding*, respectively. Intuitively, a node is declared visiting for a player at the stage in which it is clear that, by reaching that node, he can surely induce a winning play and thus winning the game. Conversely, a node is declared avoiding for a player whenever it is clear that, by reaching that node, he is not able to induce any winning play and thus losing the game. The algorithm, in turns, tries to partition all nodes of the game into these two sets. The formal definition of the sets V and A follows.

An *Extended Parity Game*, (EPG, for short) is a tuple $\langle \mathrm{Ps}_0, \mathrm{Ps}_1, \mathrm{V}, \mathrm{A}, Mv, \mathsf{p}\rangle$ where Ps_0, Ps_1, Mv are as in PG. The subsets of nodes $\mathrm{V}, \mathrm{A} \subseteq \mathrm{Ps} = \mathrm{Ps}_0 \cup \mathrm{Ps}_1$ are two disjoint sets of *Visiting* and *Avoiding* nodes, respectively. Finally, $\mathsf{p} : \mathrm{Ps} \to \mathbb{N}$ is a parity function mapping every non-visiting and non-avoiding set to a color.

The notions of histories and plays are equivalent to the ones given for PG. Moreover, as far as the definition of strategies is concerned, we say that a play π that is in $\mathrm{Ps} \cdot (\mathrm{Ps} \setminus (\mathrm{V} \cup \mathrm{A}))^* \cdot \mathrm{V} \cdot \mathrm{Ps}^\omega$ is winning for Player 0, while a play π that is in $\mathrm{Ps} \cdot (\mathrm{Ps} \setminus (\mathrm{V} \cup \mathrm{A}))^* \cdot \mathrm{A} \cdot \mathrm{Ps}^\omega$ is winning for Player 1. For a play π that

never hits either V or A, we say that it is winning for Player 0 iff it satisfies the parity condition, *i.e.*, $\min(\mathsf{Inf}(\mathsf{p}(\pi)))$ is even, otherwise it is winning for Player 1.

Clearly, PGs are special cases of EPGs in which $\mathrm{V} = \mathrm{A} = \emptyset$. Conversely, one can transform an EPG into an equivalent PG with the same winning set by simply replacing every outgoing edge with loop to every node in $\mathrm{V} \cup \mathrm{A}$ and then relabeling each node in V and A with an even and an odd number, respectively.

In order to describe how to solve EPGs, we introduce some notation. By $\mathrm{F}_i = \mathsf{p}^{-1}(i)$ we denote the set of all nodes labeled with i. Doing that, the parity condition can be described as a finite sequence $\alpha = \mathrm{F}_1 \cdot \ldots \cdot \mathrm{F}_k$ of sets, which alternates from sets of nodes with even priorities to sets of nodes with odd priorities and the other way round, forming a partition of the set of nodes, ordered by the priority assigned by the parity function. We call the set of nodes F_i an even (*resp.,* odd) parity set if i is even (*resp.,* odd).

For a given set $\mathrm{X} \subseteq \mathrm{Ps}$, by $\mathsf{force}_0(\mathrm{X}) = \{q \in \mathrm{Ps}_0 : \mathrm{X} \cap Mv(q) \neq \emptyset\} \cup \{q \in \mathrm{Ps}_1 : \mathrm{X} \subseteq Mv(q)\}$ we denote the set of nodes from which Player 0 can force to move in the set X. Analogously, by $\mathsf{force}_1(\mathrm{X}) = \{q \in \mathrm{Ps}_1 : \mathrm{X} \cap Mv(q) \neq \emptyset\} \cup \{q \in \mathrm{Ps}_0 : \mathrm{X} \subseteq Mv(q)\}$ we denote the set of nodes from which Player 1 can force to move in the set X. For example, in the PG in Figure 1, $\mathsf{force}_1(\{\mathsf{q}_6\}) = \{\mathsf{q}_2, \mathsf{q}_4, \mathsf{q}_6\}$.

We now introduce two functions that are co-inductively defined that will be used to compute the winning sets of Player 0 and Player 1, respectively.

For a given EPG $\mathcal{G}$ with α being the representation of its parity condition, V its visiting set, and A its avoiding set, we define the functions $\mathrm{Win}_0(\alpha, \mathrm{V}, \mathrm{A})$ and $\mathrm{Win}_1(\alpha, \mathrm{A}, \mathrm{V})$. Informally, $\mathrm{Win}_0(\alpha, \mathrm{V}, \mathrm{A})$ computes the set of nodes from which the player 0 has a strategy that avoids A and either force a visit in V or he wins the parity condition. The definition is symmetric for the function $\mathrm{Win}_1(\alpha, \mathrm{A}, \mathrm{V})$. Formally, we define $\mathrm{Win}_0(\alpha, \mathrm{V}, \mathrm{A})$ and $\mathrm{Win}_1(\alpha, \mathrm{A}, \mathrm{V})$ as follows.

If $\alpha = \varepsilon$ is the empty sequence, then

- $\mathrm{Win}_0(\varepsilon, \mathrm{V}, \mathrm{A}) = \mathsf{force}_0(\mathrm{V})$ and
- $\mathrm{Win}_1(\varepsilon, \mathrm{A}, \mathrm{V}) = \mathsf{force}_1(\mathrm{A})$.

Otherwise, if $\alpha = \mathrm{F} \cdot \alpha'$, for some set F, then

- $\mathrm{Win}_0(\mathrm{F} \cdot \alpha', \mathrm{V}, \mathrm{A}) = \mathrm{Ps} \setminus \mu\mathrm{Y}(\mathrm{Win}_1(\alpha', \mathrm{A} \cup (\mathrm{F} \setminus \mathrm{Y}), \mathrm{V} \cup (\mathrm{F} \cap \mathrm{Y})))$ and
- $\mathrm{Win}_1(\mathrm{F} \cdot \alpha', \mathrm{A}, \mathrm{V}) = \mathrm{Ps} \setminus \mu\mathrm{Y}(\mathrm{Win}_0(\alpha', \mathrm{V} \cup (\mathrm{F} \setminus \mathrm{Y}), \mathrm{A} \cup (\mathrm{F} \cap \mathrm{Y})))$,

where μ is the least fixed-point operator[2].

To better understand how `APT` solves a parity game we show a simple piece of execution on the example in Fig 1. It is easy to see that such parity game is won by Player 0 in all the possible starting nodes. Then, the fixpoint returns the entire set Ps. The parity condition is given by $\alpha = \mathrm{F}_1 \cdot \mathrm{F}_2 \cdot \mathrm{F}_3 \cdot \mathrm{F}_4 \cdot \mathrm{F}_5$, where $\mathrm{F}_1 = \{\mathsf{q}_1\}$, $\mathrm{F}_2 = \{\mathsf{q}_3, \mathsf{q}_4, \mathsf{q}_6\}$, $\mathrm{F}_3 = \{\mathsf{q}_0\}$, $\mathrm{F}_4 = \emptyset$, $\mathrm{F}_5 = \{\mathsf{q}_5, \mathsf{q}_6\}$. The repeated application of functions $\mathrm{Win}_0(\alpha, \mathrm{V}, \mathrm{A})$ and $\mathrm{Win}_1(\alpha, \mathrm{A}, \mathrm{V})$ returns:

[2] The unravellings of Win_0 and Win_1 have some anologies with the fixed-point formula introduced in [29] also used to solve parity games. Unlike our work, however, the formula presented there is just a translation of the Zielonka algorithm [31].

$$\mathrm{Win}_{\mathrm{o}}(\alpha, \emptyset, \emptyset) = \mathrm{Ps} \setminus \mu \mathrm{Y}^1(\mathrm{Ps} \setminus \mu \mathrm{Y}^2(\mathrm{Ps} \setminus \mu \mathrm{Y}^3(\mathrm{Ps} \setminus \mu \mathrm{Y}^4(\mathrm{Ps} \setminus \mu \mathrm{Y}^5(\mathrm{Ps} \setminus \mathsf{force}_1(\mathrm{V}^6))$$

in which the sets Y^i are the nested fixpoint of the formula, while the set V^6 is obtained by recursively applying the following:

- $\mathrm{V}^1 = \emptyset$, $\mathrm{V}^{i+1} = \mathrm{A}^i \cup (\mathrm{F}_i \setminus \mathrm{Y}^i)$, and
- $\mathrm{A}^1 = \emptyset$, $\mathrm{A}^{i+1} = \mathrm{V}^i \cup (\mathrm{F}_i \cap \mathrm{Y}^i)$.

As a first step of the fixpoint computation, we have that $\mathrm{Y}^1 = \mathrm{Y}^2 = \mathrm{Y}^3 = \mathrm{Y}^4 = \mathrm{Y}^5 = \emptyset$. Then, by following the two iterations above for the example in Figure 1, we obtain that $\mathrm{V}^6 = \{\mathsf{q}_0, \mathsf{q}_1, \mathsf{q}_2, \mathsf{q}_5\}$.

At this point we have that $\mathsf{force}_1(\mathrm{V}^6) = \{\mathsf{q}_0, \mathsf{q}_1, \mathsf{q}_5, \mathsf{q}_6\} \neq \emptyset = \mathrm{Y}^5$. This means that the fixpoint for Y^5 has not been reached yet. Then, we update the set Y^5 with the new value and compute again V^6. This procedure is repeated up to the point in which $\mathsf{force}_1(\mathrm{V}^6) = \mathrm{Y}^5$, which means that the fixpoint for Y^5 has been reached. Then we iteratively proceed to compute $\mathrm{Y}^4 = \mathrm{Ps} \setminus \mathrm{Y}^5$ until a fixpoint for Y^4 is reached. Note that the sets A^i and V^i depends on the Y^i and so they need to be updated step by step. As soon as a fixpoint for Y_1 is reached, the algorithm returns the set $\mathrm{Ps} \setminus \mathrm{Y}_1$. As a fundamental observation, note that, due to the fact that the fixpoint operations are nested one to the next, updating the value of Y^i implies that every Y^j, with $j > i$, needs to be reset to the empty set.

We now prove the correctness of this procedure. Note that the algorithm is an adaptation of the one provided by Kupferman and Vardi in [20], for which a proof of correctness has never been shown.

Theorem 1. *Let* $\mathcal{G} = \langle \mathrm{Ps}_0, \mathrm{Ps}_1, \mathrm{V}, \mathrm{A}, Mv, \mathsf{p} \rangle$ *be an* EPG *with* α *being the parity sequence condition. Then, the following properties hold.*

1. *If* $\alpha = \varepsilon$ *then* $\mathrm{Win}_{\mathrm{o}}(\mathcal{G}) = \mathrm{Win}_{\mathrm{o}}(\alpha, \mathrm{V}, \mathrm{A})$ *and* $\mathrm{Win}_1(\mathcal{G}) = \mathrm{Win}_1(\alpha, \mathrm{V}, \mathrm{A})$*;*
2. *If* α *starts with an odd parity set, it holds that* $\mathrm{Win}_{\mathrm{o}}(\mathcal{G}) = \mathrm{Win}_{\mathrm{o}}(\alpha, \mathrm{V}, \mathrm{A})$*;*
3. *If* α *starts with an even parity set, it holds that* $\mathrm{Win}_1(\mathcal{G}) = \mathrm{Win}_1(\alpha, \mathrm{V}, \mathrm{A})$.

Proof. The proof of Item 1 follows immediately by definition, as $\alpha = \epsilon$ forces the two players to reach their respective winning sets in one step.

For Item 2 and 3, we need to find a partition of F into a winning set for Player 0 and a winning set for Player 1 such that the game is invariant *w.r.t.* the winning sets, once they are moved to visiting and avoiding, respectively. We proceed by mutual induction on the length of the sequence α. As base case, assume $\alpha = \mathrm{F}$ and F to be an odd parity set. Then, first observe that Player 0 can win only by eventually hitting the set V, as the parity condition is made by only odd numbers. We have that $\mathrm{Win}_{\mathrm{o}}(\mathrm{F}, \mathrm{V}, \mathrm{A}) = \mu \mathrm{Y}(\mathrm{Ps} \setminus \mathrm{Win}_1(\varepsilon, \mathrm{A} \cup (\mathrm{F} \setminus \mathrm{Y}), \mathrm{V} \cup (\mathrm{F} \cap (\mathrm{Y})))) = \mu \mathrm{Y}(\mathrm{Ps} \setminus \mathsf{force}_1(\mathrm{A} \cup (\mathrm{F} \setminus \mathrm{Y})))$ that, by definition, computes the set from which Player 1 cannot avoid a visit to V, hence the winning set for Player 0. In the case the set F is an even parity set the reasoning is symmetric.

As an inductive step, assume that Items 2 and 3 hold for sequences α of length n, we prove that it holds also for sequences of the form $\mathrm{F} \cdot \alpha$ of length $n+1$. Suppose

that F is a set of odd priority. Then, we have that, by induction hypothesis, the formula $\mathrm{Win}_1(\alpha, \mathrm{A} \cup (\mathrm{F} \setminus \mathrm{Y}), \mathrm{V} \cup (\mathrm{F} \cap \mathrm{Y}))$ computes the winning set for Player 1 for the game in which the nodes in $\mathrm{F} \cap \mathrm{Y}$ are visiting, while the nodes in $\mathrm{F} \setminus \mathrm{Y}$ are avoiding. Thus, its complement $\mathrm{Ps} \setminus \mathrm{Win}_1(\alpha, \mathrm{A} \cup (\mathrm{F} \setminus \mathrm{Y}), \mathrm{V} \cup (\mathrm{F} \cap \mathrm{Y}))$ returns the winning set for Player 0 in the same game. Now, observe that, if a set Y' is bigger than Y, then $\mathrm{Ps} \setminus \mathrm{Win}_1(\alpha, \mathrm{A} \cup (\mathrm{F} \setminus \mathrm{Y}'), \mathrm{V} \cup (\mathrm{F} \cap \mathrm{Y}'))$ is the winning set for Player 0 in which some node in $\mathrm{F} \setminus \mathrm{Y}$ has been moved from avoiding to visiting. Thus we have that $\mathrm{Ps} \setminus \mathrm{Win}_1(\alpha, \mathrm{A} \cup (\mathrm{F} \setminus \mathrm{Y}), \mathrm{V} \cup (\mathrm{F} \cap \mathrm{Y})) \subseteq \mathrm{Ps} \setminus \mathrm{Win}_1(\alpha, \mathrm{A} \cup (\mathrm{F} \setminus \mathrm{Y}'), \mathrm{V} \cup (\mathrm{F} \cap \mathrm{Y}'))$. Moreover, observe that, if a node $q \in \mathrm{F} \cup \mathrm{A}$ is winning for Player 0, then it can be avoided in all possible winning plays, and so it is winning also in the case q is only in F. It is not hard to see that, after the last iteration of the fixpoint operator, the two sets $\mathrm{F} \setminus \mathrm{Y}$ and $\mathrm{F} \cap \mathrm{Y}$ can be considered in avoiding and winning, respectively, in a way that the winning sets of the game are invariant under this update, which concludes the proof of Item 2.

Also in the inductive case, the reasoning for Item 3 is perfectly symmetric to the one for Item 2. □

4 Implementation of `APT` in `PGSolver`

In this section we describe the implementation of `APT` in the well-known platform `PGSolver` developed in OCaml by Friedman and Lange [13], which collects the large majority of the algorithms introduced in the literature to solve parity games [14, 16, 17, 25, 26, 28, 31].

We briefly recall the main aspects of this platform. The graph data structure is represented as a fixed length array of tuples. Every tuple has all information that a node needs, such as the owner player, the assigned priority and the adjacency list of nodes. The platform implements a collection of tools to generate and solve parity games, as well as compare the performance of different algorithms. The purpose of this platform is not just that of making available an environment to deploy and test a generic solution algorithm, but also to investigate the practical aspects of the different algorithms on the different classes of parity games. Moreover, `PGSolver` implements optimizations that can be applied to all algorithms in order to improve

```
fun win i G Ps α V A =
 if(α ≠ ϵ) then
  W := Ps \(min_fp (1−i) G Ps α V A);
 else
  W :=force_i(V);

 return W;;

fun min_fp i G Ps α V A =
 Y1 := ∅;
 Y2 := ∅;
 F := head[α];
 α' := tail[α];

 A' := A ∪ (F ∩ Y1);
 V' := V ∪ (F ∩ Y1);

 Y2 := win i G Ps α' A' V' ;

 while( Y2 ≠ Y1) do
 (
  Y1 := Y2 ;

  A' := A ∪ (F ∩ Y1);
  V' := V \ (F ∩ Y1);

  Y2 := win i G Ps α' A' V' ;
 )
 done

 return Y2;;
```

Fig. 2. APT Algorithm

their performance. The most useful optimizations in practice are decomposition into strongly connected components, removal of self-cycles on nodes, and priority compression.

We have added to `PGSolver` an implementation of the `APT` algorithm introduced in Section 3. Our procedure applies the fixpoint algorithm to compute the set of winning positions in the game by means of two principal functions that implement the two functions of the algorithm core processes, i.e., function force_i and the recursive function $\mathrm{Win}_i(\alpha, V, A)$. The pseudocode of the `APT` algorithm implementation is reported in Figure 2. It takes six parameters: the Player (0 or 1), the game, the set of nodes, the condition α, the set of visiting and avoiding. Moreover, we define the function *min_fp* for the calculation of the fixed point. The whole procedure makes use of Set and List data structures, which are available in the OCaml's standard library, for the manipulation of the sets visiting and avoiding, and the accepting condition α. The tool along with the implementation of the `APT` algorithm is available for download from `https://github.com/antoniodistasio/pgsolver-APT`.

For the sake of clarity, we report that in `PGSolver` it is used the maximal priority to decide who wins a given parity game. Conversely, the `APT` algorithm uses the minimal priority. However, these two conditions are well known to be equivalent and, in order to compare instances of the same game on different implementations of parity games algorithms in `PGSolver`, we simply convert the game to the specific algorithm accordingly. For the conversion, we simply use a suitable permutation of the priorities.

5 Experiments

In this section, we report the experimental results on evaluating the performance for the `APT` algorithm implemented in `PGSolver` over the random benchmarks generated in the platform. We have compared the performance of the implementation of `APT` with those of `RE` and `SP`. We have chosen these two algorithms as they have been proved to be the best-performing in practice [13].

All tests have been run on an AMD Opteron 6308 @2.40GHz, with 224GB of RAM and 128GB of swap running Ubuntu 14.04. We note that `APT` has been executed without applying any optimization implemented in `PGSolver` [13], while `SP` and `RE` are run with such optimizations. Applying these optimization on `APT` is a topic of further research.

We evaluated the performance of the three algorithms over a set of games that are randomly generated by `PGSolver`, in which it is possible to give the number n of states and the number k of priority as parameters. We have taken 20 different game instances for each set of parameters and used the average time among them returned by the tool. For each game, the generator works as follows. For each node q in the graph-game, the priority $\mathsf{p}(q)$ is chosen uniformly between 0 and $k-1$, while its ownership is assigned to Player 0 with probability $\frac{1}{2}$, and to Player 1 with probability $\frac{1}{2}$. Then, for each node q, a number d from 1 to n is chosen uniformly and d distinct successors of q are randomly selected.

5.1 Experimental results

n	2 Pr			3 Pr			5 Pr		
	RE	SP	APT	RE	SP	APT	RE	SP	APT
2000	4.94	5.05	0.10	4.85	5.20	0.15	4.47	4.75	0.42
4000	31.91	32.92	0.17	31.63	31.74	0.22	31.13	32.02	0.82
6000	107.06	108.67	0.29	100.61	102.87	0.35	100.81	101.04	1.39
8000	229.70	239.83	0.44	242.24	253.16	0.5	228.48	245.24	2.73
10000	429.24	443.42	0.61	482.27	501.20	0.85	449.26	464.36	3.61
12000	772.60	773.76	0.87	797.07	808.96	0.98	762.89	782.53	6.81
14000	1185.81	1242.56	1.09	1227.34	1245.39	1.15	1256.32	1292.80	10.02

Table 2. Runtime executions with fixed priorities 2, 3 and 5

We ran two experiments. First, we tested games with 2, 3, and 5 priorities, where for each of them we measured runtime performance for different state-space sizes, ranging in $\{2000, 4000, 6000, 8000, 10000, 12000, 14000\}$. The results are in Table 2, in which the number of states is reported in column 1, the number of colors is reported in the macro-column 2, 3, and 5, each of them containing the runtime executions, expressed in seconds, for the three algorithms. Second, we evaluated the algorithms on games with an exponential number of nodes *w.r.t.* the number of priorities. More precisely, we ran experiments for $n = 2^k$, $n = e^k$ and $n = 10^k$, where n is the number of states and k is the number of priorities.

The experiment results are reported in Table 3. By $\texttt{abort}_T$, we denote that the execution has been aborted due to time-out (greater of one hour), while by $\texttt{abort}_M$we denote that the execution has been aborted due to mem-out.

The first experiment shows that with a fixed number of priorities (2, 3, and 5) APT significantly outperforms the other algorithms, showing excellent runtime execution even on fairly large instances. For example, for $n = 14000$, the running time for both RE and SP is about 20 minutes, while for APT it is less than a minute.

The results of the exponential-scaling experiments, shown in Table 3, give more nuanced results. Here, APT is the best performing algorithm for $n = e^k$ and $n = 10^k$. For example, when $n = 100000$ and $k = 5$, both RE and SP memout, while APT completes in just over one minute. That is, the efficiency of APT is notable also in terms of memory usage. At the same APT underperforms for $n = 2^k$. Our conclusion is that APT has superior performance when the number of priorities is logarithmic in the number of game-graph nodes, but the base of the logarithm has to be large enough. As we see experimentally, e is sufficiently large base, but 2 is not. This point deserve

n	Pr	RE	SP	APT
$n = 2^k$				
1024	10	1.25	1.25	8.58
2048	11	7.90	8.21	71.08
4096	12	52.29	52.32	1505.75
8192	13	359.29	372.16	$\texttt{abort}_T$
16384	14	2605.04	2609.29	$\texttt{abort}_T$
32768	15	$\texttt{abort}_T$	$\texttt{abort}_T$	$\texttt{abort}_T$
$n = e^k$				
21	3	0	0	0
55	4	0	0	0.02
149	5	0.01	0.01	0.08
404	6	0.14	0.14	0.19
1097	7	1.72	1.72	0.62
2981	8	24.71	24.46	7.88
8104	9	413.2.34	414.65	35.78
22027	10	$\texttt{abort}_T$	$\texttt{abort}_T$	311.87
$n = 10^k$				
10	1	0	0	0
100	2	0	0	0
1000	3	1.3	1.3	0.04
10000	4	738.86	718.24	4.91
100000	5	$\texttt{abort}_M$	$\texttt{abort}_M$	66.4

Table 3. Runtime executions with $n = e^k$ and $n = 2^k$ and $n = 10^k$

further study, which we leave to future work. In Figure 3 we just report graphically the benchmarks in the case $n = e^k$. An interested reader can find more detailed experiment results at `https://github.com/antoniodistasio/pgsolver-APT`.

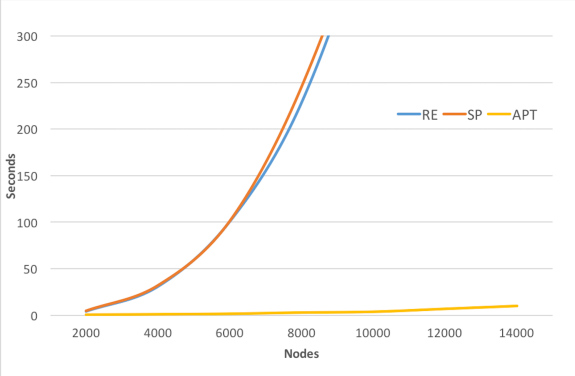

Fig. 3. Runtime executions with $n = e^k$

6 Conclusion

The `APT` algorithm, an automata-theoretic technique to solve parity games, has been designed two decades ago by Kupferman and Vardi [20], but never considered to be useful in practice [12]. In this paper, for the first time, we fill missing gaps and implement this algorithm. By means of benchmarks based on random games, we show that it is the best performing algorithm for solving parity games when the number of priorities is very small *w.r.t.* the number of states. We believe that this is a significant result as several applications of parity games to formal verification and synthesis do yield games with a very small number of priorities.

The specific setting of a small number of priorities opens up opportunities for specialized optimization technique, which we aim to investigate in future work. This is closely related to the issue of accelerated algorithms for three-color parity games [10]. We also plan to study why the performance of the `APT` algorithm is so sensitive to the relative number of priorities, as shown in Table 3.

References

1. B. Aminof, O. Kupferman, and A. Murano. Improved Model Checking of Hierarchical Systems. *Inf. Comput.*, 210:68–86, 2012.
2. A. Antonik, N. Charlton, and M. Huth. Polynomial-Time Under-Approximation of Winning Regions in Parity Games. *ENTCS*, 225:115–139, 2009.
3. D. Berwanger. Admissibility in Infinite Games. In *STACS'07*, pages 188–199, 2007.
4. K. Chatterjee, L. Doyen, T. A. Henzinger, and J.-F. Raskin. Generalized Mean-payoff and Energy Games. In *FSTTCS'10*, LIPIcs 8, pages 505–516, 2010.
5. K. Chatterjee and M. Henzinger. An $O(n^2)$ Time Algorithm for Alternating Büchi Games. In *SODA'12*, pages 1386–1399, 2012.

6. K. Chatterjee, T. A. Henzinger, and M. Jurdzinski. Mean-payoff parity games. In *LICS'05*, pages 178–187, 2005.
7. K. Chatterjee, M. Jurdzinski, and T. A. Henzinger. Quantitative stochastic parity games. In *SODA'04*, pages 121–130, 2004.
8. E.M. Clarke and E.A. Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *LP'81*, LNCS 131, pages 52–71, 1981.
9. E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. 2002.
10. L. de Alfaro and M. Faella. An Accelerated Algorithm for 3-Color Parity Games with an Application to Timed Games. In *CAV'07*, LNCS 4590, pages 108–120, 2007.
11. E.A. Emerson and C. Jutla. Tree Automata, μ-Calculus and Determinacy. In *FOCS'91*, pages 368–377, 1991.
12. O. Friedmann and M. Lange. The PGSolver collection of parity game solvers. *University of Munich*, 2009.
13. O. Friedmann and M. Lange. Solving Parity Games in Practice. In *ATVA*, LNCS 5799, pages 182–196, 2009.
14. K. Heljanko, M. Keinänen, M. Lange, and I. Niemelä. Solving Parity Games by a Reduction to SAT. *J. Comput. Syst. Sci.*, 78(2):430–440, 2012.
15. M. Jurdzinski. Deciding the Winner in Parity Games is in UP $\cap$ co-Up. *Inf. Process. Lett.*, 68(3):119–124, 1998.
16. M. Jurdzinski.Small Progress Measures for Solving Parity Games. In *STACS*, pages 290–301, 2000.
17. M. Jurdzinski, M. Paterson, and U. Zwick. A Deterministic Subexponential Algorithm for Solving Parity Games. *SIAM J. Comput.*, 38(4):1519–1532, 2008.
18. D. Kozen. Results on the Propositional μ-Calculus. *TCS*, 27(3):333–354, 1983.
19. O. Kupferman, M. Vardi, and P. Wolper. Module Checking.164(2):322–344, 2001.
20. O. Kupferman and M. Y. Vardi. Weak Alternating Automata and Tree Automata Emptiness. In *STOC*, pages 224–233, 1998.
21. O. Kupferman, M.Y. Vardi, and P. Wolper. An Automata Theoretic Approach to Branching-Time Model Checking. *Journal of the ACM*, 47(2):312–360, 2000.
22. F. Mogavero, A. Murano, and L. Sorrentino. On Promptness in Parity Games. In *LPAR'13*, LNCS 8312, pages 601–618, 2013.
23. D.E. Muller, A. Saoudi, and P.E. Schupp. Weak Alternating Automata Give a Simple Explanation of Why Most Temporal and Dynamic Logics are Decidable in Exponential Time. In *LICS'88*, pages 422–427, 1988.
24. J.P. Queille and J. Sifakis. Specification and Verification of Concurrent Programs in Cesar. In *81*, LNCS 137, pages 337–351, 1981.
25. S. Schewe. Solving Parity Games in Big Steps. In *FSTTCS'07*, LNCS 4855, pages 449–460, 2007.
26. S. Schewe. An Optimal Strategy Improvement Algorithm for Solving Parity and Payoff Games. In *CSL'08*, LNCS 5213, pages 369–384, 2008.
27. W. Thomas. Facets of Synthesis: Revisiting Church's Problem. In *FOSSACS'09*, LNCS 5504, pages 1–14, 2009.
28. J. Vöge and M. Jurdzinski. A Discrete Strategy Improvement Algorithm for Solving Parity Games. In *CAV'00*, LNCS 1855, pages 202–215, 2000.
29. I. Walukiewicz. Pushdown Processes: Games and Model Checking. In *CAV'96*, pages 62–74, 1996.
30. T. Wilke. Alternating Tree Automata, Parity Games, and Modal μ-Calculus. *Bulletin of the Belgian Mathematical Society Simon Stevin*, 8(2):359, 2001.
31. W. Zielonka. Infinite Games on Finitely Coloured Graphs with Applications to Automata on Infinite Trees. *Theor. Comput. Sci.*, 200(1-2):135–183, 1998.