Annals of Mathematics and Artificial Intelligence
https://doi.org/10.1007/s10472-020-09721-3

Improving parity games in practice

Antonio Di Stasio[1] · **Aniello Murano**[2] · **Vincenzo Prignano**[2] · **Loredana Sorrentino**[2]

Accepted: 18 November 2020Published online: 23 January 2021
© The Author(s) 2021

Abstract
Parity games are infinite-round two-player games played on directed graphs whose nodes are labeled with priorities. The winner of a play is determined by the smallest priority (even or odd) that is encountered infinitely often along the play. In the last two decades, several algorithms for solving parity games have been proposed and implemented in `PGSolver`, a platform written in OCaml. `PGSolver` includes the *Zielonka's recursive algorithm* (`RE`, for short) which is known to be the best performing one over random games. Notably, several attempts have been carried out with the aim of improving the performance of `RE` in `PGSolver`, but with small advances in practice. In this work, we deeply revisit the implementation of `RE` by dealing with the use of specific data structures and programming languages such as *Scala*, *Java*, *C++*, and *Go*. Our empirical evaluation shows that these choices are successful, gaining up to three orders of magnitude in running time over the classic version of the algorithm implemented in `PGSolver`.

Keywords Formal verification · Zielonka Recursive algorithm · PGSolver

Mathematics Subject Classification (2010) 68Q60

1 Introduction

Parity games [22, 52] are abstract infinite-round games that represent a powerful mathematical framework to address fundamental questions in computer science. They are intimately

✉ Antonio Di Stasio
distasio@diag.uniroma1.it

Aniello Murano
aniello.murano@unina.it

Vincenzo Prignano
vincenzo.prigano@gmail.com

Loredana Sorrentino
loredana.sorrentino@unina.it

[1] Università di Roma "La Sapienza", Rome, Italy

[2] Università degli Studi di Napoli Federico II, Napoli, Italy

related to other infinite-round games, such as *mean* and *discounted* payoff, *stochastic*, and *multi-agent* games [2, 9, 12–15, 38].

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 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 whether 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 [29]; whether a polynomial time solution exists is a long-standing open question [51].

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 [52], the Jurdziński's small-progress measures algorithm [30] (SPM), the strategy-improvement algorithm by Jurdziński et al. [50], the (sub-exponential) algorithm by Jurdziński et al. [32], the big-step algorithm by Schewe [44], the APT algorithm by Di Stasio et al. [20, 21].

Table 1 summarizes the classic algorithms along with their complexity, where n, e, and c denote the number of nodes, edges, and priorities, respectively.

Recently, Calude et al. [11] have given a major breakthrough providing a quasi-polynomial time algorithm for solving parity games that runs in time $O(n^{\lceil log(c)+6 \rceil})$. Previously, the best known algorithm for parity games was DD which could solve parity games in $O(n^{\sqrt{n}})$, so this new result represents a significant advance in the understanding of parity games. This new approach is based on providing a compact witness that can be used to decide whether Player 0 wins a play. Traditionally, one must store the entire history of a play, so that when the players construct a cycle, we can easily find the largest priority on that cycle. The key observation in [11] is that a witness of poly-logarithmic size can be used instead. This allows to simulate a parity game on an alternating Turing machine that uses poly-logarithmic space, which leads to a deterministic algorithm that uses quasi-polynomial time and space. This new result has already inspired follow-up works [10, 18, 23, 31, 37]. However, benchmarks in the literature have demonstrated that both on random games and real examples the quasi-polynomial is not the best performing one.

In formal system design [16, 17, 35, 43], parity games arise as a natural evaluation machinery for the automatic synthesis and verification of distributed and reactive systems [1, 36, 46], as they allow to express *liveness* and *safety* properties in a very elegant and powerful way [40]. Specifically, the model-checking question, in case the specification is given as a

Table 1 Parity algorithms along with their computational complexities

Algorithm	Computational complexity
Recursive (RE) [52]	$O(e \cdot n^c)$
Small Progress Measures (SPM) [30]	$O(c \cdot e \cdot (\frac{n}{c})^{\frac{c}{2}})$
Strategy Improvement (SI) [50]	$O(2^e \cdot n \cdot e)$
Dominion Decomposition (DD) [32]	$O(n^{\sqrt{n}})$
Big Step (SPM) [44]	$O(e \cdot n^{\frac{1}{3}c})$
APT [20]	$O(n^c)$

μ-calculus formula [34], can be rephrased, in linear-time, as a parity game [22]. 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.

All algorithms mentioned in Table 1 (and several others) have been implemented in `PGSolver`, written in *OCaml* by Oliver Friedman and Martin Lange [24, 25], a collection of tools to solve, benchmark and generate parity games, and extensively investigated experimentally. Noteworthy, `PGSolver` has allowed to declare `RE` as the best performing to solve parity games in practice, as well as explore some optimizations such as decomposition into strongly connect components, removal of self-cycles on nodes, and priority compression [3, 30].

Despite the enormous interest in finding efficient algorithms for solving parity games, less emphasis has been put on how to improve their running time choosing efficient data structures or different programming languages for their implementation. Mainly, the scientific community has relied on OCaml as the best performing programming language to be used in this setting and `PGSolver` as an optimal platform to solve parity games. However, starting from graphs with a few thousand of nodes, even using `RE`, `PGSolver` would require minutes to solve the given game, especially on dense graphs. Therefore, a natural question that arises is whether there exists a way to improve the running time of `PGSolver`. Focusing the attention on `RE`, we identify two research directions to work on, which specifically involve: the way it is implemented, and the chosen programming language. As a result we introduce a slightly improved version of the `RE` along with an optimized implementation in OCaml (and then in `PGSolver`) and *Scala* programming languages. Scala [41, 42] is a high-level language, proven to be well performing [28], with object and functional oriented features, that recently has come to the fore with useful applications in several fields of computer science including *formal verification* [5]. The choice to investigate first on Scala, among all programming languages, is substantiated as it shares several modern programming language aspects. Among the others, Scala carries functional and object-oriented features, compiles its programs for the JVM, is interoperable with Java and a high-level language with a concise and clear syntax, and the results we obtain strongly support our choice and allow to declare Scala as a first winner over OCaml, in terms of performance.

In more details, our investigation starts by looking at the main steps of `RE` and how they are implemented in `PGSolver`. Overall, we observe that `RE` requires to decompose the game into multiple smaller games, which is done by computing, in every recursive call, the *difference* between the game and a given set of nodes. This operation turns out to be quite expensive as it requires to generate a new game at each iteration as well as building the transpose of the game graph. In order to reduce the running-time complexity caused by these graph operations, we exploit a new implementation by introducing a requirement for immutability of the game ensuring that every recursive call uses the game without applying any modification to its nodes. Therefore, to construct the sub-games in the recursive calls, we keep track efficiently of each node that is going to be removed from the graph. Therefore, this improved version guarantees that the original game remains immutable tracking the removed nodes in every subsequent call and checking, in constant time, whether a node needs to be excluded or not. Finally, we also improve other parts of the algorithm, such as finding the maximal priority and obtaining all nodes with a given priority.

In our analysis, we first consider and compare four implementations of `RE`. The Classic (*C*) and Improved (*I*) Recursive (*R*) algorithms implemented in Scala (*S*) and OCaml (*O*). By means of benchmarks, we show that *IRO* gains an order of magnitude against *CRO, as well as CRS* against *CRO*. Remarkably, we show that these improvements are cumulative by proving that *IRS* gains two order of magnitude against *CRO*.

The proposed improvements turn out to be very successful in practice and the benchmarks we have run show this evidence. But, to have a complete overview we continue our investigation by also exploring these improvements in different programming languages such as *Java*, *C++*, and *Go*, and comparing their performance among them, as well as with Scala. The tool and more details can be found at https://github.com/vincepri/SPGSolver.

The combination of the improvements described above along with the use of these modern programming languages allows us to considerably gain in running time. We evaluated our implementations over both randomly generated games and real world benchmarks such as model checking problems. Our main finding is that C++ is the best performing one in both cases. Precisely, the experiments ran over random games show that the C++ implementation is three orders of magnitude faster than the classic version of the Zielonka's algorithm implemented in `PGSolver`, and with respect the improved version of `RE`, C++ is faster than Scala and Go by a factor of 4 and 1.5, respectively. They also report that Java and Scala have a very similar behavior. This trend is followed in the real world benchmarks, though they indicate more nuanced relationship between the programming languages. Overall, we have that Go running time, besides to be of the same order, tends to be closer of the C++ one. We take this as an interesting aspect to take into account. Indeed, although Go is a young programming language, it provides a high performance, super efficient concurrency handling like Java and fun to code like Python/Perl. Therefore, our results suggests the need to continue the investigation in evaluating the implementation of algorithms for solving parity games with the use of modern programming languages as Go.

Related works Several efforts in speeding up the solving process was previously attempted. In [27], a GPU (Graphics Processing Units) is used to solve parity games. A GPU can excel at problems that can be easily split into a large number of parallels tasks. A modern GPU consists of several multi-processors which act independently of each other. Hoffmann and Luttenberger [27] proposes a GPU-enabled implementation for solving parity games such as `SPM`, `RE` and a variant of `SI`. In particular, for storing the graph and the node information multiple arrays are used, and before starting the actual solving process the implementation makes use of multiple preprocessing steps. Another approach to this problem was investigated in [4] where it is implemented, on a multi-core architecture, a parallel version of the attractor algorithm, that is the main kernel of `RE`, implementing the parallel algorithm using Java and comparing it with its sequential implementation in the same language, and the widely used one in `PGSolver`. Verver [49] provides an improved version of `RE` by recording the number of remaining "escaping" edges for each vertex during the attractor computation. This accomplished by simply using an extra integer per node.

This work, based on the preliminary results in [19], has been the first to exploit the use of efficient coding and new programming languages to improve the use of `RE` in practice. Noteworthy, it has inspired the development of new and efficient tools such as *Oink* by Tom van Dijk [48]. *Oink* contains an improved version of `RE`, `SPM`, and `SI`, the quasi-polynomial algorithms, `APT`, the priority promotion [7], and the tangle algorithm [47]. It also provides a multi-core implementation of `SPM` and `RE`, inspired by the results in [4].

Outline The sequel of the paper is structured as follows. In Section 1, we give some preliminary concepts about parity games. In Section 2, we describe `RE` and how it is implemented in `PGSolver`. In Section 3, we introduce our improved version of `RE`, and provide its

implementation in OCaml. In Section 4, we describe the Scala implementation of RE. In Section 5, we analyze the benchmarks of RE and the improved version of RE in OCaml and Scala. In Section 6, we explore the improved version of RE along different programming languages, and we analyze their performance by means of benchmarks.

2 Preliminaries

In this section, we briefly recall some basic concepts regarding parity games. As notation, the positive integers are denoted by $\mathbb{N}$, and $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$. A *Parity Game* (PG, for short) is a tuple $\mathcal{G} \triangleq \langle \mathrm{P}_0, \mathrm{P}_1, Mv, \mathsf{p} \rangle$, where P_0 and P_1 are two finite disjoint sets of nodes for Player 0 and Player 1, respectively, with $\mathrm{P} = \mathrm{P}_0 \cup \mathrm{P}_1$, $Mv \subseteq \mathrm{P} \times \mathrm{P}$, is the left-total binary relation of moves, and $\mathsf{p} : \mathrm{P} \to \mathbb{N}_0$ is the priority function. Each player moves a token along nodes by means of the relation Mv. By $Mv(q) \triangleq \{q' \in \mathrm{P} : (q, q') \in Mv\}$ 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 Fig. 1. The set of players's nodes is $\mathrm{P}_0 = \{\mathsf{q}_0, \mathsf{q}_3, \mathsf{q}_4, \mathsf{q}_5\}$ and $\mathrm{P}_1 = \{\mathsf{q}_1, \mathsf{q}_2, \mathsf{q}_6\}$; 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 $\mathsf{p}(\mathsf{q}_2) = \mathsf{p}(\mathsf{q}_5) = 5$.

A *play* (*resp.*, *history*) over $\mathcal{G}$ is an infinite (*resp.*, finite) sequence $\pi = \pi_1 \cdot \pi_2 \cdot \ldots \in \mathrm{Pth} \subseteq \mathrm{P}^\omega$ (*resp.*, $\pi = \pi_1 \cdot \ldots \cdot \pi_n \in \mathrm{Hst} \subseteq \mathrm{P}^*$) 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 Fig. 1, a possible play is $\bar{\pi} = \mathsf{q}_1 \cdot \mathsf{q}_5 \cdot \mathsf{q}_2 \cdot (\mathsf{q}_3)^\omega$, while a possible history is given by $\bar{\pi} = \mathsf{q}_1 \cdot \mathsf{q}_5 \cdot \mathsf{q}_2 \cdot \mathsf{q}_3$.

For a given play $\pi = \pi_1 \cdot \pi_2 \cdot \ldots$, by $\mathsf{p}(\pi) = \mathsf{p}(\pi_1) \cdot \mathsf{p}(\pi_2) \cdot \ldots \in \mathbb{N}^\omega$ we denote the associated priority sequence. As an example, the associated priority sequence to $\bar{\pi}$ is given by $\mathsf{p}(\bar{\pi}) = 1 \cdot 5 \cdot 5 \cdot (2)^\omega$.

For a given history $\pi = \pi_1 \cdot \ldots \cdot \pi_n$, by $\mathsf{fst}(\pi) \triangleq \pi_1$ and $\mathsf{lst}(\pi) \triangleq \pi_n$ we denote the first and last node occurring in π, respectively. For the example history, we have that $\mathsf{fst}(\bar{\pi}) = \mathsf{q}_1$ and $\mathsf{lst}(\bar{\pi}) = \mathsf{q}_3$. By Hst_0 (*resp.*, Hst_1) we denote the set of histories π such that $\mathsf{lst}(\pi) \in \mathrm{P}_0$ (*resp.*, $\mathsf{lst}(\pi) \in \mathrm{P}_1$). 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

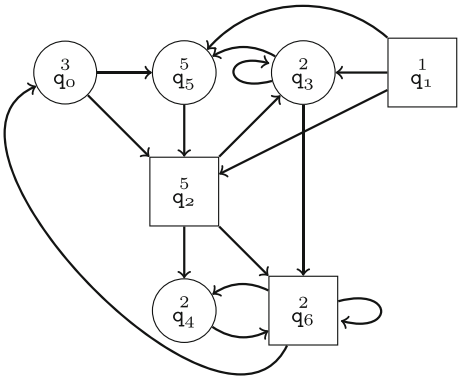

Fig. 1 A parity game

for Player 0 (resp., Player 1) if $\min(\mathsf{Inf}(\mathsf{p}(\pi)))$ is even (*resp.*, odd). In the running example, we have that $\mathsf{Inf}(\bar{\pi}) = \{\mathsf{q}_3\}$ and $\mathsf{Inf}(\mathsf{p}(\bar{\pi})) = \{2\}$ and so, π is winning for Player 0.

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

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 $\pi_i \in \mathrm{P}_0$, 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 $\hat{\pi}, \tilde{\pi} \in \mathrm{Hst}_0$ (*resp.*, $\hat{\pi}, \tilde{\pi} \in \mathrm{Hst}_1$), with $\mathsf{lst}(\hat{\pi}) = \mathsf{lst}(\tilde{\pi})$, it holds that $\mathsf{str}_0(\hat{\pi}) = \mathsf{str}_0(\tilde{\pi})$ (*resp.*, $\mathsf{str}_0(\hat{\pi}) = \mathsf{str}_1(\hat{\pi})$). 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 $\mathsf{str}_0 : \mathrm{P}_0 \to \mathrm{P}$ and $\mathsf{str}_1 : \mathrm{P}_1 \to \mathrm{P}$.

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 $q \in \mathrm{Win}_1(\mathcal{G})$ [22]. 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 [52].

Attractor We now define the notion of attractor, core of the Zielonka's algorithm. Intuitively, given a set of nodes $F \subseteq \mathrm{P}$, the i-attractor of F for a Player $i \in \{0, 1\}$, indicated by $Attr_i(\mathcal{G}, F)$, represents those nodes that i can force the play toward. That is, Player i can force any play to behave in a certain way, even though this does not mean that Player i wins the game. Formally, for all $k \in \mathbb{N}_0$:

- $Attr_i^0(\mathcal{G}, F) = F$.
- $Attr_i^{k+1}(\mathcal{G}, F) = Attr_i^k(\mathcal{G}, F) \cup \{v \in \mathrm{P}_i \mid \exists w \in Attr_i^k(\mathcal{G}, F) : (v, w) \in Mv\} \cup \{v \in \mathrm{P}_{1-i} \mid \forall w : (v, w) \in Mv \Rightarrow w \in Attr_i^k(\mathcal{G}, F)\}$.

Then, we have that $Attr_i(\mathcal{G}, F) = \bigcup_{k \in \mathbb{N}_o} Attr_i^k(\mathcal{G}, F)$.

Subgames Let A be an arbitrary attractor set. The subgame $\mathcal{G} \setminus A$ is the game restricted to the nodes $\mathrm{P} \setminus A$, i.e., $\mathcal{G} \setminus A = (\mathrm{P}_0 \setminus A, \mathrm{P}_1 \setminus A, Mv \setminus (A \times \mathrm{P} \cup \mathrm{P} \times A), \mathsf{p}_{|\mathrm{P} \setminus A})$, where $\mathsf{p}_{|\mathrm{P} \setminus A}$ is the restriction of p to A. It is worth observing that the totality of $\mathcal{G} \setminus A$ is ensured from A being an attractor.

3 Zielonka's recursive algorithm

In this section, we describe the recursive algorithm by Zielonka using the basic concepts introduced in the previous sections and some observations regarding its implementation in `PGSolver`.

Algorithm 1 Zielonka's recursive algorithm (RE).

```
1:  procedure SOLVE(G)
2:      if (P == ∅) then return (∅, ∅)
3:      else
4:          d = maximal priority in G
5:          U = { v ∈ V | p(v) = d }
6:          i = d mod 2,
7:          j = 1 − i
8:          A = Attr_i(U)
9:          (W'_0, W'_1) = SOLVE (G \ A)
10:         if W'_j == ∅ then
11:             W_i = W'_i ∪ A
12:             W_j = ∅
13:         else
14:             B = Attr_j(W'_j)
15:             (W'_0, W'_1) = SOLVE (G \ B)
16:             W_p = W'_i
17:             W_j = W'_j ∪ B
18:     return (W_0, W_1)
```

The recursive algorithm (RE, for short), reported in Algorithm 1, is one of the first exponential-time algorithm for solving parity games. It is based on the work of McNaughton [39] and it was explicitly presented as a solver for parity games by Zielonka [52]. The algorithm makes use of a divide and conquer technique and its core subroutine is the *attractor* described in Section 2.

At each step, the algorithm removes all nodes with the highest priority d, denoted by U, together with all nodes Player $i = d \ mod \ 2$ can attract to them, denoted by A, and recursively computes the winning sets (W'_0, W'_1) for Player 0 and Player 1, respectively, on the remaining subgame $\mathcal{G} \setminus A$.

At this point, there are two cases to be considered. First, if Player i wins $\mathcal{G} \setminus A$, then he also wins the whole game $\mathcal{G}$. Indeed, whenever Player $1 - i$ decides to visit A, Player i's winning strategy would be to reach U. Then, every play that visits A infinitely often has d as the highest priority occurring infinitely often, or otherwise it stays eventually in $\mathcal{G} \setminus A$, and hence is won by i.

Second, if Player i does not win the whole subgame $\mathcal{G} \setminus A$, i.e., W'_{1-i} is non empty, then Player $1-i$ wins on a subset W'_{1-i} in $\mathcal{G} \setminus A$. And, since Player i cannot force Player $1-i$ to leave W'_{1-i}, we have that Player $1-i$ also wins on W'_{1-i} in the game $\mathcal{G}$. Hence, the algorithm computes the attractor B for Player $1-i$ of W'_{1-i} and recursively solves the subgame $\mathcal{G} \setminus B$.

3.1 The implementation of RE in PGSolver

PGSolver turns out to have a very limited application in several real scenarios. In more details, even using RE (that has been shown to be the best performing in practice), PGSolver would require minutes to decide games with few thousands of nodes, especially on dense graphs. In this work we deeply study all main aspects that cause such a bad performance.

Specifically, our investigation begins with the way in which RE has been implemented in PGSolver by means of the OCaml programming language. We start observing that the graph data structure in this framework is represented as a fixed length *Array* of tuples.

Every tuple has all information that a node needs, such as the player, the assigned priority and the adjacency list. Before every recursive call is performed, the program computes the difference between the graph and the attractor, as well as it builds the transposed graph. In addition the attractor function makes use of a *TreeSet* data structure that is not available in the OCaml's standard library, but it is imported from *TCSlib*, a multi-purpose library for OCaml written by Oliver Friedmann and Martin Lange. Such library implements this data structure using *AVL-Trees* that guarantees logarithmic search, insert, and removal. Also, the same function computes the number of successors for the opponent player in *every* iteration when looping through every node in the attractor.

4 An improved implementation of `RE`

All the observations given above lead to introduce an improved version of `RE` (`IRE`, for short), we report in Algorithm 2. In Algorithm 3 we report an improved version of the attractor function that the new algorithm makes use of.

Algorithm 2 Improved `RE` (`IRE`).

```
1:  procedure WIN(G)
2:      T = G.transpose()
3:      return winI(G, T, {})
4:  procedure WINI(G, T, Removed)
5:      if |V| == |Removed| then return (∅, ∅)
6:      d = maximal priority in G
7:      U = { v ∈ V | p(v) = d }
8:      i = d mod 2
9:      j = 1 − i
10:     A = ATTR (G, T, Removed, U, i)
11:     (W'_0, W'_1) = WINI(G, T, Removed ∪ A)
12:     if W'_j == ∅ then
13:         W_i = W'_i ∪ A
14:         W_j = ∅
15:     else:
16:         B = ATTR(G, T, Removed, W'_j, j)
17:         (W'_0, W'_1) = WINI(G, T, Removed ∪ B)
18:         W_i = W'_i
19:         W_j = W'_j ∪ B
20:     return (W_0, W_1)
```

Let $\mathcal{G}$ be a graph. Removing a node from $\mathcal{G}$ and building the transposed graph takes time $\Theta(|V| + |E|)$. Thus, dealing with dense graph this operation takes $\Theta(|V|^2)$. In order to reduce the running time complexity caused by these graph operations, we introduce an immutability requirement to the graph $\mathcal{G}$ ensuring that every recursive call uses $\mathcal{G}$ without applying any modification to the state space of the graph. Therefore, to construct the subgames, in the recursive calls, we keep track of each node that is going to be removed from the graph, adding all of them to a set called *Removed*.

The improved algorithm is capable of checking whether a given node is excluded or not in constant time as well as it completely removes the need for a new graph in every recursive

call. At first glance this may seem a small improvement with respect to `RE`. However, it turns out to be very successful in practice as proved in the following benchmark section. Further evidences that boost the importance of such improvement can be related to the fact that the difference operation has somehow the same compliance of complementing automata [45]. Using our approach is like avoiding such complementation by adding constant information to the states, i.e. a flag ($removed$, $\neg removed$). Last but not least, about the actual implementation, it is also worth mentioning that general-purpose *memory allocators* are very expensive as the pre-operation cost floats around one hundred processor cycles [26]. Many efforts have been made over the years to improve memory allocation implementing custom allocators from scratch, a process known to be difficult and error prone [8].

Algorithm 3 Improved attractor.

```
1:  procedure ATTR(G, T, Removed, A, i)
2:      tmpMap = []
3:      for each x ∈ V do
4:          if x ∈ A then
5:              tmpMap[x] = 0
6:          else
7:              tmpMap[x] = −1
8:      index = 0
9:      while index < |A| do
10:         for v0 ∈ adj(T, A[index]) do
11:             if v0 ∉ Removed then
12:                 if tmpMap[v0] == −1 then
13:                     if player(v0) == i then
14:                         A = A ∪ v0
15:                         tmpMap[v0] = 0
16:                     else
17:                         adj_counter = −1
18:                         for x ∈ adj(G, v0) do
19:                             if x ∉ Removed then
20:                                 adj_counter + = 1
21:                         tmpMap[v0] = adj_counter
22:                         if adj_counter == 0 then
23:                             A = A ∪ {v0}
24:                 else if (player(v0) == i and tmpMap[v0] > 0) then
25:                     tmpMap[v0] − = 1
26:                     if tmpMap[v0] == 0 then
27:                         A = A ∪ {v0}
28:     return A
```

4.1 Implementation in OCaml for `PGSolver`

Our implementation of `IRE` in OCaml, listed in Algorithm 4, does not directly modify the graph data structure (that is represented in `PGSolver` as an array of tuples), but rather it uses a set to keep track of removed nodes.

IRE takes three parameters: a parity game, the transpose of the game graph, and a set of excluded nodes. Our improved attractor uses a *HashMap*, called *tmpMap* to keep track of the number of successors for the opponent player's nodes. In addition, we use a *Queue*, from OCaml's standard library, to loop over the nodes in the attractor. Aiming at performance optimizations, the attractor function implemented in PGSolver also returns the set of excluded nodes.

Algorithm 4 Implementation of IRE in OCaml

```
let rec win game tgraph exc =
let w = Array.make 2 InteSet.empty in
if(not ((Array.length game) =
    (InteSet.cardinal exc))) then(
    let (d,u)=(max_prio_and_set game exc) in
    let p=d mod 2 in
    let j=1-p in
    let w1=Array.make 2 InteSet.empty in
    let (attr,exc1)=attr_fun game exc tgraph u p in
    let (sol0,sol1)=win game tgraph exc1 in
    w1.(0)<-sol0;
    w1.(1)<-sol1;
    if (InteSet.is_empty w1.(j)) then (
        w.(p)<-(InteSet.union w1.(p) attr);
        w.(j)<-InteSet.empty;
    else (
        let (attr_B,exc2) =
        attr_fun game exc tgraph w1.(j) j in
        let (sol0_B,sol1_B) = win game
        tgraph exc2 in
        w1.(0)<-sol0_B;
        w1.(1)<-sol1_B;
        w.(p)<-w1.(p) ;
        w.(j)<-(InteSet.union w1.(j) attr_B);
    )
);
(w.(0),w.(1))
;;
```

5 Scala implementation

In this section, we give an implementation of IRE in the Scala programming language, starting with a brief introduction to it. Scala [41, 42] is the programming language designed by Martin Odersky, the codesigner of Java Generics and main author of *javac* compiler. Scala defines itself as a *scalable* language, statically typed, a fusion of an object-oriented language and a functional one. It runs on the *Java Virtual Machine* (JVM) and supports every existing Java library. Scala is a purely object-oriented language in which, like Java and Smalltalk, every value is an object and every operation is a method call. In addition, Scala is a

functional language where every function is a first class object, and it is equipped with efficient immutable and mutable data structures with a strong selling point given by Java interoperability. However, it is not a purely functional language as objects may change their states and functions may have side effects. The functional aspects are perfectly integrated with the object-oriented features. The combination of both styles makes possible to express new kinds of patterns and abstractions. All these features make Scala programming language as a clever choice to solve these tasks, in a strict comparison with other programming languages available such as C, C++ or Java.

In [28], researchers by Google show that Scala, even being an high level language, performs just 2.5x slower than C++ machine optimized code. In particular, Scala was shown to be faster than Java. As the paper notes: "*While the benchmark itself is simple and compact, it employs many language features, in particular high level data structures, a few algorithms, iterations over collection types, some object oriented features and interesting memory allocation patterns*".

Algorithm 5 Improved algorithm in Scala.

```
def win(G: GraphWithSets):(ArrayBuffer[Int],
  ArrayBuffer[Int]) = {val W =
    Array(ArrayBuffer.empty[Int],
    ArrayBuffer.empty[Int])
    val d = G.max_priority()
    if (d > -1) {
      val U = G.priorityMap.get(d)
    .filter(p => !G.exclude(p))
    val p = d % 2
    val j = 1 - p
    val W1 =
     Array(ArrayBuffer.empty[Int],
     ArrayBuffer.empty[Int])
    val A = Attr(G, U, p)
    val res = win(G -- A)
    W1(0) = res._1
    W1(1) = res._2
    if (W(j).size == 0) {
       W(p) = W1(p) ++= A
       W(j) = ArrayBuffer.empty[Int]
    } else {
       val B = Attr(G, W1(j), j)
       val res2 = win(G -- B)
       W1(0) = res2._1
       W1(1) = res2._2
       W(p) = W1(p)
       W(j) = W1(j) ++= B
    }
  }
(W(0), W(1))
}
```

5.1 Implementation in Scala of IRE

In this section we introduce our implementation of the IRE in Scala, reported in Algorithm 5 and 6.

Aiming at performance optimizations we use a *priority HashMap* where every *key* is a priority, and *value* is a set of nodes having *key* as priority, i.e., $priority(v) = key$ for every v beloging to $value$. Moreover, we use the data structures *HashMaps* and *ArrayLists* contained in the open source library *Trove*.

Algorithm 6 Improved attractor in Scala.

```
1: def Attr(G: GraphWithSets, A: ArrayBuffer[Int], i: Int)
2:   : ArrayBuffer[Int] = {
3:   val tmpMap = Array.fill[Int](G.nodes.size)(-1)
4:   var index = 0
5:   A.foreach(tmpMap(_) = 0)
6:   while (index < A.size) {
7:     G.nodes(A(index)).<~.foreach(v0 => {
8:       if (!G.exclude(v0)) {
9:         val flag = G.nodes(v0).player == i
10:        if (tmpMap(v0) == -1) {
11:          if (flag) {
12:            A += v0
13:            tmpMap(v0) = 0
14:          } else {
15:            val tmp = G.nodes(v0).~>
16:            .count(x => !G.exclude(x)) - 1
17:            tmpMap(v0) = tmp
18:            if (tmp == 0) A += v0
19:          }
20:        }else if (!flag && tmpMap(v0) > 0){
21:          tmpMap(v0) -= 1
22:          if (tmpMap(v0) == 0) A += v0
23:        }
24:      }
25:    })
26:    index += 1
27:  }
28:  A
29: }
```

We rely on Scala's internal features and standard library making heavy use of the dynamic *ArrayBuffer* data structure. In order to store the arena we use an array of *Node* objects. The Node class contains: a list of adjacent nodes, a list of incident nodes, its priority and the player. ArrayBuffer also implements a factory method called "$--(set : ArrayBuffer[Int])$" that takes an ArrayBuffer of integers as input, flags all the nodes in the array as excluded, and returns the reference to the new graph. In addition, there is also a method called $max\_priority()$ that will return the maximal priority in the graph and the set of nodes with that priority.

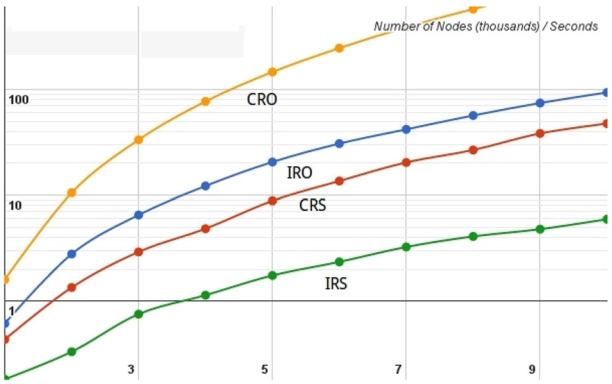

Fig. 2 Random games chart in logarithmic scale

Our attractor implementation in Scala makes use of an array of integers named $tmpMap$ that is pre-allocated using the number of nodes in the graph with a negative integer as default value; we use $tmpMap$ when looping through every node in the set A given as parameter to keep track of the number of successors for the opponent player. We add a node $v \in V$ to the attractor set when its counter (stored in *tmpMap[v]*) reaches 0 ($adj(v) \subseteq A$ and $v \in V_{opponent}$) or if $v \in V_{player}$; using an array of integers, or an HashMap guarantees a constant time check if a node was already visited and ensures that the count for the opponent's node adjacency list takes place one time only.

6 Experimental evaluations: new implementations in OCaml and Scala

In this section we study, analyze and evaluate the running time of the following implementations: *Classic Recursive in OCaml* (*CRO*), *Classic Recursive in Scala* (*CRS*), *Improved Recursive in OCaml* (*IRO*) and *Improved Recursive in Scala* (*IRS*). We have run our experiments on multiple instances of random parity games. Note that *IRS* and *CRS* do not apply any optimization steps to the arena before solving, while the OCaml implementations run those optimizations. However, to show the effectiveness of Scala implementations we keep them enable. All tests have been run on an Intel(R) Xeon(R) CPU E5620 @ 2.40 GHz, with 16GB of Ram (with no Swap available) running Ubuntu 14.04.[1] Precisely, we have used 100 random arenas generated using `PGSolver` of each of the following types, given $N = i \times 1000$ with i integer and $1 \leq i \leq 10$ and a timeout set at 600 s.

In the following, we report six tables in which we show the running time of all experiments under fixed parameters. Throughout this section we define `abort[`T`]` when the program has been aborted due to excessive time and `abort[`M`]` when the program has been killed by the Operating System due to memory consumption. In Fig. 2 we also report the trends of the four implementations using a logarithmic scale with respect to *seconds*. This figure is based on the averages of all results reported in the tables below.

[1] Note that this configuration is used for all tests ran in this work.

N nodes, N colors, $adj(1,\ N)$					N nodes, N colors, $adj(\frac{N}{2},\ N)$				
N	*IRS*	*CRO*	*CRS*	*IRO*	N	*IRS*	*CRO*	*CRS*	*IRO*
1	**0.204**	1.99	0.505	0.752	1	**0.179**	1.21	0.454	0.583
2	**0.456**	13.208	1.918	3.664	2	**0.389**	8.075	1.173	2.366
3	**1.031**	41.493	2.656	6.147	3	**0.868**	25.097	2.656	6.147
4	**1.879**	96.847	6.728	15.966	4	**1.279**	57.186	4.23	10.452
5	**2.977**	183.589	12.616	27.272	5	**2.273**	108.983	9.206	20.377
6	**3.993**	306.104	19.032	41.051	6	**2.772**	183.884	12.562	27.489
7	**4.989**	486.368	27.05	50.367	7	**3.748**	291.077	17.942	37.521
8	**6.103**	`abort`[T]	36.597	70.972	8	**3.942**	418.377	22.105	47.502
9	**7.287**	`abort`[T]	55.171	97.216	9	**4.989**	593.721	23.93	61.593
10	**8.468**	`abort`[T]	68.303	113.36	10	**6.413**	`abort`[T]	42.408	80.508

N nodes, 2 colors, $adj(\frac{N}{2},\ N)$					N nodes, 2 colors, $adj(1,\ N)$				
N	*IRS*	*CRO*	*CRS*	*IRO*	N	*IRS*	*CRO*	*CRS*	*IRO*
1	**0.189**	1.98	0.481	0.702	1	**0.159**	1.226	0.385	0.468
2	**0.469**	12.941	1.55	3.17	2	**0.341**	7.965	1.004	2.162
3	**1.046**	41.584	3.995	7.428	3	**0.797**	25.114	2.305	6.014
4	**1.712**	96.545	5.378	13.823	4	**1.123**	56.422	3.699	9.421
5	**2.414**	181.225	11.273	22.575	5	**1.704**	108.584	6.12	14.971
6	**3.458**	307.233	16.472	35.269	6	**2.243**	182.935	10.099	22.621
7	**4.612**	484.159	26.448	49.311	7	**3.324**	286.503	13.898	32.335
8	**6.003**	`abort`[T]	28.968	65.674	8	**3.95**	430.265	19.743	44.281
9	**7.03**	`abort`[T]	43.666	85.909	9	**4.597**	`abort`[T]	28.742	56.81
10	**8.938**	`abort`[T]	57.18	110.814	10	**5.651**	`abort`[T]	33.639	71.434

N nodes, $\sqrt{N}$ colors, $adj(\frac{N}{2},\ N)$					N nodes, $\sqrt{N}$ colors, $adj(1,\ N)$				
N	*IRS*	*CRO*	*CRS*	*IRO*	N	*IRS*	*CRO*	*CRS*	*IRO*
1	**0.204**	1.978	0.468	0.71	1	**0.162**	1.218	0.384	0.475
2	**0.456**	13.114	1.575	3.203	2	**0.344**	7.947	1.034	2.195
3	**1.031**	41.493	3.868	7.492	3	**0.788**	25.029	2.406	5.944
4	**1.621**	96.55	5.744	13.97	4	**1.105**	57.307	3.835	9.608
5	**2.439**	183.589	10.72	22.98	5	**1.678**	108.623	6.34	15.165
6	**3.372**	307.426	15.978	34.78	6	**2.281**	182.154	9.871	22.859
7	**4.662**	485.826	26.432	48.875	7	**3.193**	285.28	14.338	32.536
8	**6.499**	`abort`[T]	34.741	66.423	8	**4.185**	422.74	20.362	44.515
9	**7.147**	`abort`[T]	48.915	86.645	9	**5.009**	599.071	24.347	57.022
10	**8.988**	`abort`[T]	56.656	111.492	10	**5.76**	`abort`[T]	35.024	72.291

6.1 Trends analysis for random games

The speedup obtained by our implementation of `IRE` is in most cases quite noticeable. Figure 3 shows the running time trend for `RE` and `IRE` in both OCaml and Scala based on the results of the previous benchmarks. The seconds, showed on the Y-Axis, are limited to

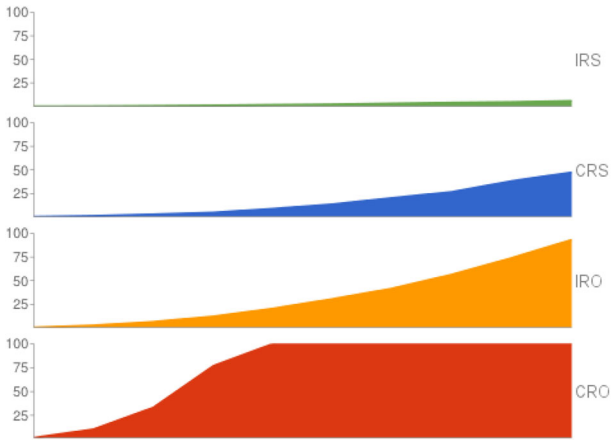

Fig. 3 Trends Chart *(seconds/number of nodes)*

[0, 100], while on the X-Axis we report the number of nodes. As a result we show that even with all preprocessing steps enabled in `PGSolver`, *IRS* is capable of gaining two orders of magnitude in running time.

6.2 Trends analysis for special games

Here, we compare the performance of *CRO* and *IRS* over non-random games generated by `PGSolver` such as clique games, ladder games, model checker ladder games, and Jurdzinski games. These experiments have been run disabling all optimizations in `PGSolver` since *IRS* does not apply such optimizations.

Clique[n] games are fully connected games without self-loops, where n is the number of nodes. The set of nodes is partitioned into V_0 and V_1 having the same size. For all $v \in V_p$, $priority(v)\,\%\,2 = p$. For our experiments we set $n = 2^k$ where $8 \leq k \leq 14$. Table 2 reports the running time for our experiments and these results are drawn in Fig. 4.

In *Ladder[n]* game, every node in V_0 has priority 2 and every node in V_1 has priority 1. In addition, each node $v \in V$ has two successors: one in V_0 and one in V_1, which form a node pair. Every pair is connected to the next pair forming a *ladder* of pairs. Finally, the last pair is connected to the top. The parameter n specifies the number of node pairs. For our tests, we set $n = 2^k$ where $8 \leq k \leq 19$, and report our experiments in Table 3 whose trend is drawn in Fig. 5, where the seconds are limited to [0,2]. As the figure shows, there is a better performance for *CRO* than *IRS* using low-scaled (up to 2^{13}) values as input parameter. This behavior is not surprising as there is a *warming-up* time required by the Java Virtual Machine.

Table 2 Clique games

n	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}
IRS	0.05	0.07	0.12	0.46	1.18	4.87	17.39
CRO	0.09	0.61	4.37	29.58	229.78	`abort`$[T]$	`abort`$[M]$

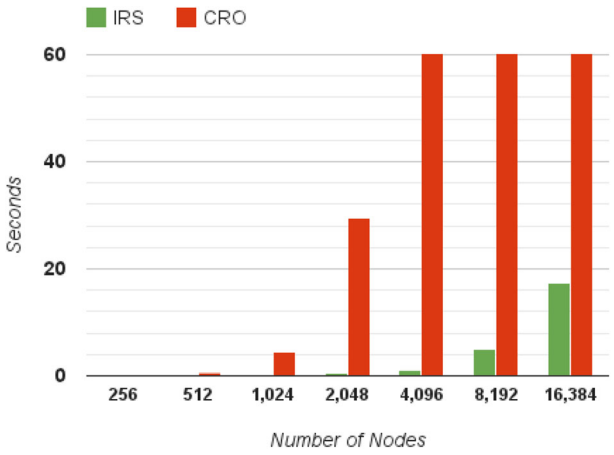

Fig. 4 Clique trends

Model Checker Ladder[n] consists of overlapping blocks of four nodes, where the parameter n specifies the number of desidered blocks. Every node is owned by player 1, i.e., $V_1 = V$ and $V_0 = \emptyset$, and the nodes are connected such that every cycle passes through a single point of colour 0. For our experiments we set $n = 2^k$ where $10 \leq k \leq 15$, and report our experiments in Table 4 below and draw the trends in Fig. 6, where the seconds are limited to [0, 2].

Jurdzinski[n, m] games are designed to generate the worst-case behavior for `SPM` [30]. The parameter n is the number of layers, where each layer has m repeating blocks that are inter-connected as described in [30]. As this game takes two parameters, in our test we ran two experiments: one where n is fixed to 10 and $m = 10 \times 2^k$, for $k = 1, \ldots, 5$ and one where m is fixed to 10 and $n = 10 \times 2^k$, for $k = 1, \ldots, 5$. The results of our experiments are reported in Table 5. The trends are drawn in Fig. 7.

7 New implementations and experimental results

Our investigation has explored two possible directions to improve the performance of `RE`: *i)* using new data structures and more efficient coding, *ii)* exploiting its implementation along a different programming language as Scala. The experiments have highlighted how the combination of these two directions turns out to be very efficient in practice, showing an improvement of up to one order of magnitude both in improving the `RE` implementation in OCaml and choosing a different programming language. Thus, we reach an improvement of up to two orders of magnitude applying both.

Table 3 Runtime executions over ladder games

n	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}
IRS	0.02	0.03	0.05	0.08	0.11	0.13	0.15	0.19	0.25	0.38	0.48	0.93
CRO	0.00	0.01	0.01	0.03	0.06	0.13	0.3	0.65	1.39	2.93	6.21	11.71

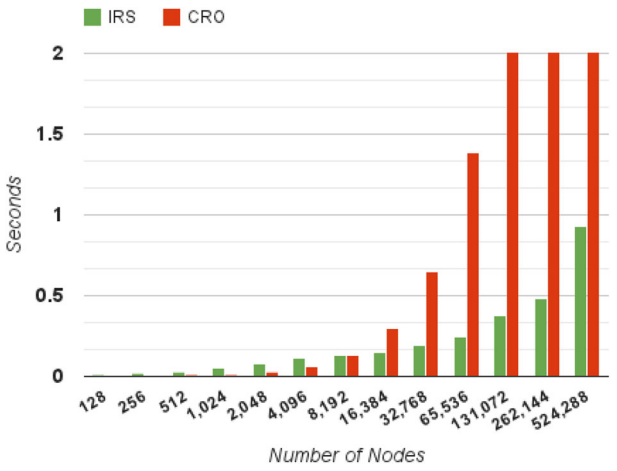

Fig. 5 Ladder trends

Table 4 Model checker ladder games

n	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}
IRS	0.04	0.07	0.12	0.14	0.16	0.19	0.21	0.26	0.39	0.65
CRO	0.01	0.02	0.05	0.10	0.22	0.47	0.99	2.12	4.16	8.31

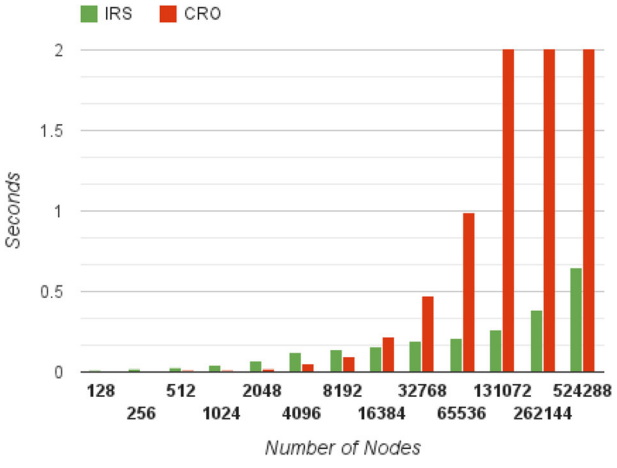

Fig. 6 Model checker ladder trends

Table 5 Model checker ladder games

m	10×2^1	10×2^2	10×2^3	10×2^4	10×2^5
IRS	0.21	0.48	1.54	4.55	15.31
CRO	0.23	0.79	3.14	15.77	65.85
n	10×2^1	10×2^2	10×2^3	10×2^4	10×2^5
IRS	0.28	0.77	3.02	30.02	232.24
CRO	0.42	2.94	22.69	184.12	`abort`$[T]$

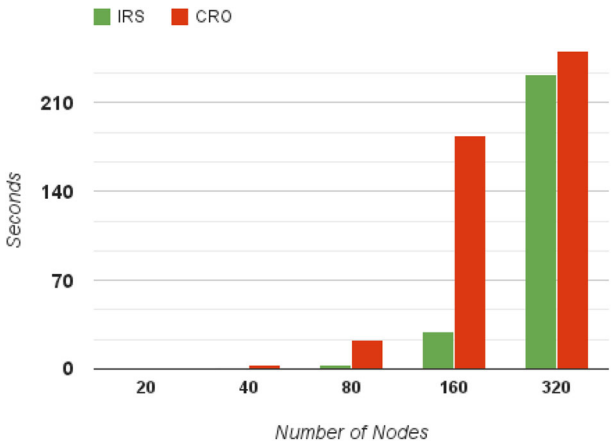

Fig. 7 Jurdiznski trends

In this section, we continue our investigation on the programming language side presenting multiple implementations in Java, C++ and Go of `IRE`. We introduce the languages used, making an overall explanation of their key aspects to achieve our primary goal. The full implementation can be found online via https://github.com/vincepri/SPGSolver.

The **Java** Programming Language was developed by James Gosling at Sun Microsystem and introduces for the first time in 1995. The latest release of Java is a huge step forward for the language that enriches the syntax and the standard library. It is a clear demonstration of a language evolution without compromising robustness, stability and still ensuring backward compatibility. Our Java solver implementation relies mainly on the standard library, Google Guava [6] library and Trove for high performing data structures. The Trove library offers regular and primitive collections for Java with high speed and memory efficient. Internally Trove does not use any java.lang.Number subclasses, in this way there is no boxing/unboxing overhead. The TIntArrayList data structure is built on top of an array using the corresponding data type (int[] in this case). Each Trove Array List has several helper method inherited from the java.util.Collections.

Modern **C++** can be seen in three parts: low level language inherited from C, advanced language features and the standard library (stdlib) that provides useful data structures and algorithms.

Our implementation in C++ makes intense use of the language's standard library and the Boost C++ libraries for string based algorithms when parsing files and timer functions. The tool was compiled with clang, a compiler front end for C, C++ and Objective-C that uses LLVM as its backend. The compiled executable is around 62KB on disk and it must be noted that was compiled with full optimizations enabled (-Ofast flag).

The C++ version, showed a huge memory footprint saving compared to garbage collected programming languages. A Python module, that makes use the C++ implementation, has also been implemented using *Boost.Python*.

The **Go** Programming Language is an efficient, statically-typed compile language developed at Google in 2007. It has built-in support for concurrency and communication, a latency-free garbage collection and high speed compilation process. The standard library provides all core packages programmers need to build real world programs, such as two fundamental built-in collection types: slices (variable-length arrays) and maps, built to be

efficient and to serve multiple purposes. The language does not hide pointers and there is no virtual machine getting in the way of performance, for this reason it is completely possible to design complex custom types with ease. Our implementation of parity games in Go strictly follows the Go rules and conventions for code syntax and only requires the standard library, keeping external dependencies at minimum.

8 Benchmarks and trends

In this section we study, analyze and evaluate the running time of our implementations.

We have run our experiments on multiple instances of random parity games.

Precisely, we have used 100 random games generated by `PGSolver` with a number of nodes defined as $N = i \times 1000$ with $1 \leq i \leq 60$, and two priorities.

It is worth to note that these implementations do not apply any preprocessing steps to the arena before solving. The graph data structure is represented, in a consistent manner across the languages, as a fixed-length Array of objects, where every node contains perfect information such as the player, priority and adjacency list.

The trends are shown in Fig. 8, which gives a clear idea of what is possible to achieve in under a second, while Fig. 9, shows the clear gap between OCaml and the other languages.

The chosen languages have deep differences between each other, Go and C++, for example, make use of a static compiler to produce a full native binary, while Java and Scala use the JVM's compiler. In addition, the JVM languages and Go are garbage-collected while C++11 uses RAII as a programming idiom where holding a resource is tied to an object lifetime and the language itself guarantees that an object is freed once control flow leaves the scope.

We finally include a plot in Fig. 10 to further compare the different implementations. Tests are performed using 20 random arenas generated through `PGSolver`, setting the number of nodes as $N = i \times 1000$ with $1 \leq i \leq 60$, number of priorities $p \in \left\{2,\ \sqrt{n},\ n\right\}$, and *minimum* and *maximum* number of edges $(min,\ max) \in \left\{(1,\ n),\ \left(\frac{n}{2},\ n\right)\right\}$. Note that

Fig. 8 Trend comparison *(seconds/number of nodes)*

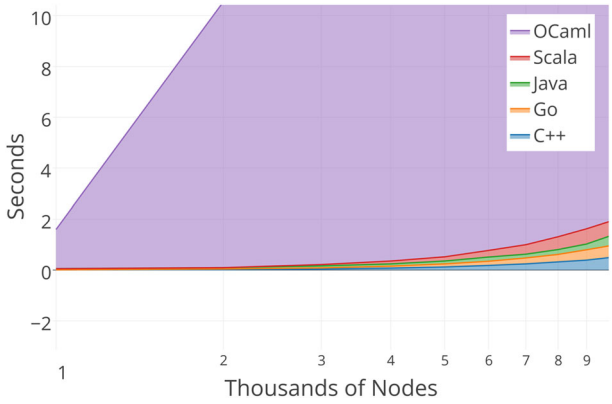

Fig. 9 Linear time comparison

the plot is shrunk after 16k nodes in order to show the behavior of the implementations up to 60K nodes.

Our benchmarks empirically show that C++ is the best performing one, in every test we have run, reducing the Scala running time by a factor of 4 and Go by a factor of ˜1.5. Scala and Java tend to have a very similar behavior over 40000 nodes, in fact in some plots the results are missing because the JVM would go over 32GB of available memory, while Go was always capable of completing the solving process even if taking more time. A further investigation on why performance was degrading so quickly for Go after the 40000 nodes threshold led to the garbage collector implementation, Go's collector is a mark-and-sweep with periodic pauses when it runs; on the other hand Java provides few different implementation for its collector allowing multiple performance optimizations.

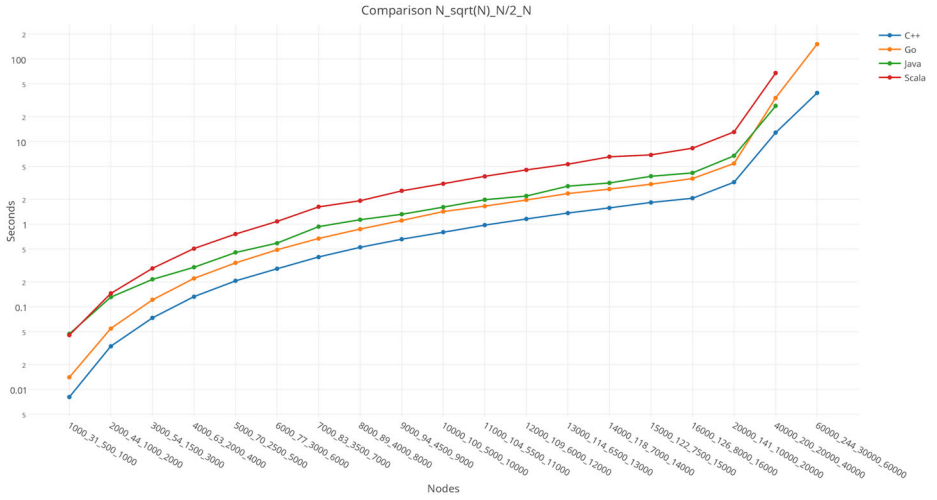

Fig. 10 Comparison among the implementations

Table 6 SWP (Sliding Window Protocol)

n	Property	IRO	IRC	IRG	IRJ	IRS	WS	DS
14,065	ND	0.073	0.00024	0.00064	0.0023	0.028	2	2
17,810	IORD1	0.16	0.00031	0.00074	0.0073	0.042	2	2
34,673	IORW	14.72	0.0016	0.0026	0.011	0.062	2	2
2,589,056	ND	0.99	0.0055	0.88	0.015	0.13	4	2
3,487,731	IORD1	2.24	0.0068	0.01	0.049	0.36	4	2
6,823,296	IORW	916.88	0.025	0.031	0.079	0.42	4	2

8.1 Benchmarks on practical model checking problems

The experiments on random games have showed an overview on the performance of our implementations, but to better understand their behavior in the practice applications of parity games we need to continue our investigation. Therefore, we evaluate the performance of our implementation of `IRE` in OCaml (*IRO*), C++ (*IRC*), Go (*IRG*), Java (*IRJ*), and Scala (IRS) on some practical model checking problems as in [33].

Specifically, we use models coming from: the Sliding Window Protocol (SWP) with window size (WS) of 2 and 4 (WS represents the boundary of the total number of packets to be acknowledged by the receiver) and the Onebit Protocol (OP). The properties we check on these models concerns: absence of deadlock (ND), a message of a certain type (d1) is received infinitely often (IORD1), and if there are infinitely many read steps then there are infinitely many write steps (IORW).

Note that, in all benchmarks, data size (DS) denotes the number of messages, and every game has 3 priorities.

As we can see, by comparing Tables 6 and 7, the experiments indicate a more nuanced relationship between the different implementations of `IRE`. Indeed, even though the experiments follow the trend showed previously for the random case, that is, C++ is the best performing one, they also show a different behavior for the other implementations. Overall, we can observe that IRJ gains one order of magnitude over IRS in all protocols and properties. Thus, the gap between C++ and Scala reaches up to two orders of magnitude, differently from what we have seen for the random case. While, IRC and IRG reduce the IRJ running time of one order of magnitude. Here, another interesting aspect is highlighted by the comparison between the IRC and IRG running time where a closer distance is showed, though IRC outperforms IRG in all cases. Finally, the experiments clearly show the significant gap that IRO gets against the other implementations.

Table 7 OP (Onebit Protocol)

n	Property	IRO	IRC	IRG	IRJ	IRS	DS
81,920	ND	0.82	0.0033	0.0054	0.0095	0.069	2
88,833	IORD1	1.55	0.0038	0.0058	0.031	0.096	2
170,752	IORW	`abort`[T]	0.013	0.017	0.045	0.21	2
289,297	ND	3.22	0.014	0.020	0.028	0.19	4
308,737	IORD1	6.08	0.015	0.021	0.085	0.36	4
607,753	IORW	`abort`[T]	0.052	0.063	0.144	0.61	4

9 Conclusions

PGSolver is a well-known framework that collects multiple algorithms to decide parity games. For several years this platform has been the only one available to solve and benchmark in practice. However, given PGSolver's limitations addressing huge graphs, several attempts of improvement have been carried out. Some of them have been implemented as preprocessing steps in the tool itself, such as priority compression or SCC decomposition and the like [25], while others chose parallelism techniques applied to the algorithms, as done in [4, 27].

In this work we start from scratch by revisiting the implementation of RE in PGSolver. We first provide an improved version by using new data structures and more efficient coding. The improved version guarantees that the original game remains immutable tracking the removed nodes in every subsequent call and checking, in constant time, whether a node needs to be excluded or not. Our preliminary results show that our implementation allows to gain up to one order of magnitude over the implementation in PGSolver.

Then, we exploit its implementation along different programming languages such as Scala, *Java*, *C++*, and *Go* and compare among them. The experimental results give a clear and perfect idea of which implementation is outperforming the solving process. C++ is the best performing one, in every run, and it is capable of gaining up to three orders of magnitude in running time over its classical implementation in PGSolver. Specifically, on random games, C++ reduces Scala running time by a factor of 4 and Go by a factor of $\sim$1.5. Go's performance behavior tends to degrade after 40000 nodes, outperformed by Java in some cases. Instead, the benchmarks executed over practical model checking problems show that Go's performance, besides being of the same order, tends to be closer of the C++ one. This highlights an interesting aspect about the use of Go. Indeed, although Go is a young programming language, it provides a high performance, super efficient concurrency handling like Java and fun to code like Python/Perl. Then, even though C++ has the better performance, a deep investigation in the benefits of using Go might be lead to appealing results.

The importance of this work, that is an extension of the results in [19], relies on the fact that has been the first to exploit the use of efficient coding and modern programming languages to improve the performance of RE, as it is used for comparisons in the literature. Indeed, our results have already inspired to development new efficient tools for solving parity games. Among others, *Oink*, developed by Tom van Dijk [48], makes use of our improvement like the immutability of the game in implementing RE, it is written in C++, and provides multi-core implementation of RE and SPM as done in [4]. Finally, this work points out interesting questions that we take into account for future works such as continue the evaluation of Go's performance in real scenarios, as well as the implementation of a multi-core version of the algorithms in Go and Java.

Funding Open access funding provided by Università degli Studi di Roma La Sapienza within the CRUI-CARE Agreement.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommonshorg/licenses/by/4.0/.

References

1. Aminof, B., Kupferman, O., Murano, A.: Improved model checking of hierarchical systems. Inf. Comput. **210**, 68–86 (2012)
2. Aminof, B., Malvone, V., Murano, A., Rubin, S.: Graded modalities in strategy logic. Inf Comput. **261**(Part), 634–649 (2018)
3. Antonik, A., Charlton, N., Huth, M.: Polynomial-time under-approximation of winning regions in parity games. ENTCS **225**, 115–139 (2009)
4. Arcucci, R., Marotta, U., Murano, A., Sorrentino, L.: Parallel parity games: a multicore attractor for the zielonka recursive algorithm. In: ICCS 2017, pp. 525–534 (2017)
5. Barringer, H., Havelund, K.: TraceContract: A Scala DSL for Trace Analysis. Springer, Berlin (2011)
6. Bejeck, B.: Getting Started with Google Guava. Packt Publishing, Burmingham (2013)
7. Benerecetti, M., Dell'Erba, D., Mogavero, F.: Solving parity games via priority promotion. In: CAV 2016, pp. 270–290 (2016)
8. Berger, E.D., Zorn, B.G., McKinley, K.S.: OOPSLA 2002: reconsidering custom memory allocation. SIGPLAN Not. **48**(4S), 46–57 (2013)
9. Berwanger, D.: Admissibility in infinite games. In: STACS 2007, pp. 188–199 (2007)
10. Boker, U., Lehtinen, K.: FSTTCS 2018, Ahmedabad, India, pp 21:1–21:22 (2018)
11. Calude, C.S., Jain, S., Khoussainov, B., Li, W., Stephan, F.: Deciding parity games in quasipolynomial time. In: STOC 2017, pp. 252–263 (2017)
12. Cermák, P., Lomuscio, A., Murano, A.: Verifying and synthesising multi-agent systems against one-goal strategy logic specifications. In: AAAI 2015, pp. 2038–2044 (2015)
13. Chatterjee, K., Jurdzinski, M., Henzinger, T.A.: Quantitative stochastic parity games. In: SODA, vol. 2004, pp. 121–130 (2004)
14. Chatterjee, K., Henzinger, T.A., Jurdzinski, M.: Mean-payoff parity games. In: LICS 2005, pp. 178–187 (2005)
15. Chatterjee, K., Doyen, L., Henzinger, T.A., Raskin, J.-F.: Generalized mean-payoff and energy games. In: FSTTCS 2010, pp. 505–516 (2010)
16. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching-time temporal logic. In: LP 1981, pp. 52–71 (1981)
17. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (2002)
18. Daviaud, L., Jurdzinski, M., Lazic, R.: A pseudo-quasi-polynomial algorithm for mean-payoff parity games. In: LICS 2018, pp. 325–334 (2018)
19. Di Stasio, A., Murano, A., Prignano, V., Sorrentino, L.: Solving parity games in scala. In: FACS 2014, pp. 145–161 (2014)
20. Di Stasio, A., Murano, A., Perelli, G., Vardi, M.Y.: Solving parity games using an automata-based algorithm. In: CIAA 2016, pp. 64–76 (2016)
21. Di Stasio, A., Murano, A., Vardi, M.Y.: Solving parity games: explicit vs symbolic. In: CIAA 2018, pp. 159–172 (2018)
22. Emerson, E.A., Jutla, C.: Tree automata, μ-calculus and determinacy. In: FOCS, vol. 1991, pp. 368–377 (1991)
23. Fearnley, J., Jain, S., Schewe, S., Stephan, F., Wojtczak, D.: An ordered approach to solving parity games in quasi polynomial time and quasi linear space. In: SPIN 2017, pp. 112–121 (2017)
24. Friedmann, O., Lange, M.: The PGSolver collection of parity game solvers. University of Munich (2009)
25. Friedmann, O., Lange, M.: Solving parity games in practice. In: ATVA 2009, pp. 182–196 (2009)
26. Gay, D., Aiken, A.: Memory Management with Explicit Regions, vol 33. ACM (1998)
27. Hoffmann, P., Luttenberger, M.: Solving parity games on the GPU. In: ATVA 2013, pp. 455–459 (2013)
28. Hundt, R.: Loop recognition in c++/java/go/scala. In: Proceedings of Scala Days (2011)
29. Jurdzinski, M.: Deciding the winner in parity games is in UP $\cap$ co-Up. Inf. Process. Lett. **68**(3), 119–124 (1998)
30. Jurdzinski, M.: Small progress measures for solving parity games. In: STACS 2000, pp. 290–301 (2000)
31. Jurdzinski, M., Lazic, R.: Succinct progress measures for solving parity games. In: LICS 2017, pp. 1–9 (2017)
32. Jurdzinski, M., Paterson, M., Zwick, U.: A deterministic subexponential algorithm for solving parity games. SIAM J. Comput. **38**(4), 1519–1532 (2008)
33. Keiren, J.J.A.: Benchmarks for parity games. In: FSEN 2015, pp. 127–142 (2015)
34. Kozen, D.: Results on the propositional μ-calculus. TCS **27**(3), 333–354 (1983)

35. Kupferman, O., Vardi, M.Y., Wolper, P.: An automata theoretic approach to branching-time model checking. J. ACM **47**(2), 312–360 (2000)
36. Kupferman, O., Vardi, M., Wolper, P.: Module checking. Inf. Comput. **164**(2), 322–344 (2001)
37. Lehtinen, K.: A modal μ perspective on solving parity games in quasi-polynomial time. In: LICS 2018, pp. 639–648 (2018)
38. Malvone, V., Murano, A., Sorrentino, L.: Concurrent multi-player parity games. In: AAMAS 2016, pp. 689–697 (2016)
39. McNaughton, R.: Infinite games played on finite graphs. Ann. Pure Appl. Logic **65**(2), 149–184 (1993)
40. Mogavero, F., Murano, A., Sorrentino, L.: On promptness in parity games. Fundam. Inform. **139**(3), 277–305 (2015)
41. Odersky, M., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., Zenger, M., et al.: An overview of the scala programming language (2004)
42. Odersky, M., Spoon, L., Venners, M.: Programming in Scala. Artima Inc (2008)
43. Queille, J.P., Sifakis, J.: Specification and verification of concurrent programs in Cesar. In: SP 1982, pp. 337–351 (1982)
44. Schewe, S.: Solving Parity Games in Big Steps. In: FSTTCS 2007, pp. 449–460 (2007)
45. Thomas, W.: Automata on infinite objects. In: Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B), pp. 133–192 (1990)
46. Thomas, W.: Facets of synthesis: revisiting church's problem. In: FOSSACS 2009, pp. 1–14 (2009)
47. van Dijk, T.: Attracting tangles to solve parity games. In: CAV 2018, pp. 198–215 (2018)
48. van Dijk, T.: Oink: an implementation and evaluation of modern parity game solvers. In: TACAS 2018, pp. 291–308 (2018)
49. Verver, M.: Practical improvements to parity game solving. Master's thesis, University of Twente (2013)
50. Vöge, J., Jurdzinski, M.: A discrete strategy improvement algorithm for solving parity games. In: CAV 2000, pp. 202–215 (2000)
51. Wilke, T.: Alternating tree automata, parity games, and modal μ-calculus. Bull. Belg. Math. Soc. Simon Stevin **8**(2), 359 (2001)
52. Zielonka, W.: Infinite games on finitely coloured graphs with applications to automata on infinite trees. Theor. Comput. Sci. **200**(1-2), 135–183 (1998)

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.