Solving Parity Games in Scala

Antonio Di Stasio, Aniello Murano(✉), Vincenzo Prignano,
and Loredana Sorrentino

Università degli Studi di Napoli Federico II, Napoli, Italy
murano@na.infn.it, vincenzo.prignano@gmail.com

Abstract. *Parity games* are two-player games, played on directed graphs, whose nodes are labeled with priorities. Along a play, the maximal priority occurring infinitely often determines the winner. In the last two decades, a variety of algorithms and successive optimizations have been proposed. The majority of them have been implemented in *PGSolver*, written in OCaml, which has been elected by the community as the de facto platform to solve efficiently parity games as well as evaluate their performance in several specific cases.

PGSolver includes the *Zielonka Recursive Algorithm* that has been shown to perform better than the others in randomly generated games. However, even for arenas with a few thousand of nodes (especially over dense graphs), it requires minutes to solve the corresponding game.

In this paper, we deeply revisit the implementation of the recursive algorithm introducing several improvements and making use of *Scala Programming Language*. These choices have been proved to be very successful, gaining up to two orders of magnitude in running time.

1 Introduction

Parity games [13,35] are abstract infinite-duration games that represents a powerful mathematical framework to address fundamental questions in computer science and mathematics. They are strict connected with other games of infinite duration, such as *mean* and *discounted* payoff, *stochastic*, and *multi-agent* games [7–10].

In formal system design and verification [12,25], parity games arise as a natural evaluation machinery to automatically and exhaustively check for reliability of distributed and reactive systems [1,3,26]. More specifically, in formal verification, *model-checking* techniques [11,31] allow to verify whether a system is correct with respect to a desired behavior by checking whether a mathematical model of the system meets a formal specification of the expected execution. In case the latter is given by means of a μ-calculus formula [24], the model checking problem can be translated, in linear-time, into a parity game [13]. Hence, every parity game solver can be used in practice as a model checker for a μ-calculus specification (and vice-versa). Using this approach, *liveness* and *safety*

Aniello Murano—Partially supported by the FP7 European Union project 600958-SHERPA and OR.C.HE.S.T.R.A. MIUR PON project.

I. Lanese and E. Madelaine (Eds.): FACS 2014, LNCS 8997, pp. 145–161, 2015.
DOI: 10.1007/978-3-319-15317-9_9

properties can be addressed in a very elegant and easy way [28]. Also, this offers a very powerful machinery to check for component software reliability [1,3].

In the basic settings, parity games are two-player turn-based games, played on directed graphs, whose nodes are labeled with *priorities* (i.e., natural numbers). The players, named *player* 0 and *player* 1, move in turn a token along graph's edges. Thus, a play induces an infinite path and player 0 wins the play if the greatest priority visited infinitely often is even; otherwise, player 1 wins the play.

The problem of finding a winning strategy in parity games is known to be in UPTime ∩ CoUPTime [21] and deciding whether a polynomial time solution exists or not is a long-standing open question. Aimed to find the right complexity of parity games, as well as come out with solutions working efficiently in practice, several algorithms have been proposed in the last two decades. In Table 1, we report the most common ones along with their known computational complexities, where parameters n, e, and d denote the number of nodes, edges, and priorities in the game, respectively (for more details, see [15,16]).

Table 1. Parity algorithms along with their computational complexities.

Condition	Complexity
Recursive [35]	$O(e \cdot n^d)$
Small Progress Measures [22]	$O(d \cdot e \cdot (\frac{n}{d})^{\frac{d}{2}})$
Strategy Improvement [34]	$O(2^e \cdot n \cdot e)$
Dominion Decomposition [23]	$O(n^{\sqrt{n}})$
Big Step [32]	$O(e \cdot n^{\frac{1}{3}d})$

All above mentioned algorithms have been implemented in *PGSolver*, written in *OCaml* by Oliver Friedman and Martin Lange [15,16], a collection of tools to solve, benchmark and generate parity games. Noteworthy, PGSolver has allowed to declare the Zielonka Recursive Algorithm as the best performing to solve parity games in practice, as well as explore some optimizations such as decomposition into strong connect components, removal of self-cycles on nodes, and priority compression [2,22].

Despite the enormous interest in finding efficient algorithms for solving parity games, less emphasis has been put on the choice of the programming language. Mainly, the scientific community relies on OCaml as the best performing programming language to be used in this setting and PGSolver as an optimal and *the de facto* platform to solve parity games. However, starting from graphs with a few thousand of nodes, even using the Zielonka algorithm, PGSolver would require minutes to decide 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. We identify three research directions to work on, which specifically involve: the algorithm itself, the way it is implemented, and the chosen programming language. As a result we introduce, in this paper, a slightly improved version of the Classic Zielonka Algorithm along with a heavily optimized implementation in *Scala Programming Language* [29,30]. Scala is a high-level language, proven to be well performing [20], 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* [4]. Our experiments show that, by using all Scala features extensively, we are able of gaining two order of magnitude in running time with respect to the implementation of the Zielonka algorithm in PGSolver.

In details, the main goal of this work is the design and development of a new tool for solving parity games, based on an improved version of the Zielonka Recursive Algorithm, with performance in mind. Classical Zielonka algorithm requires to decompose the graph game into multiple smaller arenas, which is done by computing, in every recursive call, the *difference* between the current graph and a given set of nodes. This operation (Fig. 1, lines 10 and 15) turns out to be quite expensive as it requires to generate a new graph at each iteration. Somehow such a difference operation has the flavor of the complicancy of complementing automata in formal verification [33]. Remarkably, our improved version guarantees that the original arena remains immutable by tracking the removed nodes in every subsequent call and checking, in constant time, whether a node needs to be excluded or not. Casting this idea in the above automata reasoning, it is like enriching the state space with two flags ($removed$, $\neg removed$), instead of performing a complementation.

In this paper we consider and compare four implementations. The Classic (C) and Improved (I) Recursive (R) algorithms implemented in Scala (S) and OCaml (O). Using random generated games, 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*.

We have been able to achieve this kind of performance optimization by deeply studying the way the classic Recursive algorithm has been implemented in PGSolver and concentrating on the following tasks of the algorithm, which we have deeply improved: finding the maximal priority, finding all nodes with a given priority, and removing a node (including related edges) from the graph. Parsing the graph in Scala, we allocate an *Array*, whose size is fixed to the number of nodes of the graph. In addition we populate at the same time the adjacency list and incidence list for each node, which avoids to build a transposed graph. We make also use of an open source Java library called *Trove* that provides a fast and lightweight implementation of the *java.util* Collection API.

Finally, we want to remark that, among all programming languages, we have chosen to investigate Scala as it shares several modern and useful programming language aspects. Among the others, Scala carries functional and object-oriented features, compiles its programs for the JVM, is interoperable with Java and an high-level language with a concise and clear syntax. The results we obtain strongly support our choice and allow to declare Scala as a clear winner over OCaml, in terms of performance.

Outline. The sequel of the paper is structured as follows. In Sect. 2, we give some preliminary concepts about parity games. In Sect. 3, we describe the Classic Recursive Zielonka Algorithm. In Sect. 4, we introduce our improved algorithm based on the Zielonka algorithm that we implement in Sect. 5 using Scala programming language. In Sect. 6 we study, analyze, and benchmark the Classic and Improved Algorithms in OCaml (PGSolver) and Scala.

Finally we report that the tool is available as an open source project at https://github.com/vinceprignano/SPGSolver.

2 Parity Games

In this section we report some basic concepts about parity games including the Zielonka Recursive Algorithm. For more details we refer to [14,35].

A *parity game* is a tuple $G = (V, V_0, V_1, E, \varOmega)$ where (V, E) forms a directed graph whose set of nodes is partitioned into $V = V_0 \cup V_1$, with $V_0 \cap V_1 = \emptyset$, and $\varOmega : V \rightarrow N$ is the *priority function* that assigns to each node a natural number called the *priority* of the node. We assume E to be *total*, i.e. for every node $v \in V$, there is a node $w \in V$ such that $(v, w) \in E$. In the following we also write vEw in place of $(v, w) \in E$ and use $vE := \{w \,|\, vEw\}$.

Parity games are played between two players called *player 0* and *player 1*. Starting in a node $v \in V$, both players construct an infinite path (the *play*) through the graph as follows. If the construction reaches, at a certain point, a finite sequence $v_0...v_n$ and $v_n \in V$ then player i selects a node $w \in v_nE$ and the play continues with the sequence $v_0...v_nw$. Every play has a unique winner, defined by the priority that occurs infinitely often. Precisely, the *winner* of the play $v_0v_1v_2...$ is player i iff $max\{p \,|\, \forall j \,.\, \exists k \geq j \,:\, \varOmega(v_k) = p\} \, mod \, 2 = i$. A *strategy* for player i is a partial function $\sigma : V^*V \rightarrow V$, such that, for all sequences $v_0...v_n$ with $v_{j+1} \in v_jE$, with $j = 0, ..., n-1$, and $v_n \in V_i$ we have that $\sigma(v_0...v_n) \in v_nE$. A play $v_0v_1...$ *conforms* to a strategy σ for player i if, for all j we have that, if $v_j \in V_i$ then $v_{j+1} = \sigma(v_0...v_j)$. A strategy σ for player i (σ_i) is a winning strategy in node v if player i wins every play starting in v that conforms to the strategy σ. In that case, we say that player i *wins* the game G starting in v. A strategy σ for player i is called *memoryless* if, for all $v_0...v_n \in V^*V_i$ and for $w_0...w_m \in V^*V_i$, we have that if $v_n = w_m$ then $\sigma(v_0...v_n) = \sigma(w_0...w_m)$. That is, the value of the strategy on a path only depends on the last node on that path. Starting from G we construct two sets $W_0, W_1 \subseteq V$ such that W_i is the set of all nodes v such that player i wins the game G starting in v. Parity games enjoy *determinacy* meaning that for every node v either $v \in W_0$ or $v \in W_1$ [13].

The problem of solving a given parity game is to compute the sets W_0 and W_1, as well as the corresponding *memoryless* winning strategies, σ_0 for player *0* and σ_1 for player *1,* on their respective winning regions. The construction procedure of winning regions makes use of the notion of *attractor*. Formally, let $U \subseteq V$ and $i \in \{0, 1\}$. The i-attractor of U is the least set W s.t. $U \subseteq W$ and whenever $v \in V_i$ and $vE \cap W \neq \emptyset$, or $v \in V_{1-i}$ and $vE \subseteq W$ then $v \in W$. Hence, the i-attractor of U contains all nodes from which player i can move "towards" U and player $1 - i$ *must* move "towards" U. The i-attractor of U is denoted by $Attr_i(G, U)$. Let A be an arbitrary attractor set. The game $G \setminus A$ is the game restricted to the nodes $V \setminus A$, i.e. $G \setminus A = (V \setminus A, V_0 \setminus A, V_1 \setminus A, E \setminus (A \times V \cup V \times A), \varOmega_{|V \setminus A})$. It is worth observing that the totality of $G \setminus A$ is ensured from A being an attractor.

Formally, for all $k \in \mathbb{N}$, the i-attractor is defined as follows:

$$\begin{aligned}
&Attr_i^0(U) = U \ ; \\
&Attr_i^{k+1}(U) = Attr_i^k(U) \,\cup\, \{v \in V_i \mid \exists\, w \in Attr_i^k(U) \; s.t. \; vEw\} \\
&\qquad\qquad\qquad \cup\, \{v \in V_{1-i} \mid \forall\, w \,:\, vEw \implies w \in Attr_i^k(U)\} \, ; \\
&Attr_i(U) = \textstyle\bigcup_{k \in \mathbb{N}} Attr_i^k(U).
\end{aligned}$$

3 The Zielonka Recursive Algorithm

In this section, we describe the Zielonka Recursive Algorithm using the basic concepts introduced in the previous sections and some observations regarding its implementation in PGSolver.

The algorithm to solve parity games introduced by Zielonka comes from a work of McNaughton [27]. The Zielonka Recursive Algorithm, as reported in Fig. 1, uses a divide and conquer technique. It constructs the winning sets for both players using the solution of subgames. It removes the nodes with the highest priority from the game, together with all nodes (and edges) attracted to this set. The algorithm *win*(G) takes as input a graph G and, after a number of recursive calls over ad hoc built subgames, returns the winning sets (W_0, W_1) for player 0 and player 1, respectively. The running time complexity of the Zielonka Recursive Algorithm is reported in Table 1.

```
function win(G):
  if $V == \emptyset$:
    $(W_0, W_1) = (\emptyset, \emptyset)$
  else:
    d = maximal priority in G
    $U = \{v \in V \mid priority(v) = d\}$
    p = d % 2
    j = 1 − p
    $A = Attr_p(U)$
    $(W_0^{'}, W_1^{'}) = $ win$(G \setminus A)$
    if $W_j^{'} == \emptyset$:
      $W_p = W_p^{'} \cup A$
      $W_j = \emptyset$
    else:
      $B = Attr_j(W_1^{j})$
      $(W_0^{'}, W_1^{'}) = $ win$(G \setminus B)$
      $W_p = W_p^{'}$
      $W_j = W_j^{'} \cup B$
  return $(W_0, W_1)$
```

Fig. 1. Zielonka Recursive Algorithm

3.1 The Implementation of the Zielonka Algorithm in PGSolver

PGSolver turns out to be of a very limited application in several real scenarios. In more details, even using the Zielonka Recursive Algorithm (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 beginnings with the way the (Classic) Recursive Algorithm 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 calculates the number of successors for the opponent player in *every* iteration when looping through every node in the attractor.

4 The Improved Algorithm

```
function win (G):
   T = G.transpose()
   Removed = {}
   return winI($G, T, Removed$)

function winI($G, T, Removed$):
    if $|V| == |Removed|$:
       return ($\emptyset$, $\emptyset$)
    $W = (\emptyset, \emptyset)$
    d = maximal priority in G
    $U = \{ v \in V \mid priority(v) = d \}$
    p = d % 2
    j = 1 − p
    $W' = (\emptyset, \emptyset)$
    $A$ = Attr($G, T, Removed, U, p$)
    ($W'_0, W'_1$) = winI($G, T, Removed \cup A$)
    if $W'_j == \emptyset$:
       $W_p = W'_p \cup A$
       $W_j = \emptyset$
    else:
       $B$ = Attr($G, T, Removed, W'_j, j$)
       ($W'_0, W'_1$) = winI($G, T, Removed \cup B$)
       $W_p = W'_p$
       $W_j = W'_j \cup B$
    return ($W_0, W_1$)
```

Fig. 2. Improved Recursive Algorithm

In this section we introduce an improved version based on the Classic Recursive Algorithm by Zielonka. The new algorithm is depicted in Fig. 2. In Fig. 3 we also report an improved version of the attractor function that the new algorithm makes use of.

Let G be a graph. Removing a node from G and building the transposed graph takes time $\Theta(|V| + |E|)$. Thus dealing with dense graph such 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$G$ ensuring that every recursive call uses 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*.

```
function Attr(G, T, Removed, A, i):
   tmpMap = []
   for x = 0 to $|V|$:
      if $x \in A$ tmpMap = 0
      else tmpMap = −1
   index = 0
   while $index < |A|$:
      for $v_0 \in adj(T, A[index])$:
         if $v_0 \notin Removed$:
            if tmpMap[$v_0$] == −1:
               if player($v_0$) == i:
                  $A = A \cup v_0$
                  tmpMap[$v_0$] = 0
               else:
                  adj_counter = −1
                  for $x \in adj(G, v_0)$:
                     if ($x \notin Removed$):
                        adj_counter += 1
                  tmpMap[$v_0$] = adj_counter
                  if adj_counter == 0:
                     $A = A \cup v_0$
            else if (player($v_0$) == j
                  and tmpMap[$v_0$] > 0):
               tmpMap[$v_0$] −= 1
               if tmpMap[$v_0$] == 0:
                  $A = A \cup v_0$
   return A
```

Fig. 3. Improved Recursive Attractor

The improved algorithm is capable of checking if 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 the Classic Recursive Algorithm. 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 complicance of complementing automata [33]. 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 per-operation cost floats around one hundred processor cycles [18]. Through these years many efforts have been made to improve memory allocation writing custom allocators from scratch, a process known to be difficult and error prone [5,6].

4.1 Implementation in OCaml for PGSolver

Our implementation of the Improved Recursive Algorithm, listed in Fig. 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.

The Improved Recursive Algorithm, named *solver*, takes three parameters: the Graph, its transposed one, and a set of excluded nodes. Our Improved Attractor function, uses a *HashMap*, called *tempMap* 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 that *solver* passes to the next recursive call.

```
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 (sol_0,sol_1) = win game
                             tgraph exc2 in
         w1.(0) <- sol_0;
         w1.(1) <- sol_1;
         w.(p) <- w1.(p) ;
         w.(j) <- (InteSet.union w1.(j) attr_B);
    )
  );
  (w.(0),w.(1))
;;
```

Fig. 4. Improved Recursive in OCaml

5 Scala

Scala [29,30] 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, also 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. Historically, the first generation of the JVM was entirely an interpreter; nowadays the JVM uses a Just-In-Time (*JIT*) compiler, a complex process aimed to improve performance at runtime. This process can be described in three steps: (1) source files are compiled by the Scala Compiler into Java Bytecode, that will be feed to a JVM; (2) the JVM will load the compiled classes at runtime and execute proper computation using an interpreter; (3) the JVM will analyze the application method calls and compile the bytecode into native machine code. This step is done in a lazy manner: the JIT compiles a code path when it knows that is about to be executed. JIT removed the overhead of interpretation and allows programs to start up quickly, in addition this kind of compilation has to be fast to prevent influencing the actual performance of the program. Another interesting aspect of the JVM is that it verifies every class file after loading them. This makes sure that the execution step does not violate some defined safety properties. The checks are performed by the verifier that includes a complete type checking of the entire program. The JVM is also available on all major platforms and compiled Java executables can run on all of them with no need for recompilation. The Scala compiler *scalac* compiles a Scala program into Java class files. The compiler is organized in a sequence of successive steps. The first one is called *the front-end step* and performs an analysis of the input file, makes sure that is a valid Scala program and produces an attributed abstract syntax tree (*AST*); the

```
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))
}
```

Fig. 5. Improved Algorithm in Scala

back-end step simplifies the AST and proceeds to the generation phase where it produces the actual class files, which constitute the final output. Targeting the JVM, the Scala Compiler checks that the produced code is type-correct in order to be accepted by the JVM bytecode verifier.

In [20], published by *Google*, Scala even being an high level language, performs just 2.5x slower than C++ machine optimized code. In particular it has been proved to be even 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*".

5.1 Improved Algorithm in Scala

In this section we introduce our implementation of the Improved Recursive Algorithm in Scala, listed as Figs. 5 and 6.

```
def Attr(G: GraphWithSets,
   A: ArrayBuffer[Int], i: Int)
   : ArrayBuffer[Int] = {
  val tmpMap = Array
    .fill[Int](G.nodes.size)(-1)
  var index = 0
  A.foreach(tmpMap(_) = 0)
  while (index < A.size) {
    G.nodes(A(index))
      .<  .foreach(v0 => {
        if (!G.exclude(v0)) {
          val flag = G.nodes(v0).player == i
          if (tmpMap(v0) == -1) {
            if (flag) {
              A += v0
              tmpMap(v0) = 0
            } else {
              val tmp = G.nodes(v0)
                .~>
                .count(x => !G.exclude(x)) - 1
              tmpMap(v0) = tmp
              if (tmp == 0) A += v0
            }
          } else if (!flag && tmpMap(v0) > 0){
            tmpMap(v0) -= 1
            if (tmpMap(v0) == 0) A += v0
          }
        }
    })
    index += 1
  }
  A
}
```

Fig. 6. Improved Attractor in Scala

Aiming at performance optimizations we use a *priority HashMap* where every *key* is a certain priority and a *value* is a set of each node v where $priority(v) = key$. As fast and JVM-Optimized *HashMaps* and *ArrayLists* we use the ones included in the open source library *Trove*. In addition, using the well known *strategy pattern* [17] we open the framework for further extensions and improvements. The intended purpose of our algorithm is to assert that the performance of existing tools for solving parity games can be improved using the improved algorithm and choosing Scala as the programming language. 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; the data structure 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.

The Attractor function makes deeply use of an array of integers named *tmpMap* that is preallocated 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, to serve this purpose, 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. These functions are inside a singleton object called *ImprovedRecursiveSolver* that extends the *Solver* interface.

6 Benchmarks

In this section we study, analyze and evaluate the running time of our four 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. We want to note that *IRS* does not apply any preprocessing steps to the arena before solving. All tests have been run on an Intel(R) Xeon(R) CPU E5620 @ 2.40 GHz, with 16 GB of Ram (with no Swap available) running Ubuntu 14.04. 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

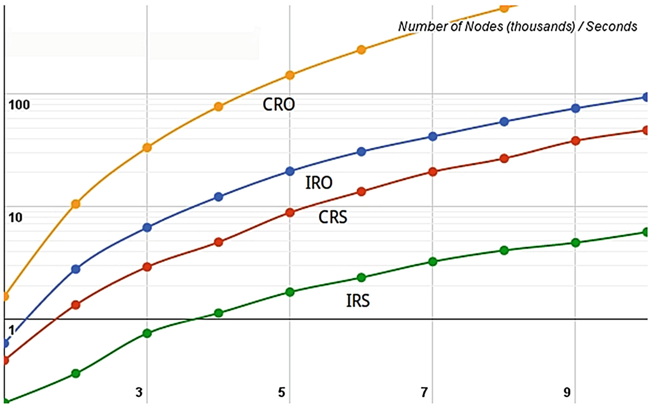

Fig. 7. Random Games Chart in Logarithmic Scale

this section we define abo_T when the program has been aborted due to excessive time and abo_M when the program has been killed by the Operating System due to memory consumption. In Fig. 7 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.

N nodes, N colors, $adj(\frac{N}{2}, N)$

N	IRS	CRO	CRS	IRO
1×10^3	**0.204**	1.99	0.505	0.752
2×10^3	**0.456**	13.208	1.918	3.664
3×10^3	**1.031**	41.493	2.656	6.147
4×10^3	**1.879**	96.847	6.728	15.966
5×10^3	**2.977**	183.589	12.616	27.272
6×10^3	**3.993**	306.104	19.032	41.051
7×10^3	**4.989**	486.368	27.05	50.367
8×10^3	**6.103**	abo_T	36.597	70.972
9×10^3	**7.287**	abo_T	55.171	97.216
10×10^3	**8.468**	abo_T	68.303	113.36

N nodes, N colors, $adj(1, N)$

N	IRS	CRO	CRS	IRO
1×10^3	**0.179**	1.21	0.454	0.583
2×10^3	**0.389**	8.075	1.173	2.366
3×10^3	**0.868**	25.097	2.656	6.147
4×10^3	**1.279**	57.186	4.23	10.452
5×10^3	**2.273**	108.983	9.206	20.377
6×10^3	**2.772**	183.884	12.562	27.489
7×10^3	**3.748**	291.077	17.942	37.521
8×10^3	**3.942**	418.377	22.105	47.502
9×10^3	**4.989**	593.721	23.93	61.593
10×10^3	**6.413**	abo_T	42.408	80.508

N nodes, 2 colors, $adj(\frac{N}{2}, N)$

N	IRS	CRO	CRS	IRO
1×10^3	**0.189**	1.98	0.481	0.702
2×10^3	**0.469**	12.941	1.55	3.17
3×10^3	**1.046**	41.584	3.995	7.428
4×10^3	**1.712**	96.545	5.378	13.823
5×10^3	**2.414**	181.225	11.273	22.575
6×10^3	**3.458**	307.233	16.472	35.269
7×10^3	**4.612**	484.159	26.448	49.311
8×10^3	**6.003**	abo_T	28.968	65.674
9×10^3	**7.03**	abo_T	43.666	85.909
10×10^3	**8.938**	abo_T	57.18	110.814

N nodes, 2 colors, $adj(1, N)$

N	IRS	CRO	CRS	IRO
1×10^3	**0.159**	1.226	0.385	0.468
2×10^3	**0.341**	7.965	1.004	2.162
3×10^3	**0.797**	25.114	2.305	6.014
4×10^3	**1.123**	56.422	3.699	9.421
5×10^3	**1.704**	108.584	6.12	14.971
6×10^3	**2.243**	182.935	10.099	22.621
7×10^3	**3.324**	286.503	13.898	32.335
8×10^3	**3.95**	430.265	19.743	44.281
9×10^3	**4.597**	abo_T	28.742	56.81
10×10^3	**5.651**	abo_T	33.639	71.434

N nodes, $\sqrt{N}$ colors, $adj(\frac{N}{2}, N)$

N	IRS	CRO	CRS	IRO
1×10^3	**0.204**	1.978	0.468	0.71
2×10^3	**0.456**	13.114	1.575	3.203
3×10^3	**1.031**	41.493	3.868	7.492
4×10^3	**1.621**	96.55	5.744	13.97
5×10^3	**2.439**	183.589	10.72	22.98
6×10^3	**3.372**	307.426	15.978	34.78
7×10^3	**4.662**	485.826	26.432	48.875
8×10^3	**6.499**	abo_T	34.741	66.423
9×10^3	**7.147**	abo_T	48.915	86.645
10×10^3	**8.988**	abo_T	56.656	111.492

N nodes, $\sqrt{N}$ colors, $adj(1, N)$

N	IRS	CRO	CRS	IRO
1×10^3	**0.162**	1.218	0.384	0.475
2×10^3	**0.344**	7.947	1.034	2.195
3×10^3	**0.788**	25.029	2.406	5.944
4×10^3	**1.105**	57.307	3.835	9.608
5×10^3	**1.678**	108.623	6.34	15.165
6×10^3	**2.281**	182.154	9.871	22.859
7×10^3	**3.193**	285.28	14.338	32.536
8×10^3	**4.185**	422.74	20.362	44.515
9×10^3	**5.009**	599.071	24.347	57.022
10×10^3	**5.76**	abo_T	35.024	72.291

6.1 Trends Analysis for Random Arenas

The speedup obtained by our implementation of the Improved Recursive Algorithm is in most cases quite noticeable. Figure 8 shows the running time trend for Improved and Classic Algorithm on each platform. The seconds are limited to [0, 100]. 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.

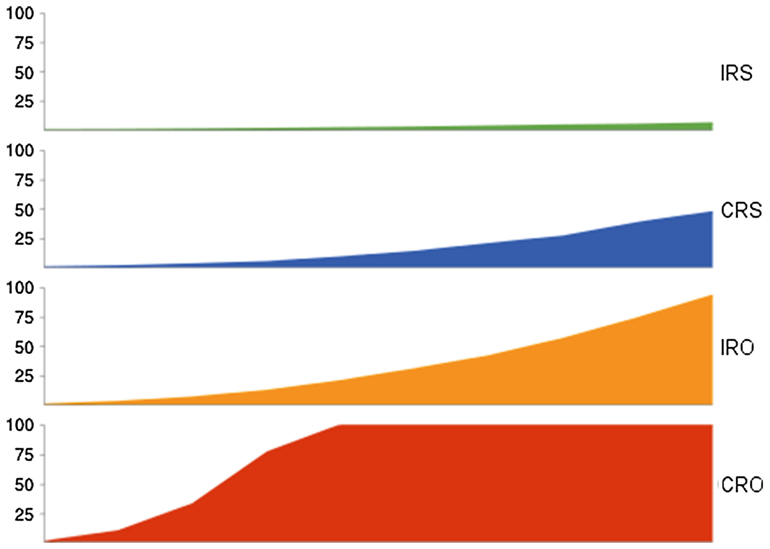

Fig. 8. Trends Chart

6.2 Trends Analysis for Special Games

Focusing on Classic Recursive in PGSolver and our Improved Recursive in Scala, here we show the running times for non-random games generated by PGSolver. In particular we use four types of non-random games, these experiments have been run against PGSolver using the Classic Recursive Algorithm with all optimizations disabled and all solutions were matched to ensure correctness.

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 below reports the running time for our experiments and these results are drawn in Fig. 9.

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	abo_T	abo_M

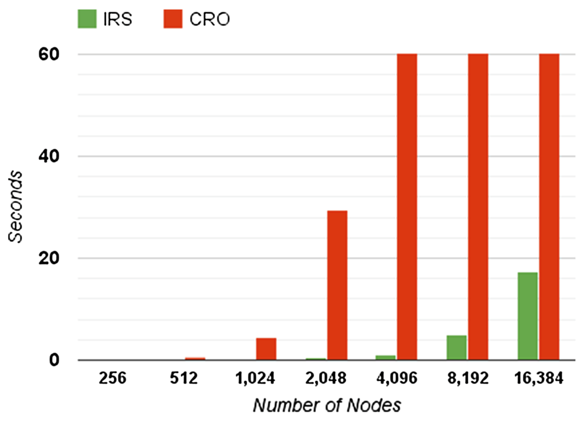

Fig. 9. Clique Trends

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 $7 \leq k \leq 19$ and report our experiments in the table below whose trend is drawn in Fig. 10. Figure 10 shows better performance for *CRO* than *IRS* using low-scaled values as input parameter. This behaviour is not surprising as there is a *warming-up* time required by the Java Virtual Machine.

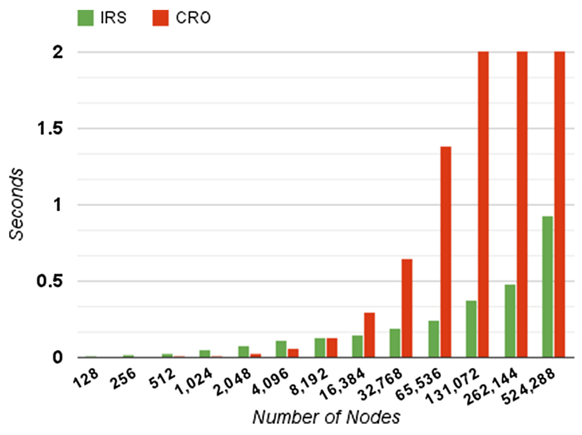

Fig. 10. Ladder Trends

n	2^7	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	0.01	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	0.00	0.01	0.01	0.03	0.06	0.13	0.3	0.65	1.39	2.93	6.21	11.71

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, $V_1 = V$ 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 $7 \leq k \leq 19$, report our experiments in the table below and draw the trends in Fig. 11.

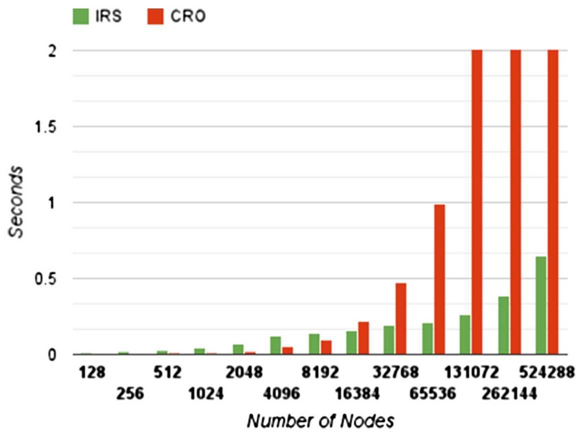

Fig. 11. Model Checker Ladder Trends

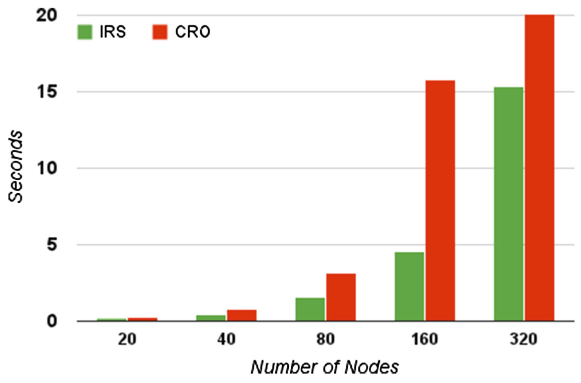

Fig. 12. Jurdiznski Trends with a Fixed Parameter of $n = 10$ Layers

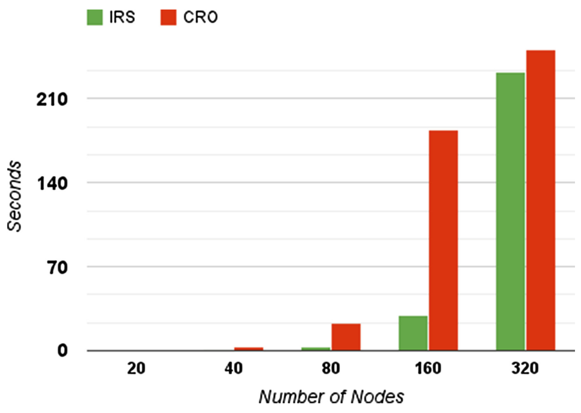

Fig. 13. Jurdiznski Trends with a Fixed Parameter of $m = 10$ Blocks

n	2^7	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	0.01	0.02	0.03	0.04	0.07	0.12	0.14	0.16	0.19	0.21	0.26	0.39	0.65
CRO	0.00	0.00	0.01	0.01	0.02	0.05	0.10	0.22	0.47	0.99	2.12	4.16	8.31

Jurdzinski[n, m] games are designed to generate the worst-case behaviour for the Small Progress Measure Solver [22]. The parameter n is the number of layers, where each layer has m repeating blocks that are inter-connected as described in [22]. 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 $k = 1, 2, 3, 4, 5$ and one where m is fixed to 10 and $n = 10 \times 2^k$, for $k = 1, 2, 3, 4, 5$. The results of our experiments are reported in the tables below. The trends are drawn in Figs. 12 and 13.

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	abo_T

7 Conclusions

PGSolver is a well-stablished framework that collects multiple algorithms to decide parity games. For several years now this platform has been the only one available to solve and benchmark in practice. Given PGSolver's limitations addressing huge graphs, several attempts of improvement have been carried out recently. Some of them have been implemented as preprocessing steps in the tool itself (such as priority compression or SCC decomposition and the like), while others chose parallelism techniques, such as Cuda [19], applied to the algorithms. However these improvements often do not show the desired performance.

In this paper we started from scratch by revisiting the Zielonka Recursive Algorithm, implemented an improved and the classic versions in Scala and

OCaml, comparing among them. The choice of Scala as a programming language has been not casual, but rather it comes out from a deep study focused on performance and simplicity. Scala is interoperable with Java libraries, has a concise and clear syntax, functional and object oriented features, runs on the Java Virtual Machine and has been proven to be high performing. Our main result is a new and fast tool for solving parity games capable of gaining up to two orders of magnitude in running time. In conclusion we state that there is place for a faster and better framework to solve parity games and this work is a starting point raising several interesting questions. For example, what if one implements the other known algorithms to solve parity games in Scala? PGSolver showed that Zielonka's algorithm is the best performing. Can one reproduce the same results in Scala? We leave all these questions as future work.

References

1. Aminof, B., Mogavero, F., Murano, A.: Synthesis of hierarchical systems. Sci. Comp. Program. **83**, 56–79 (2013)
2. Antonik, A., Charlton, N., Huth, M.: Polynomial-time under-approximation of winning regions in parity games. ENTCS **225**, 115–139 (2009)
3. Aminof, B., Kupferman, O., Murano, A.: Improved model checking of hierarchical systems. Inf. Comput. **210**, 68–86 (2012)
4. Barringer, H., Havelund, K.: TraceContract: a scala DSL for trace analysis. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 57–72. Springer, Heidelberg (2011)
5. Berger, E.D., Zorn, B.G., McKinley, K.S.: Composing high-performance memory allocators. ACM SIGPLAN Not. **36**, 114–124 (2001)
6. Berger, E.D., Zorn, B.G., McKinley, K.S.: OOPSLA 2002: reconsidering custom memory allocation. ACM SIGPLAN Not. **48**(4), 46–57 (2013)
7. Berwanger, D.: Admissibility in infinite games. In: Thomas, W., Weil, P. (eds.) STACS 2007. LNCS, vol. 4393, pp. 188–199. Springer, Heidelberg (2007)
8. Chatterjee, K., Doyen, L., Henzinger, T.A., Raskin, J.-F.: Generalized mean-payoff and energy games. In: FSTTCS'10, LIPIcs 8, pp. 505–516 (2010)
9. Chatterjee, K., Henzinger, T.A., Jurdzinski, M.: Mean-payoff parity games. In: LICS'05, pp. 178–187 (2005)
10. Chatterjee, K., Jurdzinski, M., Henzinger, T.A.: Quantitative stochastic parity games. In: SODA'04, pp. 121–130 (2004)
11. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Kozen, D. (ed.) Logic of Programs 1981. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1982)
12. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (2002)
13. Emerson, E., Jutla, C.: Tree automata, μ-calculus and determinacy. In: FOCS'91, pp. 368–377 (1991)
14. Friedmann, O.: Recursive algorithm for parity games requires exponential time. RAIRO-Theor. Inform. Appl. **45**(04), 449–457 (2011)
15. Friedmann, O., Lange, M.: The pgsolver collection of parity game solvers. University of Munich (2009)
16. Friedmann, O., Lange, M.: Solving parity games in practice. In: Liu, Z., Ravn, A.P. (eds.) ATVA 2009. LNCS, vol. 5799, pp. 182–196. Springer, Heidelberg (2009)

17. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Pearson Education, New Jersey (1994)
18. Gay, D., Aiken, A.: Memory management with explicit regions. ACM Sigplan Not. **33**, 313–323 (1998)
19. Hoffmann, P., Luttenberger, M.: Solving parity games on the GPU. In: Van Hung, D., Ogawa, M. (eds.) ATVA 2013. LNCS, vol. 8172, pp. 455–459. Springer, Heidelberg (2013)
20. Hundt, R.: Loop recognition in c++/java/go/scala. In: 2011 Proceedings of Scala Days (2011)
21. Jurdzinski, M.: Deciding the winner in parity games is in up $\cap$ co-up. Inf. Process. Lett. **68**(3), 119–124 (1998)
22. Jurdziński, M.: Small progress measures for solving parity games. In: Reichel, H., Tison, S. (eds.) STACS 2000. LNCS, vol. 1770, pp. 290–301. Springer, Heidelberg (2000)
23. Jurdzinski, M., Paterson, M., Zwick, U.: A deterministic subexponential algorithm for solving parity games. SIAM J. Comput. **38**(4), 1519–1532 (2008)
24. Kozen, D.: Results on the propositional mu-calculus. TCS **27**(3), 333–354 (1983)
25. Kupferman, O., Vardi, M., Wolper, P.: An automata theoretic approach to branching-time model checking. JACM **47**(2), 312–360 (2000)
26. Kupferman, O., Vardi, M., Wolper, P.: Module checking. IC **164**(2), 322–344 (2001)
27. McNaughton, R.: Infinite games played on finite graphs. Ann. Pure Appl. Log. **65**(2), 149–184 (1993)
28. Mogavero, F., Murano, A., Sorrentino, L.: On promptness in parity games. In: McMillan, K., Middeldorp, A., Voronkov, A. (eds.) LPAR-19 2013. LNCS, vol. 8312, pp. 601–618. Springer, Heidelberg (2013)
29. Odersky, M., Altherr, P., Cremet, V., Emir, B., Maneth, S., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., Zenger, M.: An overview of the scala programming language (2004)
30. Odersky, M., Spoon, L., Venners, B.: Programming in Scala. Artima Inc, Sunnyvale (2008)
31. Queille, J., Sifakis, J.: Specification and verification of concurrent programs in CESAR. In: Dezani-Ciancaglini, M., Montanari, U. (eds.) International Symposium on Programming. LNCS, vol. 137, pp. 337–351. Springer, Heidelberg (1982)
32. Schewe, S.: Solving parity games in big steps. In: Arvind, V., Prasad, S. (eds.) FSTTCS 2007. LNCS, vol. 4855, pp. 449–460. Springer, Heidelberg (2007)
33. Thomas, W.: Automata on infinite objects. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, vol. B, pp. 133–191. MIT Press, Cambridge (1990)
34. Vöge, J., Jurdzinski, M.: A discrete strategy improvement algorithm for solving parity games. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 202–215. Springer, Heidelberg (2000)
35. Zielonka, W.: Infinite games on finitely coloured graphs with applications to automata on infinite trees. Theor. Comput. Sci. **200**(1–2), 135–183 (1998)