Solving Parity Games in Practice

Oliver Friedmann and Martin Lange

Dept. of Computer Science, University of Munich, Germany

Abstract. Parity games are 2-player games of perfect information and infinite duration that have important applications in automata theory and decision procedures (validity as well as model checking) for temporal logics. In this paper we investigate practical aspects of solving parity games. The main contribution is a suggestion on how to solve parity games efficiently in practice: we present a generic solver that intertwines optimisations with any of the existing parity game algorithms which is only called on parts of a game that cannot be solved faster by simpler methods. This approach is evaluated empirically on a series of benchmarking games from the aforementioned application domains, showing that using this approach vastly speeds up the solving process. As a side-effect we obtain the surprising observation that Zielonka's recursive algorithm is the best parity game solver in practice.

1 Introduction

Parity games are two-player games of perfect information played on directed graphs whose nodes are labeled with priorities. The winner of a play is determined by the parity (even or odd) of the *maximal* priority occurring infinitely often. Parity games have various applications in computer science, and the theory of formal languages and automata in particular. They are closely related to other games of infinite duration, in particular mean and discounted payoff as well as stochastic games [4,14]. An efficient parity game solver may be extendable to efficient solvers for those games as well.

Solving a parity game is equivalent (from a complexity-theoretic point of view) under linear-time reductions to the model checking problem for the modal μ-calculus [14]. Hence, any parity game solver is also a model checker for the μ-calculus (and vice-versa) and all its fragments like CTL, PDL, CTL*, etc. However, typical verification problems result in parity games with few priorities only for which specialised algorithms should be more efficient than a general solver.

Parity games also arise in decision procedures for temporal logics. While the satisfiability problem for linear-time logics like LTL, PSL or the linear-time μ-calculus reduces – in one form or the other – to the inclusion problem for nondeterministic Büchi automata (NBA) and therefore requires *complementation* thereof, branching-time logics require the determinisation of NBA in addition. So far, the only known constructions for determinising and complementing an NBA are Safra's [10], Piterman's [9], and Kähler and Wilke's [7]. The first one

Z. Liu and A.P. Ravn (Eds.): ATVA 2009, LNCS 5799, pp. 182–196, 2009.

yields a deterministic Streett automaton which is algorithmically not very easy to handle. The two others yield parity automata. Using these, the satisfiability (or validity) problem for branching-time logics not only reduces to the solving of parity games, there also does not seem to be a feasible alternative. The same holds for controller synthesis problems which are tackled by a reduction to the satisfiability problem of, typically, some branching-time logic like the modal μ-calculus [2]. Hence, being able to solve parity games well in practice is also vital for obtaining good satisfiability and controller synthesis tools.

A variety of algorithms for solving parity games has been invented so far. The most prominent deterministic ones are the constructive proof of memoryless determinacy by Zielonka [18] which yields a recursive algorithm, the local μ-calculus model checker by Stevens and Stirling [13], Jurdziński's small progress measures algorithm [5] with a symbolic version [8], the strategy improvement algorithm by Jurdziński and Vöge [17] with a locally optimal variation by Schewe [12], and the subexponential algorithm by Jurdziński, Paterson and Zwick [6] with a so-called big-step variant by Schewe [11]. This variety is owed to the theoretical challenge of answering the question whether parity games can be solved in polynomial time, rather than practical motivations. Nonetheless, a parity game solver that is efficient in practice is necessary for practical decision procedure for branching-time logics and for controller synthesis, and may even be used as a model checker. Van de Pol and Weber describe a parallel implementation of Jurdziński's small progress measures algorithm [16] but it turns out that in many cases, this algorithm is not the most efficient one. Also, their implementation does not feature known tricks that are supposed to be optimisations to any parity game solver.

The literature contains a few suggestions on how to tune a parity game solver. Jurdziński [5] mentions decomposition into SCCs and solving SCC-wise, removal of self-cycles on nodes, and priority compression. Huth et al. [1] mention the latter two and, in addition, priority propagation. In any case, these are suggested heuristics that have not been put to the test yet. While it is plausible that they are useful in speeding up parity game solvers in practice, no proper evidence of this has been given so far.

In this paper we present a rigorous empirical treatment of these optimisations. After recalling the theory of parity games in Sect. 2, we shortly describe such optimisations in Sect. 3 and devise a so-called generic solver. It is an algorithm which employs some of these optimisations in a certain order and fashion, and intertwines them with calls to a real algorithm for solving parity games. The choice of the optimisations and the design of the order etc. is motivated by common sense *and* experience in practice. Hence, this paper presents a particular way of employing particular optimisations that has turned out to be successful while others are less succesful (or even harmful). The success is quantified in Sect. 4 which examines the result that employing these optimisations has on the times needed to solve certain games. There are no families of games that people agree on as standard benchmarks. We therefore use hand-made games, some of which are taken from application domains listed above.

The approach presented in this paper is implemented in a publicly available tool which shows that parity games can – despite the lack of proof or fact about being polynomial-time solvable – be solved efficiently in practice. It also bears some surprises. All of the deterministic algorithms for solving parity games that have appeared in the literature so far have been implemented in the tool which therefore allows them to be compared w.r.t. their usability in practice. As a result, the small progress measures algorithm as well as the strategy improvement turn out to be generally slower than the recursive algorithm. This is a huge surprise since this algorithm was commonly accepted to be a constructive proof of determinacy but nothing more, in particular not to have any practical relevance at all. Furthermore, there are "optimisations" that have been suggested as a means for speeding up the solving which one should not employ because they turn out to slow down the solving.

The rest of the paper is organised as follows. Sect. 2 recalls parity games and necessary technicalities. Sect. 3 describes the aforementioned optimisations and presents the generic solver which is assembled out of these. Sect. 4 evaluates the solver empirically on some families of benchmarking games.

2 Preliminaries

A *parity game* is a tuple $G = (V, V_0, V_1, E, \Omega)$ where (V, E) forms a directed graph in which each node has at least one successor. The set of nodes is partitioned into $V = V_0 \cup V_1$ with $V_0 \cap V_1 = \emptyset$, and $\Omega : V \to \mathbb{N}$ is the *priority function* that assigns to each node a natural number called the *priority* of the node. We write $|\Omega|$ for the index of the parity game, that is the number of different priorities assigned to its nodes. The graph is required to be total, i.e. for every $v \in V$ there is a $w \in W$ s.t. $(v, w) \in E$. Here we only consider games based on finite graphs.

We also use infix notation vEw instead of $(v, w) \in E$ and define the set of all *successors* of v as $vE := \{w \mid vEw\}$, as well as the set of all *predecessors* of w as $Ew := \{v \mid vEw\}$. For a set $U \subseteq V$ and nodes $v, w \in V$ we will write $G \setminus U$ for the game that is obtained from G by eliminating all nodes in U, i.e. $(V \setminus U, V_0 \setminus U, V_1 \setminus U, E \setminus (V \times U \cup U \times V), \Omega)$ and $G \setminus \{(v, w)\}$ for the game that results from eliminating a possible edge between v and w – assuming that the result is still total – i.e. $(V, V_0, V_1, E \setminus \{(v, w)\}, \Omega)$.

The game is played between two players called 0 and 1 in the following way. Starting in a node $v_0 \in V$ they construct an infinite path through the graph as follows. If the construction so far has yielded a finite sequence $v_0 \dots v_n$ and $v_n \in V_i$ then player i selects a $w \in v_n E$ and the play continues with the sequence $v_0 \dots v_n w$.

Every play has a unique winner given by the *parity* of the greatest priority that occurs infinitely often in a play. The winner of the play $v_0 v_1 v_2 \dots$ is player i iff $\max\{p \mid \forall j \in \mathbb{N}\ \exists k \geq j : \Omega(v_k) = p\} \equiv_2 i$ (where $i \equiv_2 j$ holds iff $|i - j| \mod 2 = 0$). That is, player 0 tries to make an even priority occur infinitely often without any greater odd priorities occurring infinitely often, player 1 attempts the converse.

A *positional strategy* for player i in G is a – possibly partial – function $\sigma : V_i \to V$. A play $v_0 v_1 \ldots$ *conforms* to a strategy σ for player i if for all $j \in \mathbb{N}$ we have: if $v_j \in V_i$ then $v_{j+1} = \sigma(v_j)$. Intuitively, conforming to a strategy means to always make those choices that are prescribed by the strategy. A strategy σ for player i is a *winning strategy* in node v if player i wins every play that begins in v and conforms to σ. We say that player i *wins* the game G starting in v iff he/she has a winning strategy for G starting in v.

With G we associate two sets $W_0, W_1 \subseteq V$; W_i is the set of all nodes v s.t. player i wins the game G starting in v. Here we restrict ourselves to positional strategies because it is well-known that a player has a (general) winning strategy iff she has a positional winning strategy for a given game. In fact, parity games enjoy positional determinacy meaning that for every node v in the game either $v \in W_0$ or $v \in W_1$ [3]. Furthermore, it is not difficult to show that, whenever player i has winning strategies σ_v for all $v \in U$ for some $U \subseteq V$, then there is also a single strategy σ that is winning for player i from every node in U.

The problem of solving a given parity game is to compute W_0 and W_1 as well as corresponding winning strategies σ_0 and σ_1 for the players on their respective winning regions. We will write $[]$ for the strategy with empty domain, and $\sigma[v \mapsto w]$ with vEw for the strategy that behaves like σ on all nodes in $V \setminus \{v\}$ and that maps v to w. Given two strategies σ, σ' for player i, we define their right-join $\sigma \nleftarrow \sigma'$ as $(\sigma \nleftarrow \sigma')(v) = \sigma(v)$ if $\sigma'(v)$ is undefined and $(\sigma \nleftarrow \sigma')(v) = \sigma'(v)$ otherwise.

Let $U \subseteq V$ and $i \in \{0, 1\}$. The i-attractor of U contains all nodes from which player i can move "towards" U and player $1 - i$ must move "towards" U. Attractors will play an important role in the solving procedure described below because they can efficiently be computed using breadth-first search on the inverse graph underlying the game. At the same time, it is possible to construct an *attractor strategy* which is a positional strategy in a reachability game. Following this strategy guarantees player i to reach a node in U eventually, regardless of the opponent's choices. Define, for all $k \in \mathbb{N}$:

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

Note that any attractor on a finite game is necessarily finite, and the approximation defined above thus terminates after at most $|V|$ many steps. It is also not difficult to see that $Attr_i(U)$ can be computed in time $\mathcal{O}(|E|)$ for any i and U if the set operations take constant time only, using boolean arrays for example. The corresponding attractor strategy is defined as

$$\sigma_i^{Attr}(v) := \begin{cases} w, & \text{if there is } k > 0 \text{ s.t. } v \in (V_i \cap Attr_i^k(U)) \setminus Attr_i^{k-1}(U) \\ & \text{and } w \in Attr_i^{k-1}(U) \cap vE \\ \bot, & \text{otherwise} \end{cases}$$

Note that the choice of w is not unique, but any w with the prescribed property will suffice.

An important property that has been noted before [18,14] is that removing the i-attractor of any set of nodes from a game will still result in a total game graph.

3 Universal Optimisations and a Generic Solver

This section describes some universal optimisations – in the form of pre-transformations or incomplete solvers. These try to efficiently reduce the overall complexity of a given parity game in order to reduce the effort spent by any solver. Clearly, such optimisations have to ensure that a solution of the modified game can be effectively and efficiently translated back into a valid solution of the original game. Here we describe four: (1) SCC decomposition [5]; (2) detection of three special cases – two from [1]; (3) priority compression [5], and (4) priority propagation [1]. At the end we present a generic solver that makes use of most of them. The choice is motivated by two facts: their worst-case running time is low in comparison to that of a real solver: at most $\mathcal{O}(|\Omega| \cdot |E|)$, resp. $\mathcal{O}(|V| \log |V|)$. More importantly, they are empirically found to be beneficial.

3.1 SCC Decomposition

Let $G = (V, V_0, V_1, E, \Omega)$ be a parity game. A *strongly connected component* (SCC) is a non-empty set $C \subseteq V$ with the property that every node in C can reach every other node in C, i.e. uE^*v for all $u, v \in C$ (where E^* denotes the reflexive-transitive closure of E). We always assume SCCs to be maximal. We call an SCC C *proper* if $|C| > 1$ or $C = \{v\}$ for some v with vEv. Every parity game $G = (V, V_0, V_1, E, \Omega)$ can, in time $\mathcal{O}(|E|)$, be partitioned into SCCs $C_0, ..., C_n$ using Tarjan's algorithm for example [15].

There is a topological ordering $\rightarrow$ on these SCCs which is defined as $C_i \rightarrow C_j$ iff $i \neq j$ and there are $u \in C_i$, $v \in C_j$ with uEv. An SCC C is called *final* if there is no SCC C' s.t. $C \rightarrow C'$. Note that every finite graph must have at least one final SCC.

Parity games can be solved SCC-wise. Each play eventually gets trapped in an SCC, and the winner of the play is determined by the priorities of the nodes in this SCC alone, in particular not by priorities of nodes not in this SCC. Hence, an entire parity game can be solved by solving its SCCs starting with the final ones and working backwards in their order.

It is reasonable to assume that SCC decomposition speeds up the solving of a game. Suppose that the time it takes to solve a game G is $f(G)$, and that G can be decomposed into SCCs $C_0, \ldots, C_n$. Then solving SCC-wise will take time $f(C_1) + \ldots + f(C_n) + \mathcal{O}(|G|)$ which is asymptotically better than $f(G)$ if f is superlinear. Note that it takes at least linear time to solve a parity game because every node has to be visited at least once in order to determine which W_i it belongs to.

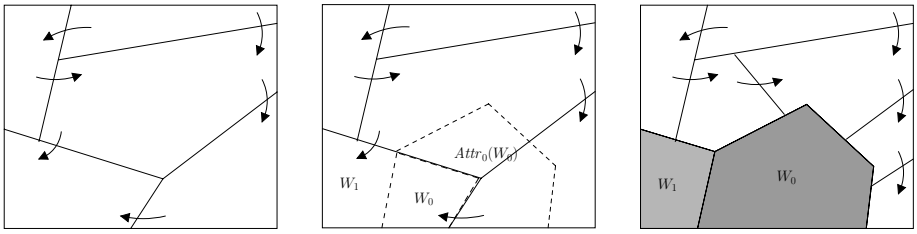

Fig. 1. Solving a game SCC-wise with refined decompositions

A naïve implementation of SCC-wise solving handles final SCCs and replaces the winning regions in those with two single self-looping nodes that are won by the respective players, and then continues with the next SCCs. A slightly more clever way is the following, as suggested by Jurdziński [5]. First, let W_i^G and σ_i^G be empty sets resp. strategies for the players $i \in \{0,1\}$ on G.

1. Decompose G into SCCs $C_0, \ldots, C_n$. W.l.o.g. say that $C_0, \ldots, C_m$ are final for some $m \leq n$. Then one solves these obtaining winning regions W_i^j and strategies σ_i^j for $i \in \{0,1\}$ and $j = 0, \ldots, m$. Add W_i^j to W_i^G for every i, j and add σ_i^j to σ_i^G via right-join.
2. Compute $A_i := Attr_i(W_i^0 \cup \ldots \cup W_i^m)$ for $i \in \{0,1\}$ and corresponding attractor strategies σ_i which are also added to W_i^G and σ_i^G via right-join.
3. Repeat step 1 with $(G \setminus A_0) \setminus A_1$ until G is entirely solved.

Note that the attractors of the winning regions in some SCC can extend into SCCs further up, and eliminating them can result in a finer SCC structure than before. Hence, it suffices to decompose those of $C_{m+1}, \ldots, C_n$ that intersect with one of the attractors.

An example is depicted in Fig. 1. On the left it shows a parity game that is decomposed into 5 SCCs of which one is final. That is then solved using an arbitrary solver which partitions it into the winning regions W_0 and W_1. The middle then shows the attractor of W_0 reaching into other SCCs. On the right it shows the shaded regions already declared as winning for the respective players, and the two affected non-final SCCs being decomposed into SCCs again. This then yields a smaller parity game with 6 SCCs which can be solved iteratively until the winning regions partition the entire game.

3.2 Detection of Special Cases

There are certain games that can be solved very efficiently. W.l.o.g. we assume games to be proper and final SCCs. Note that non-proper SCCs are being solved using attractor computations in the procedure described above.

Self-cycle games. Suppose there is a node v such that vEv. Then there are two cases depending on the node's owner p and the parity of the node's priority. If

$\Omega(v) \not\equiv_2 p$ then taking the edge (v, v) is always a bad choice for player p and this edge can be removed from the game. If v is v's only successor then v itself can be removed and the process iterated in order to preserve totality. If $\Omega(v) \equiv_2 p$ then taking this edge is always good in the sense that the partial function $[v \mapsto v]$ is a winning strategy for player p on v. Hence, its attractor can be removed as described above. It is therefore possible to remove self-cycles from a game in time $\mathcal{O}(|E|)$, returning winning sets W_i and strategies σ_i for $i \in \{0, 1\}$ that result from the attractors of those nodes that are good for the respective players.

One-parity games. If all nodes in a proper SCC have the same parity the whole game is won by the corresponding player no matter which choice she makes. Hence, a winning strategy can be found by random choice in time $\mathcal{O}(|V|)$.

One-player games. A game G is a one-player game for player i iff for all $v \in V_{1-i}$ we have $|vE| = 1$. It can be solved in time $\mathcal{O}(|\Omega| \cdot |E|)$ as follows. Consider the largest priority p in G, and let $P := \{v \mid \Omega(v) = p\}$. There are two cases.

- If $p \equiv_2 i$ then player i wins the entire G because it is assumed to be a proper SCC and player $1 - i$ does not make any choices, so she can reach a node in P from any node in the game. A winning strategy can easily be constructed from an attractor strategy for P.
- If $p \not\equiv_2$ then let $A := Attr_{1-i}(P)$. Note that A consists of all nodes from which player i has to move through a node with priority p which is bad for her. Let $C_0, \ldots, C_m$ be a decomposition of $G \setminus A$ into SCCs. Now, player i wins from all nodes in G iff she wins from all nodes in one of $C_0, \ldots, C_m$, simply because they are part of the original SCC G in which player $1 - i$ does not move, so the attractor of any winning node is always the entire G.

This gives a simple recursive algorithm which considers the largest priority and either terminates or removes attractors, decomposes into SCCs and calls itself recursively on the sub-SCCs. If player $1 - i$ does not win on the entire G, then player $1 - i$ wins on the entire G, and this is the case if in all the recursive calls no sub-SCC is won by player i. Clearly, this can be realised in time $\mathcal{O}(|\Omega| \cdot |E|)$.

3.3 Priority Compression

The complexity of a parity game rises with $|\Omega|$. This optimisation step attempts to reduce this number. Note that it is not the actual values of priorities that determine the winner. It is rather their *parity* on the one hand and their *ordering* on the other. For instance, if there are two priorities $p_1 < p_2$ in a game with $p_1 \equiv_2 p_2$ but there is no p' such that $p_1 < p' < p_2$ and $p' \not\equiv_2 p_1$ then every occurrence of p_2 can be replaced by p_1.

The *compression* of G is a partial mapping $\omega : \mathbb{N} \to \mathbb{N}$ that is defined on all $\Omega(v)$ for any node v of the underlying game G; monotonic ($x \leq y$ implies $\omega(x) \leq \omega(y)$); decreasing ($\omega(x) \leq x$); parity-preserving ($\omega(x) \equiv_2 x$); dense ($\omega(x) < \omega(y) - 1$ implies $\exists z.\omega(x) < \omega(z) < \omega(y)$); and minimal ($\min\{\omega(x) \mid \omega(x) \neq \bot\} < 2$). Note that a compression of G is unique and can easily be

computed in time $\mathcal{O}(|V| \log |V|)$: sort all nodes in ascending order of priority and then construct ω in a single sweep through this order, starting with 0 or 1 depending on the least priority in G.

If $G = (V, V_0, V_1, E, \Omega)$ and ω is its compression then $Comp(G) = (V, V_0, V_1, E, \omega \circ \Omega)$. It is the case that G and $Comp(G)$ have the same winning regions and strategies.

3.4 Priority Propagation

Suppose a play passes through a node v. Then it ultimately has to pass through *some* of its successors as well. If the priority of v is at most as high as that of all its successors, then one can replace the priority of v with the minimum of its successors' priorities without changing the winning regions and strategies. This is *backwards propagation*. Equally, in *forwards propagation* one replaces a node's priority with the minimum of the priorities of all its predecessors if that is greater than the current priority. This is sound because a node can only contribute to the determination of the winner of a play if it is visited repeatedly, i.e. if the play passes through one of its predecessors as well.

Technically, priority propagation on $G = (V, V_0, V_1, E, \Omega)$ computes a game $G = (V, V_0, V_1, E, \Omega')$ s.t. for all $v \in V$: $\Omega'(v) := \max\{\Omega(v), \min\{\Omega(w) \mid w \in U_v\}\}$ where $U_v = vE$ in case of backwards propagation and $U_v = Ev$ otherwise.

Note that propagation can be iterated, the forwards and backwards facets can be intertwined, and this process is guaranteed to terminate because priorities are at most increased but never beyond the maximal priority in G. Hence, the worst-case running time is $\mathcal{O}(|\Omega| \cdot |V| \cdot |E|)$, on average it will be much faster though.

Note that, even though priority propagation *increases* priorities, it *decreases* the range of priorities in a game, and is therefore supposedly beneficial to preceed compression, because it allows for more compressed priorities. Empirically, however, the use of priority propagation turns out to be harmful. This is why it is not considered in the generic solver presented next.

3.5 A Generic Solver

We propose to solve parity games using the following generic algorithm which takes as parameter a real solver and applies it only where necessary. It relies heavily on SCC decomposition and attractor computations. Self-cycle elimination is done first because it can only be applied once and for all. Then the game is decomposed, and from then on, only final SCCs are being solved. Their priorities are being compressed – note that compression within an SCC rather than the entire game generally leads to better results – and are checked for being special cases. If this does not solve the SCC then the parameter solver is put to work on it. Finally, attractors of computed winning regions are formed, and the SCC decomposition is refined accordingly.

```
    GenericSolver(G = (V, V_0, V_1, E, Ω), S) =
1       initialise empty winning regions W_0, W_1 and strategies σ_0, σ_1
2       eliminate self-cycles from G
3       while G is not empty do
4           decompose G into SCCs
5           for each final SCC C do
6               if C is a one-player-SCC then
7                   solve C directly
8               else
9                   if C is a one-parity-SCC then
10                      solve C directly
11                  else
12                      compress priorities on C
13                      solve C using S
14              compute and remove attractors of the winning regions in C
```

Here we assume that the procedures in lines 2,7,11 and 13 update the variables $W_0, W_1, \sigma_0, \sigma_1$ with the information about winning regions and strategies that they have found on parts of the game. Hence, the solution to the entire game is stored in these variables in the end. Note that this generic algorithm is sound whenever the backend S is sound, meaning that the answer it computes for a node is correct. It is complete – an answer is computed for every node – if the backend is complete. However, this is not necessary. In order to guarantee completeness one does not need completeness of the backend. Instead it suffices if the backend solves at least one node of a given game. Then the generic algorithm will eventually terminate with $W_0 \cup W_1 = V$.

4 Empirical Evaluation

The generic solver described above, together with 8 real solvers from the literature has been implemented in a tool called PGSolver[1]. The tool is written in OCaml; and it uses standard array representations for manipulating game graphs, in particular no symbolic methods.

Here we report on some of PGSolver's runtime result on benchmarking families of games. These benchmarks should cover typical applications of parity games, in particular games from the area of model checking and decision problems for branching-time logics. However, we remark that so far there is no standard collection of parity game benchmarks. Here we start with the following.

- *Decision procedures.* We apply to certain (hard) formulas the exponential reduction of the validity problem for the modal μ-calculus to the parity game problem using Piterman's determinisation procedure [9].
- *Model checking.* We encode two verification problems (fairness and reachability) as parity games.

[1] publicly available via `http://www.tcs.ifi.lmu.de/pgsolver`

– *Random games.* Because of the absence of meaningful standardised parity game benchmarks we also evaluate the generic solver on random games.

The benchmarking games are presented in detail in the following. We only report on runtime results of the generic solver using the recursive algorithm, strategy improvement and the small progress measures algorithm as a backend because, on a separate note, these three algorithms turn out to be best in the sense that in general, they solve games faster than the other algorithms like the local model checker for example. This holds regardless of whether they are used directly or as a backend to the generic solver.

In order to exhibit the benefits of using the generic solver we present runtime results using various combinations of these optimisations: table columns labeled *all* contain running times obtained from the generic solver as presented above, i.e. using removal of self-cycles, SCC decomposition, priority compression and detection of special cases. Equally, columns *none* indicate using none of these, i.e. the entire game is solved by the backend. Columns labeled *scc* contain results from using SCC decomposition only, while *cyc* indicates the application of both SCC decomposition and removal of self-cycles. Finally, columns named *pcsg* imply the application of SCC decomposition, priority compression as well as detection of special cases. Note that the series presented in the tables to follow do not start with the smallest instances. We only present instances with non-negligable running times. On the other hand, the solving of larger instances not presented in the tables anymore has experienced time-outs after one hour, marked †, or the games were already to large to be stored in the heap space.

All tests have been carried out on a machine with two 2.4GHz Intel® Xeon™ processors and 4GB RAM space. The implementation does not (yet) support parallel computations, hence, each test is run on one processor only.

Decision Procedures. Consider the following μ-calculus formulas $\varphi_n := \psi_n \vee \neg\psi_n$, $n \in \mathbb{N}$, where

$$\psi_n \;:=\; \mu X_1.\nu X_2 \ldots \sigma_n X_n.\Big(q_1 \vee \Diamond\Big(X_1 \wedge \big(q_2 \vee \Diamond(X_2 \wedge \ldots (q_n \vee \Diamond X_n))\big)\Big)\Big)$$

with $\sigma_n = \mu$ if n is odd, and $\sigma_n = \nu$ otherwise. Obviously, φ_n is valid. It has been chosen because of its high alternation depth which requires a relatively large NBA $\mathcal{A}_n$ that checks for the unfoldings of ν-formulas. The nodes of the parity game G_n resulting from φ_n are sets of subformulas of φ_n together with a state of a deterministic parity automaton $\mathcal{B}_n$ which is equivalent to $\mathcal{A}_n$ and which gives the game node its priority. The number of priorities in $\mathcal{B}_n$ depends on the size of $\mathcal{A}_n$ [9]. Hence, φ_n is chosen in order to yield games of large index.

Another family to be considered is the following μ-calculus formula

$$\begin{aligned}\varphi'_n \;:=\; \nu X.\big(&\underbrace{q \wedge \Diamond(q \wedge \Diamond(\ldots \Diamond(q}_{2n \text{ times}} \wedge \Diamond(\neg q \wedge \Diamond X))\ldots)\big)\\ &\rightarrow\; \nu Z.\mu Y.(\neg q \wedge \Diamond Z) \vee (q \wedge \Diamond(q \wedge \Diamond Y))\end{aligned}$$

which describes the language inclusion $((aa)^n b)^\omega \subseteq ((aa)^* b)^\omega$.

				Recursive Algorithm					Strategy Improvement					Small Progress Measures				
	n	nodes	sccs	all	cyc	pcsg	scc	none	all	cyc	pcsg	scc	none	all	cyc	pcsg	scc	none
φ_n	2	462	84	0.0s	0.0s	0.0s	0.0s	0.0s	0.0s	0.2s	0.0s	0.2s	1.0s	0.0s	0.1s	0.0s	0.1s	0.3s
	3	2.5K	219	0.0s	0.0s	0.0s	0.0s	0.5s	0.2s	5.8s	0.2s	6.2s	4.7m	0.2s	1.6s	0.2s	1.7s	4.6s
	4	14K	966	0.1s	0.1s	0.2s	0.2s	6.4s	7.4s	4.1m	8.1s	4.6m	†	0.6s	12s	0.8s	13s	39s
	5	58K	3.4K	0.6s	0.7s	1.2s	1.2s	53s	49s	33m	75s	40m	†	2.2s	56s	2.9s	60s	2.4m
	6	262K	15K	5.0s	5.2s	29s	29s	17m	12m	†	14m	†	†	5.1s	6.7m	32s	7.8m	†
	7	982K	55K	22s	25s	4.6m	5.2m	†	†	†	†	†	†	35s	42m	5.1m	44m	†
φ'_n	10	4.5K	1.1K	0.1s	0.1s	0.1s	0.2s	0.3s	0.1s	22m	0.1s	30m	†	0.0s	0.4s	0.0s	0.4s	0.9s
	50	21.5K	5.5K	0.5s	0.5s	3.3s	4.5s	7.8s	0.6s	†	6.5s	†	†	0.0s	13.6s	0.0s	15s	61s
	100	42.6K	10.8K	0.6s	0.7s	20s	20s	2.7m	1.1s	†	21s	†	†	0.1s	2.2m	0.2s	2.3m	6.5m
	500	211K	54.1K	5.4s	6.7s	13m	13m	29m	6.5s	†	11m	†	†	0.3s	†	5.1s	†	†
	1K	423K	108K	6.5s	7.6s	53m	54m	†	7.9s	†	43m	†	†	0.6s	†	17s	†	†
	2K	846K	216K	24s	27s	†	†	†	18s	†	†	†	†	1.4s	†	87s	†	†

Fig. 2. Runtime results on games from the decision procedures domain

The times needed to solve the resulting games as well as their sizes are presented in Fig. 2.

Model Checking I. We encode a simple fairness verification problem as a parity game. States of a transition system modelling an *elevator* for n floors are of type $\{1,\ldots,n\} \times \{\mathtt{o},\mathtt{c}\} \times (\bigcup\{Perm(S) \mid S \subseteq \{1,\ldots,n\})$. The first component describes the current position of the elevator as one of the floors. The second component indicates whether the door is *open* or *closed*. The third component – a permutation of a subset of all available floors – holds the *requests*, i.e. those floors that should be served next. The transitions on these are as follows.

- At any moment, any request or none can be issued. For simplicity reasons, we assume that at most one floor is added to the requests per transition. Note that nondeterministically, no request can be issued, and a request for a certain floor that is already contained in the current requests does not change them.
- If the door is open then it is closed in the next step, the current floor does not change.
- If it is closed, the elevator moves one floor (up or down) into the direction of the first request. If the floor reached that way is among the requested ones, the door is opened and that floor is removed from the current requests. Otherwise, the door remains closed.

We consider two different implementations of this elevator model: the first one stores requests in FIFO style, the second in LIFO style. The games G_n (with FIFO), resp. G'_n (with LIFO) result from encoding the model checking problem for this transition system and the CTL* formula $\mathtt{A}(\mathtt{GF}\mathit{isPressed} \rightarrow \mathtt{GF}\mathit{isAt})$ as a parity game [14]. Proposition *isPressed* holds in any state s.t. the request list contains the number n, and *isAt* holds in a state where the current floor is n. Hence, this formula requires all runs of the elevator to satisfy the following fairness property: if the top floor is requested infinitely often then it is being served infinitely often. It can easily be formulated in the modal μ-calculus using a formula of size 11 and alternation depth 2 (of type ν–μ–ν). Hence the resulting

				Recursive Algorithm					Strategy Improvement					Small Progress Measures				
	n	nodes	sccs	all	cyc	pcsg	scc	none	all	cyc	pcsg	scc	none	all	cyc	pcsg	scc	none
G_n	3	564	95	0.0s	0.0s	0.0s	0.0s	0.0s	0.0s	0.1s	0.0s	0.1s	0.10s	0.0s	0.0s	0.0s	0.0s	0.1s
	4	2.6K	449	0.1s	0.1s	0.1s	0.1s	0.2s	0.1s	1.8s	0.1s	2.0s	3.1s	0.1s	0.4s	0.1s	0.4s	0.4s
	5	15.6K	2.6K	0.4s	0.5s	0.6s	0.7s	1.4s	0.5s	2.0s	0.7s	2.2s	2.3s	0.5s	2.9s	0.7s	3.0s	3.9s
	6	108K	18K	3.1s	4.7s	4.9s	6.0s	11s	3.1s	†	4.5s	†	†	4.0s	33s	5.8s	33s	37s
	7	861K	143K	34s	44s	50s	73s	1.8m	36s	†	53s	†	†	39s	6.7m	59s	6.9m	7.6m
G'_n	3	588	99	0.0s	0.0s	0.0s	0.0s	0.0s	0.0s	0.0s	0.0s	0.0s	0.0s	0.0s	0.0s	0.0s	0.0s	0.0s
	4	2.8K	473	0.1s	0.1s	0.1s	0.1s	0.1s	0.1s	0.2s	0.1s	0.3s	0.7s	0.1s	0.2s	0.1s	0.2s	0.2s
	5	16.3K	2.7K	0.6s	0.7s	0.8s	0.9s	1.0s	0.6s	2.4s	0.8s	5.7s	13s	0.5s	1.3s	0.5s	1.5s	1.5s
	6	111K	18.5K	3.8s	4.3s	5.6s	6.0s	7.1s	3.8s	21s	8.7s	46s	5.3m	5.2s	20s	7.0s	24s	24s

Fig. 3. Runtime results on games from the elevator verification example

parity games have constant index 3. Note that G_n encodes a positive instance of the model checking problem whereas G'_n encodes a negative one. The times needed to solve them as well as their sizes are presented in Fig. 3. Larger instances caused out-of-memory failures due to the size of the underlying transition system.

Model Checking II. Typical verification problems often lead to special parity games for which there are specialised solvers. For instance, CTL model checking problems lead to alternation-free parity games, i.e. those in which every SCC is a single-parity SCC with index 0 or 1. We therefore consider a second set of benchmarks from the verification domain in the form of very special games. We model the well-known *Towers of Hanoi* represented as a transition system in which states consist of three stacks containing the numbers $\{1, \ldots, n\}$. The initial state is $([1, \ldots, n], [], [])$, and each state has up to 6 successors resulting from shifting the top element of one stack to another for as long as the top of that is not smaller.

The property to be tested is the CTL formula $\texttt{EF}\mathit{fin}$, where *fin* holds in the state $([], [1, \ldots, n], [])$ only. The resulting game $G_n = (V, V_0, V_1, E, \Omega)$ is special because $V_1 = \emptyset$ and only priorities 0 and 1 are being assigned to the states. The times needed to solve these games and their sizes are shown in Fig. 4. Note that the interesting part of solving the games of the former example is the computation of the winning *regions* which show those states from which the elevator has fair runs. Here, however, the interesting part is the computation of the winning *strategy* for player 0 since it represents a strategy for solving the Towers-of-Hanoi game.

Random Games. Finally, we evaluate the generic solver on random games. Note that the standard model of a random game which chooses, for each node, some d successors and randomly assigns priorities as well as node owners, leads to graphs which typically consist of one large SCC and several 1-node SCCs which have successors in the large one. Those do not add significantly to the runtime of the solving process which is predominantly determined by the large SCC. Hence, SCC decomposition would not necessarily prove to be useful in this random model. The truth, however, is that SCC decomposition is indeed useful in general but this random model creates special games on which it is not. While special

			Recursive Algorithm					Strategy Improvement					Small Progress Measures				
n	nodes	sccs	all	cyc	pcsg	scc	none	all	cyc	pcsg	scc	none	all	cyc	pcsg	scc	none
5	972	244	0.0s	0.0s	0.0s	0.0s	0.0s	0.0s	0.0s	0.0s	0.0s	0.9s	0.0s	0.0s	0.0s	0.0s	0.3s
6	2.9K	730	0.0s	0.0s	0.0s	0.0s	0.1s	0.0s	0.0s	0.0s	0.0s	15s	0.0s	0.0s	0.0s	0.0s	2.2s
7	8.7K	2.1K	0.1s	0.1s	0.1s	0.1s	0.2s	0.1s	0.1s	0.1s	0.1s	5.4m	0.1s	0.1s	0.1s	0.1s	13s
8	26K	6.5K	0.2s	0.2s	0.4s	0.4s	0.7s	0.2s	0.2s	0.4s	0.4s	†	0.2s	0.2s	0.4s	0.4s	77s
9	78K	19K	0.7s	0.7s	1.0s	1.0s	2.2s	0.7s	0.7s	1.0s	1.0s	†	0.7s	0.7s	1.0s	1.0s	9.9m
10	236K	59K	2.3s	2.3s	3.3s	3.3s	4.1s	2.3s	2.3s	3.3s	3.3s	†	2.3s	2.3s	3.3s	3.3s	37m
11	708K	177K	7.2s	7.2s	13s	13s	21s	7.2s	7.2s	13s	13s	†	7.2s	7.2s	13s	13s	†

Fig. 4. Runtime results on games from the Towers-of-Hanoi example

		Recursive Algorithm					Strategy Improvement					Small Progress Measures				
nodes	avg.sccs	all	cyc	pcsg	scc	none	all	cyc	pcsg	scc	none	all	cyc	pcsg	scc	none
1K	31	0.0s	0.0s	0.0s	0.0s	0.0s	0.0s	0.0s	0.3s	0.3s	1.2s	0.0s	0.0s	†	†	†
2K	71	0.0s	0.0s	0.0s	0.0s	0.1s	0.0s	0.0s	0.3s	0.3s	6.7s	0.0s	0.0s	†	†	†
5K	130	0.1s	0.1s	0.1s	0.1s	0.2s	0.1s	0.1s	0.7s	0.7s	61s	0.1s	0.1s	†	†	†
10K	244	0.2s	0.2s	0.2s	0.2s	0.5s	0.2s	0.2s	0.8s	0.9s	5.9m	0.4s	0.4s	†	†	†
20K	458	0.3s	0.4s	0.3s	0.4s	1.1s	0.4s	0.4s	2.7s	2.7s	32m	0.7s	0.7s	†	†	†
50K	1K	0.8s	1.1s	0.8s	1.1s	2.9s	1.1s	1.1s	3.7s	3.7s	†	1.7s	1.8s	†	†	†
100K	1.5K	2.7s	4.1s	2.9s	4.1s	6.3s	3.9s	4.0s	6.0s	11s	†	6.2s	6.5s	†	†	†
200K	2.3K	5.9s	8.4s	5.5s	8.4s	14s	8.4s	8.4s	16s	22s	†	13s	14s	†	†	†
500K	3.4K	16s	20s	18s	19s	60s	19s	20s	34s	34s	†	30s	31s	†	†	†
1M	12K	99s	2.1m	1.7m	2.3m	14m	1.7m	1.7m	13m	26m	†	2.8m	3.0m	†	†	†

Fig. 5. Runtime results on random games

games are important to consider, random games should be more general ones since a random model is typically employed in order to capture all sorts of other games. Thus, we enhance this simple random model in order to obtain more interesting games of size n: first, create clusters of sizes $< n$ according to this model, then combine these whilst adding random edges between the clusters. Fig. 5 presents the average number of SCCs that these random games posses, as well as the corresponding average runtime results. Each row represents 100 random games of corresponding size.

5 Conclusions

The previous section shows that it is possible to solve large parity games efficiently in practice. Contrary to common believe, even a large number of priorities does not necessarily pose a great difficulty in practice. All in all, there are five notable, maybe even surprising observations that can generally be made here.

(1) The recursive algorithm is much better than the other two algorithms if applied without any optimisation and preprocessing techniques. We believe that this is due to the nature of the recursive algorithm being itself based on a continuous decomposition of the game.

(2) SCC decomposition alone is highly profitable already, and in general even moreso when combined with any of the other optimisations, particularly the elimination of self-cycles.

(3) Not every optimisation speeds up all algorithms likewise. The recursive algorithm seems to profit more from self-cycle elimination than from priority compression and solving of special cases. This could be due to the fact that without eliminating self-cycles it expectedly requires a deep recursive descend in order to detect them by the recursion mechanism. On the other hand, the considered special cases can be solved quite fast by the recursive algorithm. Regarding strategy iteration and small progress measures iteration, it is the other way round: the detection of special cases as well as priority compression speed up the algorithms much more than treating self-cycles beforehand. The former is not surprising because both algorithms summarize nodes with the same priority and it is clear that the direct solution of special cases is faster than applying the iteration techniques to them. Finally it is also clear why the strategy iteration does not profit as much from elimination of self-cycles as the recursive algorithm since strategy iteration basically detects cycles and computes attractor-like strategies seeking cycles which is similar to the preprocessing technique that removes self-cycles. Similarly, the small progress measures algorithm easily detects self-cycles and back-propagates them through the graph which also corresponds to the computation of attractor-like strategies.

(4) There are even complex instances like the Towers-of-Hanoi example that are completely solved by the generic algorithm, i.e. without calling the backend even once. Obviously, generic optimisations as discussed in this paper have not at all the potential to give rise to a polynomial time algorithm that solves arbitrary parity games. But solving real-world parity game problems in practice can be heavily sped up by generic optimisation techniques.

(5) In general, it is advisable to enable all optimisations. Thus even an inexperienced user is on the safe side by activitating all of them. Also, using can cause tremendous speed-ups, which is witnessed for example in the drop of the average runtime from 14 to 1.5 minutes on random games with 1 million nodes using the recursive algorithm.

Despite developing additional universal optimisation techniques and parallelizing existing backend algorithms, there is another approach that should turn out to be of high value for practical solving: since it is very unlikely that one would ever find a real-world family of games on which *all* of the known backend algorithms show bad performance, there is an immediate improvement for the generic solver: it could take an arbitrary number of complete solvers as arguments and run them in parallel on those parts that cannot be solved by simpler methods. As soon as any of them provides a solution, the other computations can be killed.

References

1. Antonik, A., Charlton, N., Huth, M.: Polynomial-time under-approximation of winning regions in parity games. Technical report, Dept. of Computer Science, Imperial College London (2006)
2. Arnold, A., Vincent, A., Walukiewicz, I.: Games for synthesis of controllers with partial observation. Theor. Comput. Sci. 303(1), 7–34 (2003)

3. Emerson, E.A., Jutla, C.S.: Tree automata, μ-calculus and determinacy. In: Proc. 32nd Symp. on Foundations of Computer Science, San Juan, Puerto Rico, pp. 368–377. IEEE, Los Alamitos (1991)
4. Jurdziński, M.: Deciding the winner in parity games is in $UP \cap$co-UP. Inf. Process. Lett. 68(3), 119–124 (1998)
5. 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)
6. Jurdziński, M., Paterson, M., Zwick, U.: A deterministic subexponential algorithm for solving parity games. In: Proc. ACM-SIAM Symp. on Discrete Algorithms, SODA 2006, pp. 114–123. ACM/SIAM (2006) (to appear)
7. Kähler, D., Wilke, T.: Complementation, disambiguation, and determinization of Büchi automata unified. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part I. LNCS, vol. 5125, pp. 724–735. Springer, Heidelberg (2008)
8. Lange, M.: Solving parity games by a reduction to SAT. In: Majumdar, R., Jurdziński, M. (eds.) Proc. Int. Workshop on Games in Design and Verification, GDV 2005 (2005)
9. Piterman, N.: From nondeterministic Büchi and Streett automata to deterministic parity automata. In: Proc. 21st Symp. on Logic in Computer Science, LICS 2006, pp. 255–264. IEEE Computer Society, Los Alamitos (2006)
10. Safra, S.: On the complexity of ω-automata. In: Proc. 29th Symp. on Foundations of Computer Science, FOCS 1988, pp. 319–327. IEEE, Los Alamitos (1988)
11. 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)
12. Schewe, S.: An optimal strategy improvement algorithm for solving parity and payoff games. In: Kaminski, M., Martini, S. (eds.) CSL 2008. LNCS, vol. 5213, pp. 369–384. Springer, Heidelberg (2008)
13. Stevens, P., Stirling, C.: Practical model-checking using games. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, pp. 85–101. Springer, Heidelberg (1998)
14. Stirling, C.: Local model checking games. In: Lee, I., Smolka, S.A. (eds.) CONCUR 1995. LNCS, vol. 962, pp. 1–11. Springer, Heidelberg (1995)
15. Tarjan, R.E.: Depth-first search and linear graph algorithms. SIAM J. Computing 1, 146–160 (1972)
16. van de Pol, J., Weber, M.: A multi-core solver for parity games. Electr. Notes Theor. Comput. Sci. 220(2), 19–34 (2008)
17. Vöge, J., Jurdziński, 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)
18. Zielonka, W.: Infinite games on finitely coloured graphs with applications to automata on infinite trees. TCS 200(1–2), 135–183 (1998)