Solving Parity Games: Explicit vs Symbolic*

Antonio Di Stasio[1], Aniello Murano[1], Moshe Y. Vardi[2]

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

Abstract. In this paper we provide a broad investigation of the symbolic approach for solving Parity Games. Specifically, we implement in a fresh tool, called `SymPGSolver`, four symbolic algorithms to solve Parity Games and compare their performances to the corresponding explicit versions for different classes of games. By means of benchmarks, we show that for random games, even for constrained random games, explicit algorithms actually perform better than symbolic algorithms. The situation changes, however, for structured games, where symbolic algorithms seem to have the advantage. This suggests that when evaluating algorithms for parity-game solving, it would be useful to have real benchmarks and not only random benchmarks, as the common practice has been.

1 Introduction

Parity games (PGs) [12, 24] are abstract games with a key role in automata theory and formal verification [7, 9, 18, 19, 23]. PGs are two-player turn-based games played on directed graphs whose nodes are labeled with priorities. Players take turns moving a token along the graph's edges, starting from an initial node. A play induces an infinite path and Player 0 wins the play if the smallest priority visited infinitely often is even. Solving a PG amounts checking whether Player 0 can force such a winning play. Several algorithms to solve PGs have been proposed aiming to tighten the asymptotic complexity of the problem, as well as to work well in practice. Well known are *Recursive* (`RE`) [24], small-progress measures (`SPM`) [14], and `APT` [10, 18], the latter originated to deal with the emptiness of parity automata. Notably, all these algorithms are *explicit*, that is, they are formulated in terms of the underlying game graphs. Due to the exponential growth of finite-state systems, and, consequently, of the corresponding game graphs, the state-explosion problem limits the scalability of these algorithms in practice.

Symbolic algorithms are an efficient way to deal with extremely large graphs. They avoid explicit access to graphs by using a set of predefined operations that manipulate Binary Decision Diagrams (BDDs) [3] representing these graphs. This enables handling large graphs succinctly, and, in general, it makes symbolic algorithms scale better than explicit ones. For example, in hardware model checking symbolic algorithms enable going from millions of states to 10^{20} states

* Work supported by NSF grants CCF-1319459 and IIS-1527668, NSF Expeditions in Computing project "ExCAPE: Expeditions in Computer Augmented Program Engineering" and GNCS 2018: Logica, Automi e Giochi per Sistemi Auto-adattivi.

and more [4, 20]. In contrast, in the context of PG solvers, symbolic algorithms have been only marginally explored. In this direction we just mention a symbolic implementation of `RE` [2, 16], which, however, has been done for different purposes and no benchmark comparison with the explicit version has been carried out. Other works close to this topic and worth mentioning are [5, 8], where a symbolic version of `SPM` has been theoretically studied but not implemented.

In this work we provide the first broad investigation of the symbolic approach for solving PGs. We implement four symbolic algorithms and compare their performances to the corresponding explicit versions for different classes of PGs. Specifically, we implement in a new tool, called `SymPGSolver`, the symbolic versions of `RE`, `APT`, and two variants of `SPM`. The tool also allows to generate random games, as well as compare the performance of different symbolic algorithms.

The main result we obtain from our comparisons is that for random games, and even for constrained random games (see Section 4), explicit algorithms actually perform better than symbolic ones, most likely because BDDs do not offer any compression for random sets. The situation changes, however, for structured games, where symbolic algorithms sometimes outperform explicit algorithms. This is similar to what has been observed in the context of model checking [11]. We take this as an important development because it suggests a methodological weakness in this field of investigation, due to the excessive reliance on random benchmarks. We believe that, in evaluating algorithms for PG solving, it would be useful to have real benchmarks and not only random benchmarks, as the common practice has been. This would lead to a deeper understanding of the relative merits of PG solving algorithms, both explicit and symbolic.

2 Explicit and Symbolic Parity Games

Explicit Parity Games. 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 binary relation of moves, and $\mathsf{p} : \mathrm{P} \to \mathbb{N}$ is the priority function. 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 q.

A *play* over $\mathcal{G}$ is an infinite sequence $\pi = q_1 q_2 \ldots \in \mathrm{P}^\omega$ of nodes that agree with Mv, *i.e.*, $(q_i, q_{i+1}) \in Mv$, for each $i \in \mathbb{N}$. By $\mathsf{p}(\pi) = \mathsf{p}(q_1)\mathsf{p}(q_2)\ldots \in \mathbb{N}^\omega$ we denote the priority sequence associated to π, and by $\mathsf{Inf}(\pi)$ and $\mathsf{Inf}(\mathsf{p}(\pi))$, the sets of nodes and priorities that occur infinitely often in π and $\mathsf{p}(\pi)$, respectively. A play π is *winning* for Player 0 if $\min(\mathsf{Inf}(\mathsf{p}(\pi)))$ is even. Player 0 (Player 1) strategy is a function $\mathsf{str}_0 : \mathrm{P}^*\mathrm{P}_0 \to \mathrm{P}$ ($\mathsf{str}_1 : \mathrm{P}^*\mathrm{P}_1 \to \mathrm{P}$) that agrees with Mv. Given a node q, $\mathsf{play}(q, \mathsf{str}_0, \mathsf{str}_1)$ is the unique play starting in q that agrees with both str_0 and str_1. Player 0 *wins* the game $\mathcal{G}$ from q if a strategy str_0 exists such that, for all strategies str_1 it holds that $\mathsf{play}(q, \mathsf{str}_0, \mathsf{str}_1)$ is winning for Player 0. Then q is declared *winning* for Player 0. By $\mathrm{Win}_0(\mathcal{G})$ we denote the set of winning nodes in $\mathcal{G}$ for Player 0. Parity games enjoy determinacy, *i.e.*, for every node q, either $q \in \mathrm{Win}_0(\mathcal{G})$ or $q \in \mathrm{Win}_1(\mathcal{G})$ [12]. Also, if Player 0 has a winning strategy from a node q, then she has a memoryless one from q [24]. A strategy str_0 is

memoryless if, for all prefixes of plays ρ_1, ρ_2, it holds that $\mathsf{str}_{\mathrm{o}}(\rho_1) = \mathsf{str}_{\mathrm{o}}(\rho_2)$ iff last nodes of ρ_1 and ρ_2 coincide. Then, one can use $\mathsf{str}_{\mathrm{o}}$ defined as $\mathsf{str}_{\mathrm{o}} : \mathrm{P}_{\mathrm{o}} \to \mathrm{P}$.

Symbolic Parity Games. We start with some notation. In the sequel we use symbols x_i for propositions (variables), l_i for literals, *i.e.*, positive or negative variables, f for a generic Boolean formula, $||f||$ for the set of interpretations that makes the formula f true, and $\lambda(f) \subseteq V$ for the set of variables in f.

Definition 1. *Given a PG $\mathcal{G} \triangleq \langle \mathrm{P}_{\mathrm{o}}, \mathrm{P}_{\mathrm{1}}, Mv, \mathsf{p} \rangle$, the corresponding* symbolic PG *(SPG, for short) is the tuple $\mathcal{F} = (\mathcal{X}, \mathcal{X}_M, f_{\mathrm{P}_0}, f_{\mathrm{P}_1}, f_{Mv}, \eta_{\mathsf{p}})$ defined as follows:*

- *$\mathcal{X} = \{x_1, \ldots, x_n\}$, with $n = \lceil log_2(|\mathrm{P}|) \rceil$, is the set of* propositions *used to encode nodes in $\mathcal{G}$, i.e., to each $v \in \mathrm{P}$ we associate a Boolean formula $f_v = l_{v,1} \wedge \ldots \wedge l_{v,n}$ where $l_{v,i}$ is either x_i or $\overline{x_i}$. We also associate to v the interpretation $\mathcal{X}_v \in 2^{\mathcal{X}}$, i.e., the subset of variables appearing positively in f_v.*
- *$\mathcal{X}_M = \{x'_1, \ldots, x'_n\}$, with $n = \lceil log_2(|\mathrm{P}|) \rceil$, is the set of propositions used to encode the successor nodes such that $\mathcal{X} \cap \mathcal{X}_M = \emptyset$. We extend to $\mathcal{X}_M$ the definitions of f_v and X_v as used in the previous item.*
- *f_{P_i}, for $i \in \{0, 1\}$, is a Boolean formula such that $||f_{\mathrm{P}_i}|| = \mathrm{P}_i$.*
- *f_{Mv} is a Boolean formula over the propositions $\mathcal{X} \cup \mathcal{X}_M$ such that $||f_{Mv}|| = Mv$.*
- *η_{p} is the symbolic representation of the priority function p; formally, it is a function $\eta_{\mathsf{p}} : 2^{\mathcal{X}} \to \mathbb{N}$ associating to each interpretation X_v a natural number.*

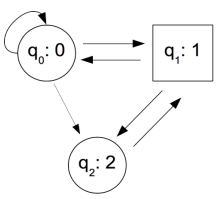

Fig. 1. A parity game

Example. Consider the PG depicted in Fig. 1. It has $\mathrm{P}_{\mathrm{o}} = \{q_0, q_2\}$ (circles) and $\mathrm{P}_{\mathrm{1}} = \{q_1\}$ (squares); Mv is given by arrows; and $\mathsf{p}(q_i) = i$, for $1 \leq i \leq 3$. The correlating SPG $\mathcal{F} = (\mathcal{X}, \mathcal{X}_M, f_{\mathrm{P}_0}, f_{\mathrm{P}_1}, f_{Mv}, \eta_{\mathsf{p}})$ is as follows: $\mathcal{X} = \{x_1, x_2\}$ and $\mathcal{X}_M = \{y_1, y_2\}$ are the set of propositions; $f_{\mathrm{P}_0} = (\overline{x_1} \wedge \overline{x_2}) \vee (x_1 \wedge \overline{x_2})$ and $f_{\mathrm{P}_1} = (\overline{x_1} \wedge x_2)$ are Boolean formulas representing P_0 and P_1, respectively; $f_{Mv} = (\overline{x_1} \wedge \overline{y_1} \wedge \overline{x_2} \wedge \overline{y_2}) \vee (\overline{x_1} \wedge \overline{y_1} \wedge \overline{x_2} \wedge y_2) \vee (\overline{x_1} \wedge \overline{y_1} \wedge x_2 \wedge \overline{y_2}) \vee (\overline{x_1} \wedge y_1 \wedge \overline{x_2} \wedge \overline{y_2}) \vee (\overline{x_1} \wedge y_1 \wedge x_2 \wedge \overline{y_2}) \vee (x_1 \wedge \overline{y_1} \wedge \overline{x_2} \wedge y_2)$ is the Boolean formula for Mv; finally, the function η_{p}, given by $\eta_{\mathsf{p}}(0,0) = 0$, $\eta_{\mathsf{p}}(0,1) = 1$ and $\eta_{\mathsf{p}}(1,0) = 2$, represents the priority function p.

To solve an SPG we compute the Boolean formulas $f_{\mathrm{Win}_{\mathrm{o}}}$ over $\mathcal{X}$ that is satisfied by those interpretations that correspond to winning nodes for Player 0.

For technical reasons, we also need the definition of symbolic sub-games.

Definition 2. *Let $\mathcal{G} \triangleq \langle \mathrm{P}_{\mathrm{o}}, \mathrm{P}_{\mathrm{1}}, Mv, \mathsf{p} \rangle$ be a PG and $U \subseteq \mathrm{P}$. By $\mathcal{G} \setminus U = (\mathrm{P}_0 \setminus U, \mathrm{P}_1 \setminus U, Mv \setminus (U \times \mathrm{P} \cup \mathrm{P} \times U), \mathsf{p}_{|\mathrm{P} \setminus U})$ we denote the PG restricted to nodes $\mathrm{P} \setminus U$.*

Let f_U be a Boolean formula such that $||f_U|| = U$ and $\mathcal{F} = (\mathcal{X}, \mathcal{X}_M, f_{\mathrm{P}_0}, f_{\mathrm{P}_1}, f_{Mv}, \eta_{\mathsf{p}})$ be the corresponding SPG of the PG $\mathcal{G}$. By $\mathcal{F}_{\mathrm{P} \setminus U} = (\mathcal{X}, \mathcal{X}_M, f'_{\mathrm{P}_0}, f'_{\mathrm{P}_1}, f'_{Mv}, \eta'_{\mathsf{p}})$ we denote the SPG of $\mathcal{G} \setminus U$, where:

- *$f'_{\mathrm{P}_i} = f_{\mathrm{P}_i} \wedge \neg f_U$, for $i \in \{0, 1\}$, is the Boolean formula for nodes $v \in \mathrm{P}_i \setminus U$;*
- *$f_{Mv'} = f_{Mv} \wedge \neg(f_U \vee f'_U)$, where $||f'_U|| = U$ and $\lambda(f'_U) = \mathcal{X}_M$, is the Boolean formula representing moves restricted to $Mv \setminus (U \times \mathrm{P} \cup \mathrm{P} \times U)$;*
- *$\eta'_{\mathsf{p}} = 2^{\mathcal{X}} \to \mathbb{N}$ is the symbolic representation of $\mathsf{p}_{|\mathrm{P} \setminus U}$ that associates to the interpretations X_v satisfying the Boolean formula $f_{\mathrm{P}} \wedge \neg f_U$ a natural number.*

3 Solving Parity Games: Explicit vs Symbolic Algorithms

3.1 Explicit Algorithms

Small Progress Measures Algorithm (SPM) [13]. The core idea of SPM is a *progress measure* based on a *ranking function* that assigns to each node a vector of counters collecting the number n of times Player 1 can force a play to visit an odd priority until a lower priority is seen. If this value is sufficiently large, then the node is declared winning for Player 1. SPM computes the *progress measure* by updating the values of a node according to those associated to its successors, *i.e.*, by computing a least fixed-point for all nodes with respect to the ranking function.

We fix some notation. Let $\mathcal{G}$ be a PG with maximal priority c and $d \in \mathbb{N}^c$ be a c-tuple of non-negative integers. By $<$ we denote the usual lexicographic ordering over $\mathbb{N}^c$. For all $n \in \mathbb{N}$, by $[n]$ we denote the set $\{0, \ldots, n-1\}$. For each odd number i, by n_i we denote the number of nodes in $\mathcal{G}$ with priority i. For i even, we set $n_i = 0$. The progress measure domain is defined as $M_G^\top = M_G \cup \{\top\}$ with $M_G = (M_0 \times \ldots \times M_{c-1})$ and $M_i = [n_i]$. The element $\top$ is the biggest value such that $m < \top$ for all $m \in M_G$. For $d = (d_0, \ldots, d_{c-1})$ and $l < c$, we set $\langle d \rangle_l = (d_0, \ldots, d_l, 0, \ldots, 0)$, i.e., all $d_{i>l}$ in d are set to 0. By $\mathrm{inc}(d)$ we denote the smallest tuple $d' \in M_G^\top$ such that $d < d'$. This notion easily extends to tuples in $\mathbb{N}^l$ by defining $\mathrm{inc}_l(d)$ with $l > 0$ to be the smallest tuple $d' \in M_G^\top$ such that $d <_l d'$ iff $\langle d \rangle_l < \langle d' \rangle_l$. In particular, for $d = \top$ we have $\mathrm{inc}_l(d) = d$. Otherwise, $\mathrm{inc}_l(d) = \langle d \rangle_l$ if l is even and $\min\{y \in M_G^\top | y >_l d\}$ if l is odd. To conclude we introduce a *ranking function* $\varrho : \mathrm{P} \to M_G^\top$ that associates to each node either a c-tuple in M_G or the value $\top$, and a function *Lift* that defines the increment of a node v based on its priority and the values of its neighbors. The formal definition of *Lift* follows.

$$\mathrm{Lift}(\varrho, v)(u) = \begin{cases} \mathrm{inc}_{\mathsf{p}(v)}(\min\{\varrho(w) | (v,w) \in Mv\}), & \text{if } v \in \mathrm{P}_0 \\ \mathrm{inc}_{\mathsf{p}(v)}(\max\{\varrho(w) | (v,w) \in Mv\}), & \text{if } v \in \mathrm{P}_1 \\ \varrho(u), & \text{otherwise} \end{cases}$$

Lift is monotone and the progress measures over v is the least fixed point of $Lift(\cdot, v)$. The solution algorithm starts by setting 0 to every node. Then, it applies the lift as long as $\mathrm{Lift}(\varrho, v)(u) > \varrho(v)$ for some node v. Next lemma relates the solution of a PG $\mathcal{G}$ with the least fixed point calculation of Lift.

Lemma 1 ([13]). *If ϱ is a progress measures function then the set of nodes v with $\varrho(v) < \top$ is the set of winning nodes for Player 0.*

The APT Algorithm (APT) [10]. APT was first introduced by Kupferman and Vardi in [18] to solve parity games via emptiness checking of parity automata. It makes use of two special sets of nodes, V and A, called *Visiting* and *Avoiding*, respectively. Intuitively, a node is visiting for a player at the stage in which it is clear that, by reaching that node, he can surely induce a winning play. The

reasoning is symmetric for the avoiding set. The algorithm, in turns, tries to partition all nodes of the game into these two sets. Some formal details follow.

Given a PG $\mathcal{G}$, an *Extended Parity Game*, (EPG, for short) is a tuple $\langle \mathrm{P_0}, \mathrm{P_1}, \mathrm{V}, \mathrm{A}, Mv, \mathsf{p}\rangle$ where $\mathrm{P_0}$, $\mathrm{P_1}$, $\mathrm{P} = \mathrm{P_0} \cup \mathrm{P_1}$, Mv, and p are as in PG. Moreover, the sets $\mathrm{V}, \mathrm{A} \subseteq \mathrm{P}$ are two disjoint sets of *Visiting* and *Avoiding* nodes, respectively. For EPGs we make use of the same notion of play as given for PG. A play π in $\mathrm{P} \cdot (\mathrm{P} \setminus (\mathrm{V} \cup \mathrm{A}))^* \cdot \mathrm{V} \cdot \mathrm{P}^\omega$ is winning for Player 0, while a play π in $\mathrm{P} \cdot (\mathrm{P} \setminus (\mathrm{V} \cup \mathrm{A}))^* \cdot \mathrm{A} \cdot \mathrm{P}^\omega$ is winning for Player 1. A play π that never hits either V or A is declared winning for Player 0 iff it satisfies the parity condition, *i.e.*, $\min(\mathsf{Inf}(\mathsf{p}(\pi)))$ is even, otherwise it is winning for Player 1.

To solve an EPG, `APT` makes use of two functions: $\mathsf{force}_i(X)$ and $\mathrm{Win}_i(\alpha, \mathrm{V}, \mathrm{A})$. For $\mathrm{X} \subseteq \mathrm{P}$, $\mathsf{force}_i(\mathrm{X}) = \{q \in \mathrm{P}_i : \mathrm{X} \cap Mv(q) \neq \emptyset\} \cup \{q \in \mathrm{P}_{1-i} : \mathrm{X} \subseteq Mv(q)\}$ is the set of nodes from which Player i can force, in one step, a move to X. The function $\mathrm{Win}_i(\alpha, \mathrm{V}, \mathrm{A})$ denotes the nodes from which Player i has a strategy that avoids A, and either forces a visit to V or satisfies the parity condition α. Note that in `APT` α is given as a finite sequence $\alpha = \mathrm{F_o} \cdot \ldots \cdot \mathrm{F}_k$ of sets, where $\mathrm{F}_j = \mathsf{p}^{-1}(j)$, i.e., the set of nodes with priority j. Formally, $\mathrm{Win}_i(\alpha, \mathrm{V}, \mathrm{A})$ is defined as follows. If $\alpha = \varepsilon$, then $\mathrm{Win}_i(\alpha, \mathrm{V}, \mathrm{A}) = \mathsf{force}_i(\mathrm{V})$. Otherwise, if $\alpha = \mathrm{F} \cdot \alpha'$, for some set F, then $\mathrm{Win}_i(\alpha, \mathrm{V}, \mathrm{A}) = \mu\mathrm{Y}(\mathrm{P} \setminus (\mathrm{Win}_{1-i}(\alpha', \mathrm{A} \cup (\mathrm{F} \setminus \mathrm{Y}), \mathrm{V} \cup (\mathrm{F} \cap \mathrm{Y}))))$, where μ is the least fixed-point operator.

Recursive Zielonka Algorithm (`RE`) [24]. Introduced by Zielonka, `RE` makes use of a divide and conquer technique. The core subroutine of `RE` is the *attractor*. Intuitively, given a set of nodes U the attractor of U for a Player i represents those nodes that i can force the play toward. At each step, the algorithm removes all nodes with the highest priority p, together with all nodes Player $i = p \ mod \ 2$ can attract to them, and recursively computes the winning sets (W_0, W_1) for Player 0 and Player 1, respectively, on the remaining subgame. If Player i wins the subgame, then he also wins the whole starting game. Otherwise if Player i does not win the subgame, *i.e.*, W_{1-i} is non empty, the algorithm computes the attractor for Player $1 - i$ of W_{1-i} and recursively solves the subgame.

3.2 Symbolic Algorithms

We now describe symbolic versions of the explicit algorithms listed in Section 3.1.

SPG Implementation. An SPG can be implemented using Binary Decision Diagrams (BDDs) and Algebraic Decision Diagrams (ADDs) [1] to represent and manipulate the associated Boolean functions introduced along with its definition. ADDs were introduced to extend BDDs by allowing values from any arbitrary finite domain to be associated with the terminal nodes of the diagram, *i.e.*, an ADD can be seen as a BDD whose leaves may take on values belonging to a set of constants different from 0 and 1. Given an SPG $\mathcal{F} = (\mathcal{X}_1, \mathcal{X}_2, f_{\mathrm{P}_0}, f_{\mathrm{P}_1}, f_{Mv}, \eta_{\mathsf{p}})$ with maximal priority c, we use BDDs to represent the Boolean formulas f_{P_0}, f_{P_1} and f_{Mv}, and an ADD for the function η_{p}. Moreover, we decompose the function η_{p} into a sequence of BDDs $\mathcal{B} = \langle B_0, \ldots, B_{c-1}\rangle$ where each B_i encodes

the nodes with priority i, to easily manage the selection of a set of nodes with a specific priority. In the sequel, by BDD (*resp.*, ADD) f, we denote the BDD (*resp.*, ADD) representing the function f.

Symbolic SPM (SSP) [5]. This is the first symbolic implementation of SPM we are aware of, and which we describe with some minor corrections compared to the one in [5]. Lift is encoded by using ADDs and the algorithm computes the progress measure as the least fixed point f_G of $\text{Lift}(f, v)$ on a ranking function here given by the function $f : \text{P} \to D$, with $D = M_G \cup \{\infty, -\infty\}$. The algorithm takes as input an SPG $\mathcal{F}$ and returns an ADD representing the least fixed point f_G such that the set of winning nodes for Player 0 is $\{v | f_G(v) < \infty\}$, and the set of winning nodes for Player 1 is $\{v | f_G(v) = \infty\}$. See Algorithm 1.

Algorithm 1 Symbolic Small Progress Measures

```
1: procedure PARITY (F)
2:     f =→ (f_P, −∞);
3:     repeat
4:         f_old = f; f = false;
5:         for j = 0 to c − 1 do
6:             f = f OR MAXeo(f_old, j) OR MINeo(f_old, j);
7:     until f = f_old
```

The algorithm calls the procedure MAXeo (*resp.,* MINeo), which given an ADD $f : \text{P} \to D$, the BDD f_{Mv}, and $1 \leq j \leq k$, returns an ADD that assigns to every node $v \in \text{P}_1$ (*resp.,* $v \in \text{P}_0$,), with $\mathsf{p}(v) = j$, the value $\text{inc}_j(\max\{f(v') | (v, v') \in Mv\})$ (*resp.,* $\text{inc}_j(\min\{f(v') | (v, v') \in Mv\})$).

MINeo (*resp.,* MAXeo) aims at constructing an ADD that represents the ranking function $f_{\min}(v) = \min\{f(v') | (v, v') \in Mv\}$ (*resp.,* $f_{\max}(v) = \max\{f(v') | (v, v') \in Mv\}$). To do this, given an ADD $f : \text{P} \to D$ and the BDD f_{Mv}, it is generated an ADD $f_{suc} : (\text{P} \times \text{P}) \to D$ such that $f_{suc}(v, v') = d$ if $(v, v') \in Mv$ and $f(v') = d$. Then, the ADD f_{suc} is given in input to the procedure MIN, described in Algorithm 2, that constructs the ADD for f_{min}. The procedure MAX is defined similarly. Let n be an ADD node, we refer to the left and right successors of n as $n.l$ and $n.r$, respectively, and refer to the variable that n represents as $n.v$.

Algorithm 2 Procedure MIN

```
1: procedure MIN(ADD n)
2:     if n is a terminal node then
3:         return n
4:     if n.v is in X then
5:         return (n.v AND MIN(n.r)) OR ( NOT n.v AND MIN(n.l))
6:     if n.v is in X' then
7:         return MERGE(MIN(n.r), MIN(n.l)))
```

The procedure MIN calls the procedure MERGE, reported in Algorithm 3, that gets in input the pointer to the roots n_1 and n_2 of two ADDs representing the functions f_1 and f_2, both from some set $U \subseteq \text{P}$ to D, and merges them to an ADD in which every $u \in U$ is mapped into $\min(f_1(u), f_2(u))$.

Algorithm 3 Procedure `MERGE`

```
1: procedure MERGE(ADD n1, ADD n2)
2:     if n1 and n2 are a terminal nodes then
3:         return min(n1, n2)
4:     if o(n1.v) < o(n2.v) then
5:         return (n1.v AND MERGE(n1.r, n2)) OR ( NOT n1.v AND MERGE(n1.l, n2))
6:     if o(n1.v) > o(n2.v) then
7:         return (n2.v AND MERGE(n2.r, n1)) OR ( NOT n2.v AND MERGE(n2.l, n1))
8:     return (n1.v AND MERGE(n1.r, n2.r)) OR ( NOT n1.v AND MERGE(n1.l, n2.l))
```

Set-Based Symbolic `SPM` (`SSP2`) [8]. This is a symbolic implementation of `SPM` that has been introduced very recently. It allows to use only basic set operations like $\cup$, $\cap$, $\setminus$, $\subseteq$, and one-step predecessor operations for its description. Unlike the implementation described previously, the ranking function is implicitly encoded by using sets of nodes. This allows representing the *Lift* operator just by BDDs.

To encode the ranking function the algorithm defines for each rank $r \in M_G^\top$ the set S_r containing the nodes with rank r or higher. Formally, given the ranking function $\varrho : \mathrm{P} \to M_G^\top$, the corresponding sets are defined as $S_r = \{v | \varrho(v) \geq r\}$. Conversely, given the family of sets $\{S_r\}_r$, the corresponding ranking function, say $\varrho_{\{S_r\}_r}$, is given by $\varrho_{\{S_r\}_r}(v) = \max\{r \in M_G^\top | v \in S_r\}$. This formulation encodes the ranking function with sets but uses exponential in c many sets.

Space is reduced to a linear number of sets by encoding the value of each coordinate of the rank r, separately. In detail, for each odd priority i, the algorithm defines the sets $C_0^i, \ldots, C_{n_i}^i$. Each set C_x^i with $x \in \{0, \ldots, n_i\}$ contains the nodes that have x as i-th coordinate of their rank. Therefore, the algorithm has to construct the set S_r whenever it needs it.

Let $Cpre_i(\mathrm{X}) = \{q \in \mathrm{P}_i : \mathrm{X} \cap Mv(q) \neq \emptyset\} \cup \{q \in \mathrm{P}_{1-i} : \mathrm{X} \subseteq Mv(q)\}$ the one-step controllable predecessor operator. The algorithm starts initializing the sets S_r for $r > 0$ to empty, and S_0 with the set of all nodes P. The rank r initially is set to the second lowest rank $\mathrm{inc}((0, \ldots, 0))$. Then, at each iteration the set S_r is updated for the current value of r by using the *Lift* encoded by the $Cpre_i$ operator. After the update of S_r, it is checked if $S'_r \supseteq S_r$ for all $r' < r$, *i.e.*, if the property of the *anti-monotonicity* is preserved. Anti-monotonicity together with the definition of the sets S'_r allows to decide whether the rank of a node v can be increased to r by only considering one set S'_r. If the anti-monotonicity is preserved, then for $r < \top$ the value of r is increased to the next highest rank and for $r = \top$ the algorithm terminates. Otherwise the nodes newly added to S_r are also added to all sets with $r' < r$ that do not already contain them; the variable r is then updated to the lowest r' for which a new node is added to S'_r in this iteration. Due to lack of space, we omit the algorithm (see [8] for more details).

Symbolic versions of `RE` (`SRE`) and `APT` (`SAPT`). `RE` and `APT` can be easily rephrased symbolically by using BDDs to represent the operations they make use of set basic operations like union, intersection, complement, and inclusion; the controllable predecessor operators used to implement the function force_i in `APT`, and the attractor in `RE`; the symbolic construction of a subgame used in `RE` and implemented following the definition of symbolic subgame reported previously.

4 Experimental Evaluations: Methodology and Results

We now analyze the performance of the introduced symbolic approach to solve PGs and compare with the explicit one. We have implemented the symbolic algorithms described in Section 3.2 in a fresh tool, called `SymPGSolver` (Symbolic Parity Games Solver). `SymPGSolver`[1] is implemented in C++ and uses the CUDD[2] package as the underlying BDD and ADD library. The platform provides a collection of tools to randomly generate and solve SPGs, as well as compare the performance of different symbolic algorithms.

We have also compared them with *Oink*, a platform recently developed in C++ by Tom van Dijk [22], which collects the large majority of explicit PGs algorithms introduced in the literature [6, 14, 15, 24].

4.1 Experimental results

In this section we report on some experimental results on evaluating the performance for the explicit algorithms `RE`, `APT`, and `SPM` as well as their corresponding symbolic versions `SRE`, `SAPT`, `SSP` and `SSP2`. All tests have been run on an Intel Core i7 @2.40GHz, with 16GB of RAM running macOS 10.12. We have used different classes of parity games: random games with linear structures, ladder games, clique games as well as games corresponding to practical model checking problems. Random games are generated by `SymPGSolver`, while for ladder and clique games we use *Oink*. We have taken 100 different instances for each class of games and used the average time execution. In all tests, we use $\texttt{abort}_T$ to denote an aborted execution due to time-out (greater than 200 seconds). On the class of ladder games and in model checking problems the benchmarks have been executed using the variable ordering given by the heuristic WINDOW2 module available in the CUDD package.

Random Games with linear structure. Tabakov and Vardi showed that in the context of automata-theoretic problems, explicit algorithms generally dominate symbolic algorithms, as BDDs do not offer any compression for random sets [21]. We found that the same holds for parity-game solving (we omit details due to lack of space). In [21] it was observed that, in case of random games with linear structures, the symbolic algorithms are the best performing ones. Hence, we have investigated the same class here as well, but with a different outcome.

A random game with *linear structure* is built by restricting the transition relation as follows: a node v_i can make a transition to node v_j, where $0 \leq i, j \leq |\mathrm{P}| - 1$, if and only if $|i - j| \leq d$, where d is named as the *distance* parameter.

Table 1 collects the running time of the symbolic algorithms on random games with linear structures having priorities 2, 3, and 5, and distance $d = 25$. The results show that `SAPT` performs better than the others in solving games with $n \leq 10,000$ nodes and 2 priorities, while `SRE` is the best performing in all other cases. Also, they show that `SSP` and `SSP2` have the worst performances in all

[1] The tool is available for download from https://github.com/antoniodistasio/sympgsolver
[2] http://vlsi.colorado.edu/ fabio/CUDD/

	2 Pr				3 Pr				5 Pr			
n	SRE	SAPT	SSP	SSP2	SRE	SAPT	SSP	SSP2	SRE	SAPT	SSP	SSP2
1,000	0.04	0.03	29.89	0,95	0.05	0.10	18.9	1,44	0.05	0.45	15.75	$\texttt{abort}_T$
2,000	0.14	0.12	128.06	2,87	0.13	0.18	79.22	26,24	0.12	1.34	69.6	$\texttt{abort}_T$
3,000	0.25	0.23	$\texttt{abort}_T$	10,15	0.21	0.41	193.06	75,49	0.21	2.03	135.04	$\texttt{abort}_T$
4,000	0.33	0.30	$\texttt{abort}_T$	32,42	0.28	0.60	$\texttt{abort}_T$	146,58	0.3	3.01	$\texttt{abort}_T$	$\texttt{abort}_T$
7,000	0.79	0.73	$\texttt{abort}_T$	$\texttt{abort}_T$	0.65	1.44	$\texttt{abort}_T$	$\texttt{abort}_T$	0.59	7.20	$\texttt{abort}_T$	$\texttt{abort}_T$
10,000	1.16	1.12	$\texttt{abort}_T$	$\texttt{abort}_T$	0.93	2.19	$\texttt{abort}_T$	$\texttt{abort}_T$	1.08	11.72	$\texttt{abort}_T$	$\texttt{abort}_T$
20,000	2.78	3.10	$\texttt{abort}_T$	$\texttt{abort}_T$	2.33	6.34	$\texttt{abort}_T$	$\texttt{abort}_T$	3.69	43.87	$\texttt{abort}_T$	$\texttt{abort}_T$
100,000	19.21	24.4	$\texttt{abort}_T$	$\texttt{abort}_T$	24.38	65.11	$\texttt{abort}_T$	$\texttt{abort}_T$	24.89	$\texttt{abort}_T$	$\texttt{abort}_T$	$\texttt{abort}_T$

Table 1. Runtime executions of the symbolic algorithms

instances, with SSP overcoming SSP2 of more than 200 seconds on games with $3,000$ nodes. In Table 2 we collect the execution time of the explicit algorithms on the same set of games. The results highlight that the explicit algorithms are faster than the symbolic ones in all instances.

	2 Pr			3 Pr			5 Pr		
n	RE	APT	SPM	RE	APT	SPM	RE	APT	SPM
1,000	0.0008	0.0006	0.0043	0.0008	0.0007	0.0049	0.0008	0.0008	0.0053
2,000	0.0015	0.0012	0.0084	0.0017	0.0016	0.0096	0.0019	0.0029	0.011
3,000	0.0023	0.0017	0.012	0.0025	0.0022	0.014	0.0029	0.0073	0.020
4,000	0.0031	0.0022	0.016	0.0033	0.0028	0.019	0.0035	0.0066	0.027
7,000	0.0051	0.0039	0.025	0.0053	0.0048	0.032	0.0056	0.012	0.039
10,000	0.0065	0.0057	0.035	0.0067	0.0076	0.046	0.0069	0.018	0.051
20,000	0.013	0.011	0.078	0.014	0.021	8.32	0.17	0.019	107.2
100,000	0.094	0.081	0.44	0.099	0.10	1.47	0.10	0.59	80.37

Table 2. Runtime executions of the explicit algorithms

Ladder Games. In a ladder game, every node in P_i has priority i. In addition, each node $v \in \mathrm{P}$ has two successors: one in P_0 and one in P_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 m specifies the number of node pairs. Formally, a ladder game of index m is $\mathcal{G} = (\mathrm{P}_0, \mathrm{P}_1, Mv, \mathsf{p})$ where $\mathrm{P}_0 = \{0, 2, \ldots, 2m-2\}$, $\mathrm{P}_1 = \{1, 3, \ldots, 2m-1\}$, $Mv = \{(v, w) | w \equiv_{2m} v + i$ for $i \in \{1, 2\}\}$, and $\mathsf{p}(v) = v \ mod \ 2$. Tables 3 and 4 report the benchmarks.

m	SRE	SAPT	SSP	SSP2
1,000	0	0.00013	24.86	0.47
10,000	0.00009	0.00016	$\texttt{abort}_T$	41.22
100,000	0.0001	0.00018	$\texttt{abort}_T$	$\texttt{abort}_T$
1,000,000	0.00012	0.00022	$\texttt{abort}_T$	$\texttt{abort}_T$
10,000,000	0.00015	0.00025	$\texttt{abort}_T$	$\texttt{abort}_T$

Table 3. Runtime executions of the symbolic algorithms on ladder games.

m	RE	APT	SPM
1,000	0.0007	0.0006	0.002
10,000	0.006	0.005	0.0017
100,000	0.057	0.054	0.18
1,000,000	0.59	0.56	1.84
10,000,000	6.31	5.02	20.83

Table 4. Runtime executions of the explicit algorithms on ladder games.

Benchmarks indicate that SRE and SAPT outperform their explicit versions, showing an excellent runtime execution even on fairly large instances. Indeed, while RE needs 6.31 seconds for games with index $m = 10M$, SRE takes just 0.00015 seconds. Tests also show that SSP and SSP2 have yet the worst performance.

Clique Games. Clique games are fully connected games without self-loops, where P_0 (*resp.,* P_1) contains the nodes with an even index (*resp., odd*) and each node $v \in \mathrm{P}$ has as priority the index of v. An important feature of the clique games is the high number of cycles, which may pose difficulties for certain algorithms. Formally, a clique game of index n is $\mathcal{G} = (\mathrm{P}_0, \mathrm{P}_1, Mv, \mathsf{p})$ where $\mathrm{P}_0 = \{0, 2, \ldots, n-2\}$, $\mathrm{P}_1 = \{1, 3, \ldots, n-1\}$, $Mv = \{(v, w) | v \neq w\}$, and $\mathsf{p}(v) = v$. Benchmarks on clique games are reported in Tables 5 and 6.

n	SRE	SAPT	SSP	SSP2
2,000	0.007	0.003	5.53	$\mathtt{abort}_T$
4,000	0.018	0.008	19.27	$\mathtt{abort}_T$
6,000	0.025	0.012	39.72	$\mathtt{abort}_T$
8,000	0.037	0.017	76.23	$\mathtt{abort}_T$

Table 5. Runtime executions of the symbolic algorithms on clique games

n	RE	APT	SPM
2,000	0.021	0.0105	0.0104
4,000	0.082	0.055	0.055
6,000	0.19	0.21	0.22
8,000	0.35	0.59	0.63

Table 6. Runtime executions of the explicit algorithms on clique games

Benchmarks show that SAPT is the best one among the symbolic algorithms in all instances, SAPT and SRE outperform the explicit ones (as in ladder games), and the symbolic versions of SPM do not show good results even on small games.

Finally, we evaluate the symbolic and explicit approaches on some practical model checking problems as in [17]. 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), the Onebit Protocol (OP), and the Lifting Truck (Lift). The properties we check on these models concern: absence of deadlock (ND), a message of a certain type (d1) is received infinitely often (IORD1), if there are infinitely many read steps then there are infinitely many write steps (IORW), liveness, and safety. Note that, in all benchmarks, data size (DS) denotes the number of messages.

n	Pr	Property	SRE	SAPT	SSP	SSP2	RE	APT	SPM	WS	DS
14,065	3	ND	0.00009	0.00006	3.30	0.0001	0.004	0.004	0.029	2	2
17,810	3	IORD1	0.0003	0.0005	$\mathtt{abort}_T$	85.4	0.006	0.006	0.037	2	2
34,673	3	IORW	0.0006	0.0008	164.73	56.44	0.015	0.014	0.053	2	2
2,589,056	3	ND	0.0002	$\mathtt{abort}_T$	$\mathtt{abort}_T$	0.29	1.02	0.93	9.09	4	2
3,487,731	3	IORD1	$\mathtt{abort}_T$	$\mathtt{abort}_T$	$\mathtt{abort}_T$	$\mathtt{abort}_T$	1.81	1.4	17.45	4	2
6,823,296	3	IORW	0.3	$\mathtt{abort}_T$	$\mathtt{abort}_T$	$\mathtt{abort}_T$	3.87	3.13	22.26	4	2

Table 7. SWP (Sliding Window Protocol)

As we can see, by comparing Tables 7, 8, and 9, the experiments indicate more nuanced relationship between the symbolic and explicit approaches. Indeed,

n	Pr	Property	SRE	SAPT	SSP	SSP2	RE	APT	SPM	DS
81,920	3	ND	0.00002	31.69	1.37	0.0016	0.031	0.034	0.22	2
88,833	3	IORD1	0.0027	0.003	$\texttt{abort}_T$	$\texttt{abort}_T$	0.036	0.0038	0.27	2
170,752	3	IORW	14.37	98.4	$\texttt{abort}_T$	$\texttt{abort}_T$	0.07	0.07	0.47	2
289,297	3	ND	0.0001	154.89	12.3	0.0058	0.13	0.12	1.34	4
308,737	3	IORD1	0.0088	0.009	$\texttt{abort}_T$	$\texttt{abort}_T$	0.14	0.13	1.37	4
607,753	3	IORW	43.7	$\texttt{abort}_T$	$\texttt{abort}_T$	$\texttt{abort}_T$	0.29	0.27	2.06	4

Table 8. OP (Onebit Protocol)

n	Pr	Property	SRE	SAPT	SSP	SSP2	RE	APT	SPM	DS
328	1	ND	0.00002	0.002	0.005	0.00002	0.0001	0.0001	0.0004	2
308	1	safety	0.00002	0.003	0.028	0.00002	0.0001	0.0001	0.0004	2
655	3	liveness	0.00008	0.0001	5.52	0.09	0.0003	0.0002	0.001	2
51.220	1	safety	0.0001	1.48	32.14	0.00002	0.01	0.01	0.09	4
53.638	1	ND	0.0001	0.2	4.67	0.0001	0.017	0.015	0.07	4
107,275	3	liveness	0.005	0.001	$\texttt{abort}_T$	$\texttt{abort}_T$	0.03	0.03	0.18	4

Table 9. Lift (Lifting Truck)

they show a different behavior depending on the protocol and the property we are checking. Overall, we note that SRE outperforms the other symbolic algorithms in all protocols, although the advantage over RE is discontinued. Specifically, SRE is the best performing in checking absence of deadlock in all three protocols, but for IORD1 in the SWP protocol with $WS = 2$, or for IORW in the OP protocol, RE exhibits a significant advantage. Differently, SAPT and SSP2 show better performances on a smaller number of properties. Moreover, the results highlights that SSP exhibits the worst performances in all protocols and properties.

5 Concluding Remarks

In this paper we have compared for the first time the performances of different symbolic and explicit versions of classic algorithms to solve parity games. To this aim we have implemented in a fresh tool, which we have called SymPGSolver, the symbolic versions of Recursive [24], APT [10, 18], and the small-progress-measures algorithms presented in [5] and [8].

Our analysis started from constrained random games [21]. The results show that on these games the explicit approach is better than the symbolic one, exhibiting a different behavior than the one showed in [21]. To gain a fuller understanding of the performances of the symbolic and the explicit algorithms, we have further tested the two approaches on structured games. Precisely, we have considered ladder games, clique games, as well as game models coming from practical model-checking problems. We have showed several cases in which the symbolic algorithms have the advantage over the explicit ones.

Our empirical study let us to conclude that on comparing explicit and symbolic algorithms for solving parity games, it would be useful to have real scenarios and not only random games, as the common practice has been.

References

1. R. Iris Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. *Formal Methods in System Design*, pages 171–206, 1997.
2. Marco Bakera, Stefan Edelkamp, Peter Kissmann, and Clemens D. Renner. Solving μ-calculus parity games by symbolic planning. In *MoChArt 2008*, pages 15–33, 2008.
3. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, pages 677–691, 1986.
4. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^20 states and beyond. In *LICS 1990*, pages 428–439, 1990.
5. D. Bustan, O. Kupferman, and M. Y. Vardi. A measured collapse of the modal μ-calculus alternation hierarchy. In *STACS 2004*, pages 522–533, 2004.
6. C. S. Calude, S. Jain, B. Khoussainov, W. Li, and F. Stephan. Deciding parity games in quasipolynomial time. In *STOC 2017*, pages 252–263, 2017.
7. P. Cermák, A. Lomuscio, and A. Murano. Verifying and synthesising multi-agent systems against one-goal strategy logic specifications. In *AAAI 2015*, pages 2038–2044, 2015.
8. K. Chatterjee, W. Dvorák, M. Henzinger, and V. Loitzenbauer. Improved set-based symbolic algorithms for parity games. In *CSL 2017*, pages 18:1–18:21, 2017.
9. E.M. Clarke and E.A. Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *LP 1981*, LNCS 131, pages 52–71, 1981.
10. A. Di Stasio, A. Murano, G. Perelli, and M. Y. Vardi. Solving parity games using an automata-based algorithm. In *CIAA 2016*, pages 64–76, 2016.
11. C. Eisner and D. A. Peled. Comparing symbolic and explicit model checking of a software system. In *SPIN 2002*, pages 230–239, 2002.
12. E.A. Emerson and C. Jutla. Tree Automata, μ-Calculus and Determinacy. In *FOCS 1991*, pages 368–377, 1991.
13. M. Jurdzinski. Deciding the Winner in Parity Games is in UP $\cap$ co-Up. *Inf. Process. Lett.*, 68(3):119–124, 1998.
14. M. Jurdzinski.Small Progress Measures for Solving Parity Games. In *STACS 2000*, LNCS 1770, pages 290–301, 2000.
15. M. Jurdzinski and R. Lazic. Succinct progress measures for solving parity games. In *LICS 2017*, pages 1–9, 2017.
16. G. Kant and J. van de Pol. Generating and solving symbolic parity games. In *GRAPHITE 2014*, pages 2–14, 2014.
17. J.J. A. Keiren. Benchmarks for parity games. In *FSEN 2015*, pages 127–142, 2015.
18. O. Kupferman and M. Y. Vardi. Weak Alternating Automata and Tree Automata Emptiness. In *STOC 1998*, pages 224–233, 1998.
19. O. Kupferman, M.Y. Vardi, and P. Wolper. An Automata Theoretic Approach to Branching-Time Model Checking. *Journal of the ACM*, 47(2):312–360, 2000.
20. K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
21. D. Tabakov. Evaluation of explicit and symbolic automata-theoretic algorithm. Master's thesis, Rice University, 2005.
22. Tom van Dijk. Oink: An implementation and evaluation of modern parity game solvers. In *TACAS 2018*, LNCS 10805, pages 291–308. Springer, 2018.
23. T. Wilke. Alternating Tree Automata, Parity Games, and Modal μ-Calculus. *Bulletin of the Belgian Mathematical Society Simon Stevin*, 8(2):359, 2001.
24. W. Zielonka. Infinite Games on Finitely Coloured Graphs with Applications to Automata on Infinite Trees. *Theor. Comput. Sci.*, 200(1-2):135–183, 1998.