BDD-Based Parity Game Solving

A Comparison of Zielonka's Recursive Algorithm and Priority Promotion

Lisette Sanchez, Wieger Wesselink, and Tim A.C. Willemse

Eindhoven University of Technology, The Netherlands

Abstract. Parity games are two player games with omega-winning conditions, played on finite graphs. Several algorithms for solving parity games have been proposed in the literature, and while the problem was recently shown to be solvable in quasi-polynomial time, so far, the question whether such games can be solved in polynomial time remains elusive. In practice, parity games play an important role in verification, satisfiability and synthesis. It is therefore important to identify algorithms that can efficiently deal with large and complex games that arise from such applications. In this paper, we describe our experiments with BDD-based implementations of Zielonka's recursive algorithm and the more recent Priority Promotion algorithm. We conclude that overall, Zielonka's BDD-based algorithm beats the BDD-based Priority Promotion algorithm by a small margin for games that are characteristic of practical verification problems.

1 Introduction

Parity games [10, 18, 21] are infinite duration games played by two player on a finite directed graph. Each vertex in the game graph is owned by one of the two players and vertices are assigned a colour, or *priority.* The game is played by pushing a single token along the edges in the graph; the choice to which vertex the token is to move next is decided by the player owning the vertex currently holding the token. A parity winning condition determines the winner of this infinite play. A vertex in the game is won by the player that has a strategy for which, no matter how the opponent plays, every play from that vertex is won by her; the winner of each vertex is uniquely determined [18]. The parity game solving problem is to compute the set of vertices won by each player. From a practical viewpoint, parity games are interesting since they underlie typical verification, satisfiability and synthesis problems, see [7, 10, 1].

The simplicity of the game can be deceptive. In spite of a continued research effort, no polynomial algorithm for solving such games has been found. The problem of solving a parity game is known to be in $\mathrm{UP} \cap \mathrm{coUP}$ [13], a class that neither precludes nor predicts the existence of a polynomial algorithm. In recent years, several new algorithms have been devised for solving parity games, and only last year, the problem was shown to run in *quasi-polynomial time* [6].

Most parity game solving algorithms fall in one of two categories: 'strategy identification' (SI) algorithms and 'dominion identification' (DI) algorithms. The SI category of algorithms directly computes the winning strategies for both players; *e.g.* by means of policy iteration or by maintaining some statistics about the gameplays that can emerge from a vertex. The recent quasi-polynomial algorithms [6, 15] all fall in this category, but also classical algorithms such as Jurdziński's *small progress measures* algorithm [14]. The DI category of algorithms all proceed by (recursively) decomposing the game graph in *dominions*: small subgraphs that are won by a single player and from which the opponent cannot escape. A classical exponent of the latter category is Zielonka's *recursive algorithm* [21], but also the recently introduced Priority Promotion algorithm [5]. While DI algorithms typically have a worst-case running time complexity that is theoretically less attractive than that of SI algorithms, in practice, the SI are significantly outperformed by DI algorithms [19].

Since parity games that originate from practical verification problems can become quite large, it is natural to study techniques that help to solve such games, relying on compact, symbolic representations of the game graph such as Binary Decision Diagrams (BDDs). Unlike SI algorithms, DI algorithms typically compute with subgraphs and as such are likely candidates admitting a symbolic implementation. Indeed, several BDD-based implementations of Zielonka's algorithm have been studied in the past [2].

In this paper, we describe our implementation of the PP algorithm using BDDs. Moreover, we compare its performance to a BDD-based implementation of Zielonka's algorithm. In particular, we reassess the conclusions from [19] in which Zielonka's algorithm and the PP algorithm were shown to have similar performance when computing with a non-symbolic representation of the game graph. A few samples taken from a set of benchmarks for parity games encoding verification problems point at a much better scalability of our BDD-based solvers, compared to their explicit counterparts. Apart from comparing the performance on games originating from typical verification problems, we have assessed the performance of both BDD-based implementations by generating BDDs representing random game graphs. While we confirm that also in the symbolic setting the two algorithms perform similarly, our observations indicate that for games with a modest number of priorities in the game, Zielonka beats the PP algorithm by a small margin.

This paper is structured as follows. In Section 2, we introduce parity games and the relevant concepts. The two algorithms that we compare are then introduced and discussed in Section 3 and we describe how these can be implemented using BDD techniques in Section 4. In Section 5, we describe our experimental evaluation of our implementations and we finish with conclusions in Section 6.

2 Parity Games

A parity game is an infinite duration game, played by players *odd*, denoted by 1 and *even*, denoted by 0, on a directed, finite graph. The game is formally defined as follows.

Definition 1 (Parity game). *A parity game is a tuple* $(V, E, p, (V_0, V_1))$, *where:*

- V *is a finite set of vertices, partitioned in a set* V_0 *of vertices owned by player* 0*, and a set of vertices* V_1 *owned by player* 1*,*
- $E \subseteq V \times V$ *is the edge relation; we assume that* E *is total,* i.e. *for all* $v \in V$*, there is some* $w \in V$ *such that* $(v, w) \in E$*,*
- $p{:}V \to \mathbb{N}$ *is a priority function that assigns priorities to vertices.*

We depict parity games as graphs in which diamond-shaped vertices represent vertices owned by player 0 and box-shaped vertices represent vertices owned by player 1. Priorities, associated with vertices, are typically written inside vertices, see Figure 1.

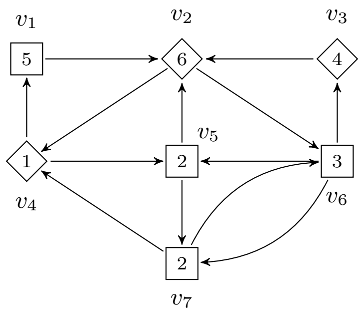

Fig. 1. A parity game with 7 vertices, 6 different priorities, 3 vertices owned by player 0 and 4 vertices owned by player 1.

Plays, Strategies and Winning. We write $v \to w$ whenever $(v, w) \in E$. Henceforth, $\alpha \in \{0, 1\}$ denotes an arbitrary player. We write $\bar{\alpha}$ for α's opponent; *i.e.* $\bar{0} = 1$ and $\bar{1} = 0$. A sequence of vertices $v_1, \ldots, v_n$ is a *path* if $v_m \to v_{m+1}$ for all $1 \leq m < n$. Infinite paths are defined in a similar manner. We write π_n to denote the n^{th} vertex in a path π.

A game starts by placing a token on vertex $v \in V$. Players move the token indefinitely according to a single simple rule: if the token is on some vertex $v \in V_\alpha$, player α gets to move the token to an adjacent vertex. The *parity* of the highest priority that occurs infinitely often on a thus constructed infinite sequence of vertices through the game defines the *winner* of the path: player 0 wins if, and only if this priority is even.

More formally, the moves of players 0 and 1 are determined by their respective *strategies*. Informally, a strategy for a player α determines, for a vertex $\pi_i \in V_\alpha$ the next vertex π_{i+1} that will be visited if the token visits π_i. In general, a strategy is a partial function $\sigma{:}V^* \times V_\alpha \to V$ which, given a history of vertices visited by the token and a vertex on which the token currently resides, determines the next vertex. For simplicity and (due to the positional determinacy theorem for parity games [10]) without a loss of generality, we only consider *positional strategies*. A positional strategy (hereafter simply referred to as *strategy*) for a player α is a partial function $\sigma{:}V_\alpha \to V$ that is compatible with E, *i.e.* for all $v \in V_\alpha$ it is the case that $\sigma(v) \in \{w \in V \mid v \to w\}$. An infinite path π is *compatible* with a given strategy σ if for all vertices π_i for which σ is defined, we have $\pi_{i+1} = \sigma(\pi_i)$. We refer to an infinite path through the graph that is compatible with a strategy for player 0 and a strategy for player 1 as a *play*. A strategy is *closed* on a set U iff every play that is compatible with that strategy remains within U. We say that a set of vertices U is α-*closed* iff α has a strategy that is closed on U.

A play is won by player 0 iff the *maximal* priority that occurs infinitely often along that play is *even*. More formally, given a play π, player 0 wins π iff:

$$\max\{k \in \mathbb{N} \mid \forall j \in \mathbb{N} : \exists i \in \mathbb{N} : i > j \wedge k = p(\pi_i)\} \text{ is } \textit{even}$$

Strategy σ for player α is *winning* from a vertex v if and only if α is the winner of every play starting in v that is compatible with σ. A set of vertices W_α is won by α if for each vertex $v \in W_\alpha$, player α has some winning strategy from v. Another consequence of the aforementioned positional determinacy theorem is that the set of vertices W_0 won by player 0 and the set of vertices W_1, won by player 1, forms a partition of the set V: every vertex is won by exactly one player.

Example 1. Consider the parity game depicted in Figure 1. The vertices v_5, v_6 and v_7 are won by player 1 whereas vertices v_1, v_2, v_3 and v_4 are won by player 0. To see why vertex v_6 and v_7 are won by player 1, consider her strategy σ defined by $\sigma(v_7) = v_6$ and $\sigma(v_6) = v_5$. Any play that is compatible with this strategy infinitely often visits vertex v_6, which has a dominating odd priority. Hence, such plays are won by player 1. Furthermore, note that this strategy is *closed* on $\{v_6, v_7\}$. □

Subgames, Attractors and Dominions. For a game $G = (V, E, p, (V_0, V_1))$ and a set $U \subseteq V$, we define the *subgame* of G, denoted $G \cap U$, as the maximal substructure $(V', E', p', (V'_0, V'_1))$ that is obtained by restricting the graph (V, E) to U and lifting the restriction to the other elements of G, *i.e.* $V' = U$, $E' = E \cap (U \times U)$, $p'(v) = p(v)$ for all $v \in U$, and $V'_0 = V_0 \cap U$ and $V'_1 = V_1 \cap U$. If the edge relation E' of $G \cap U$ is again total, then the subgame is again a parity game. We furthermore use the abbreviation $G \setminus A$ to denote the subgame $G \cap (V \setminus A)$.

Parity game solving can essentially be reduced to identifying or computing appropriate subgraphs that are entirely won by a single player and from which the opponent cannot escape. A subgame that has this property for a player α is called an α-*dominion*. Technically, an α-dominion is a non-empty set of vertices $D_\alpha \subseteq V$ such that player α has a winning strategy from *all* vertices in the set D_α that is closed on D_α. We note that W_0, the set of vertices won by player 0, is a 0-dominion; likewise W_1 is a 1-dominion.

Example 2. Reconsider the parity game from Figure 1. We already identified a closed strategy that is winning for player 1 on the set of vertices $\{v_5, v_6\}$. So this set is a 1-dominion. This is not a maximal 1-dominion: the winning set $\{v_5, v_6, v_7\}$ is also a 1-dominion. $\square$

For a non-empty set $U \subseteq V$ and a player α, the α-attractor into U, denoted $Attr_\alpha(U)$, is the least set of vertices for which player α can force play into U. We define $Attr_\alpha(U)$ as $\bigcup\limits_{i \geq 0} Attr^i_\alpha(U)$, the limit of approximations of the sets $Attr^n_\alpha(U)$, which are inductively defined as follows:

$$\begin{aligned}
Attr^0_\alpha(U) &= U \\
Attr^{n+1}_\alpha(U) &= Attr^n_\alpha(U) \\
&\cup \{v \in V_\alpha \mid \exists u \in Attr^n_\alpha(U) :\ v \to u\} \\
&\cup \{v \in V_{\bar{\alpha}} \mid \forall u \in V : v \to u \implies u \in Attr^n_\alpha(U)\}
\end{aligned}$$

The *confined* α-attractor, denoted $Attr_\alpha(T, U)$, is defined analogously by limiting the set of vertices to those from T in each approximation. That is, it represents the subset of vertices $T' \subseteq T$ from which α can force play to U while remaining in T. We note that for arbitrary set $U \subseteq V$, the subgame $G \setminus Attr_\alpha(U)$ is again a parity game; that is, the edge relation of the subgame is again total. Furthermore, the α-attractor into an α-dominion D_α is again an α-dominion. Finally, observe that the set $V \setminus Attr_\alpha(U)$ is $\bar{\alpha}$-closed.

Example 3. The 0-attractor into the set $\{v_2\}$ contains the vertices v_1 (which is owned by 1 but trivially attracted to v_6 since it has but one successor vertex), v_3 and v_4. Vertex v_5 does not belong to the 0-attractor into $\{v_2\}$ as player 1 can choose to move to v_7. For similar reasons, vertices v_6 and v_7 do not belong to the 0-attractor into $\{v_2\}$. Note that the set $\{v_5, v_6, v_7\}$ is 1-closed. $\square$

3 Two Parity Game Solving Algorithms

Algorithms for solving parity games essentially either use clever statistics for identifying winning strategies, or they employ a decomposition of the game play by identifying dominions in subgames. In this paper we focus on two algorithms that fit in the latter class.

3.1 Zielonka's Recursive Algorithm

The algorithm by Zielonka is a divide and conquer algorithm that searches for dominions in subgames. The algorithm is essentially distilled from a constructive proof of the positional determinacy of parity games by, among others, Zielonka [21]. Despite the fact that the algorithm has a relatively bad theoretical worst-case complexity (it runs in $\mathcal{O}(mn^d)$ where n is the number of vertices, m the number of edges and d is the number of different priorities in the game) and exponential worst-case examples are known for various classes of special games [12], the algorithm remains among the most successful solvers for parity games in practice, see [11] and the recent exploration [19].

Zielonka's algorithm (see Algorithm 1) constructs winning regions for both players out of the solution of subgames with fewer different priorities and fewer vertices. For this, the algorithm relies on the fact that higher priorities in the game dominate lower priorities, and that any forced revisit of these higher priorities is beneficial to the player aligning with the parity of the priority. For non-trivial games, the algorithm relies on attractor set computations (see line 8) to identify the set of vertices that can be forced to visit the maximal priority in the game; the remaining vertices are then solved recursively (see line 9).

The outcome of the recursion can be either that the entire subgame is won by the player (say α) that has the same parity as the maximal priority in the game *or* the opponent $\bar{\alpha}$ wins a non-empty set of vertices. In case α wins all vertices in the subgame, she wins the entire game (lines 11-12). This can be seen as follows: since the subgame is $\bar{\alpha}$-closed, the opponent can choose to stay within the subgame; choosing to do so means she will lose. So the only option she has is to escape. But the only escape she has leads to the maximal priority which has the parity of α. Note that if α is forced to leave this maximal priority she will again end up in the subgame; any play that does so *ad infinitum* visits the maximal priority infinitely often and is therefore won by α. In case α does *not* win the entire subgame, the set of vertices won by $\bar{\alpha}$ in the subgame are also won by $\bar{\alpha}$ in the larger game since the subgame was $\bar{\alpha}$-closed; in other words, it is a $\bar{\alpha}$-dominion and so is its $\bar{\alpha}$-attractor (the set B computed in line 13). Removing this dominion and recursively solving the remaining subgame (line 14) then leads to a solution of the entire game.

Several optimisations can be applied to Zielonka's algorithm. For instance, rather than solving the entire game at once, one can first decompose a game into strongly connected components and first solve the bottom components. The SCC decomposition can be integrated tightly in the algorithm so that in each recursive call first an SCC decomposition is performed. While this may sound expensive, in [12] it is shown that this actually allows the algorithm to run in polynomial time on many practically relevant classes of special games such as solitaire and dull games for which the original algorithm might require exponential time otherwise. Another observation, made in *e.g.* [17] is that in case $W_{\bar{\alpha}} = \mathit{Attr}_{\bar{\alpha}}(W_{\bar{\alpha}})$ after the first recursive call, no second recursive call is needed.

3.2 Priority Promotion

The recent *Priority Promotion* (PP) algorithm [5] starts, like Zielonka's recursive algorithm, with a game decomposition that aims at identifying dominions for a given player. In contrast to Zielonka's algorithm, however, the PP algorithm does not solve explicit subgames. Instead, within a fixed game, it uses a dominion

Algorithm 1 Zielonka's Algorithm

```
function Zielonka(G)
    if $V = \emptyset$ then
        $(W_0, W_1) \leftarrow (\emptyset, \emptyset)$
    else
        $m \leftarrow \max\{p(v) \mid v \in V\}$
        $\alpha \leftarrow m \mod 2$
        $U \leftarrow \{v \in V \mid \Omega(v) = m\}$
        $A \leftarrow Attr_\alpha(U)$
        $(W'_0, W'_1) \leftarrow$ Zielonka($G \setminus A$)
        if $W'_{\bar{\alpha}} = \emptyset$ then
            $(W_\alpha, W_{\bar{\alpha}}) \leftarrow (A \cup W'_\alpha, \emptyset)$
        else
            $B \leftarrow Attr_{\bar{\alpha}}(W'_{\bar{\alpha}})$
            $(W'_0, W'_1) \leftarrow$ Zielonka($G \setminus B$)
            $(W_\alpha, W_{\bar{\alpha}}) \leftarrow (W'_\alpha, W'_{\bar{\alpha}} \cup B)$
        end if
    end if
    return $(W_0, W_1)$
end function
```

searcher that maintains a set of vertices, along with an updated (promoted) priority mapping and zooms in on a dominion within that set of vertices. Once an α-dominion is returned by the dominion searcher, its α-attractor is removed from the game and the search for another dominion continues, see Algorithm 2, until the entire game is solved.

Algorithm 2 Priority Promotion Solver

```
function SolvePP($G$)
    $(W_0, W_1) \leftarrow (\emptyset, \emptyset)$
    repeat
        $m \leftarrow \max\{p(v) \mid v \in V\}$
        $(W'_0, W'_1) \leftarrow$ SearchDominion($G, V, p, m$)
        $(W_0, W_1) \leftarrow (W_0 \cup W'_0, W_1 \cup W'_1)$
        $G \leftarrow G \setminus (W'_0 \cup W'_1)$
    until $V = \emptyset$
    return $(W_0, W_1)$
end function
```

The main complexity and novelty of the PP algorithm lies in the way it identifies a dominion. To this end, it relaxes the notion of a dominion to a *quasi* α-dominion. A quasi α-dominion is a set U of vertices for which α has a strategy that guarantees that every play that remains within U is won by α, or that exits U via the $\bar{\alpha}$-*escape* of U. The α-escape of a set U, denoted $esc^\alpha_G(U)$ is the set of vertices from which α can force play out of U in a single move; *i.e.* $v \in esc^\alpha_G(U)$, for a game G holds iff $v \in V_\alpha$ and for all w for which $v \to w$ we have $w \notin U$, or $v \in V_{\bar{\alpha}}$ and there is some $w \notin U$ such that $v \to w$. A quasi α-dominion U in a game G is said to be *α-closed* iff $esc_G^{\bar{\alpha}}(U) = \emptyset$; otherwise it is said to be α-open. Note that a α-closed quasi α-dominion is a α-dominion. When the $\bar{\alpha}$-escape set of a quasi α-dominion only contains vertices with the highest priority in the game, the α quasi-dominion is called a *α-region*.

By searching for quasi-dominions, the algorithm avoids the 'hard' problem of identifying the (set of) priorities that lead to a player winning its dominion. Under certain conditions, these quasi-dominions offer just enough guarantees that when composed, the result is a dominion. Regions meet these conditions. The dominion searcher, see Algorithm 3, essentially traverses a partial order of *states* which contain information about the open regions the searcher has computed up to that point. Once the searcher finds a closed region it terminates and returns this dominion to the solver. Conceptually the searcher assigns to each α-region it finds a priority that under-approximates the best value the opponent $\bar{\alpha}$ must visit when escaping from the α-region. A higher α-region U_1 can then be merged with a lower region U_2, yielding a new α-region and improving the under-approximation of U_2 by promoting its best escape value.

The searcher first checks whether the set of vertices U that dominate the current state induces a closed region A within the entire game; if so, the region is returned and the search is finished (lines 2-8). If the region is open, the opponent $\bar{\alpha}$ may escape. When the region is open in the current state, she may try to escape to some inferior priority within the subgraph; if the region is closed in the current state, she can only escape to some region in the larger game and end up in a priority that dominates m_g.

In case the region is open in the current state (line 11-14), the priority function of the current state is updated to set all vertices in A to the currently dominating priority m_g. This is achieved by the update $p_g^* \leftarrow p_g[U \mapsto m_g]$ which is defined as $p_g^*(v) = m_g$ in case $v \in U$ and $p_g^*(v) = p_g(v)$ otherwise. The new priority for the subgraph that will be explored is set to the next largest priority in the graph (line 13) and A is removed from the subgraph (line 14). This new state is then explored recursively (see line 20). In case the region is closed in the current state (line 16-18), the opponent must escape (if she wants to) to vertices in the larger game. The *best escape priority* she can force is the minimal priority among the set of vertices that she can force play to in a single step. This is given by the function $bep_{p_g}^{\alpha}(A)$ which yields the minimum priority (according to p_g) of the set $\{w \in V \setminus A \mid \exists v \in A \cap V_\alpha : v \to w\}$. The dominating priority m_g is updated to m_g^* to reflect this best escape priority (see line 16) after which p_g is updated to p_g^*. In this update, all vertices in A are set to priority m_g^* while all vertices with priorities exceeding m_g^* remain unchanged. All vertices with priorities dominated by m_g^* are reset to their original value. This is achieved by the update in line 17. Here, $p_g^{\geq m}$ yields the partial function that coincides with p_g on the (maximal) domain $V' \subset V$ for which $p_g(v) \geq m$ for all $v \in V'$, and is undefined elsewhere. The *update* $p \uplus p_g$, for a partial function p_g is then defined as $(p \uplus p_g)(v) = p(v)$ in case $v \notin \mathsf{dom}(p_g)$ and $p_g(v)$ otherwise. The subgraph that is explored next is set to all vertices with (promoted) priority no larger than m_g^*. This newly constructed state is then again recursively explored (see line 20). The PP algorithm resets all information in lower regions when promoting vertices. As observed in [3, 4], this can be improved by only resetting regions of the opponent, or even a subset of that. While this affects the performance for some artificial worst-case examples, these improvements do not lead to improvements on games stemming from practical applications, nor on random games [19].

Algorithm 3 Priority Promotion Dominion Searcher

```
 1: function SearchDominion($G, V_g, p_g, m_g$)
 2:     $\alpha \leftarrow m_g \mod 2$
 3:     $U \leftarrow \{v \in V_G \mid p_g(v) = m_g\}$
 4:     $A \leftarrow Attr_\alpha(V_g, U)$
 5:     $X \leftarrow esc_G^{\bar{\alpha}}(A)$
 6:     if $X = \emptyset$ then
 7:         $(W'_\alpha, W'_{\bar{\alpha}}) \leftarrow (Attr_\alpha(A), \emptyset)$
 8:         return $(W'_0, W'_1)$
 9:     else
10:         $X \leftarrow esc_{G \cap V_g}^{\bar{\alpha}}(A)$
11:         if $X \neq \emptyset$ then
12:             $p_g^* \leftarrow p_g[A \mapsto m_g]$
13:             $m_g^* \leftarrow \max\{k \mid k < m_g \wedge \exists v \in V_g : p_g^*(v) = k\}$
14:             $V_g^* \leftarrow V_g \setminus A$
15:         else
16:             $m_g^* \leftarrow bep_{p_g}^{\bar{\alpha}}(A)$
17:             $p_g^* \leftarrow (p \uplus p_g^{\geq m_g^*})[A \mapsto m_g^*]$
18:             $V_g^* \leftarrow \{v \in V \mid p_g^*(v) \leq m_g^*\}$
19:         end if
20:         return SearchDominion($G, V_g^*, p_g^*, m_g^*$)
21:     end if
22: end function
```

4 Implementing Parity Game Solvers Using BDDs

An explicit representation of a parity game G that encodes a verification problem of a (software or hardware) system quickly requires too much main memory. A symbolic representation of the game graph may then help to sidestep this problem. Binary Decision Diagrams (BDDs) are essentially a clever data structure that can be used to concisely and canonically represent propositional formulae, which in turn can be used to characterise sets. In our setting we have to represent the following sets: the set of vertices V of a game, the edges E, the partition (V_0, V_1) of V and the priority function p. For most parity games stemming from practical verification problems, the domain of p is small. For that reason, it suffices to represent p as a collection of ordered sets.

We assume that the reader is familiar with BDDs and the algorithms manipulating these; for a comprehensive treatment we refer to [20, 9]. For the computations involved in the parity game solving algorithms of the previous section we provide the key ingredients below; each operation is described using an expression for the BDD-based implementation.

We assume that $\boldsymbol{x}$ is the vector of Boolean variables used to span the set of vertices V; we assume ι is a total injective mapping from V to truth-assignments for $\boldsymbol{x}$. The i-th Boolean variable of $\boldsymbol{x}$ is denoted $\boldsymbol{x}_i$; this notation extends to other vectors of Boolean variables. We represent a set $U \subseteq V$ by a propositional formula encoded as a BDD $\mathcal{U}(\boldsymbol{x})$ that ranges over $\boldsymbol{x}$; as is standard, for a truth-assignment $\boldsymbol{u}$ for $\boldsymbol{x}$ we have $\mathcal{U}(\boldsymbol{u}) = 1$ iff $\iota(v) = \boldsymbol{u}$ for some $v \in U$. This way, we can represent, *e.g.* the sets V_0 and V_1 by BDDs $\mathcal{V}_0(\boldsymbol{x})$ and $\mathcal{V}_1(\boldsymbol{x})$. Note that we only need representations of two of the three sets V, V_0 and V_1; the third can be constructed efficiently by computing the complement of the BDD. The set of edges is given by a BDD $\mathcal{E}(\boldsymbol{x}, \boldsymbol{x}')$ that ranges over $\boldsymbol{x}$ and $\boldsymbol{x}'$, where $\boldsymbol{x}'$ is the vector of Boolean variables used to represent a successor vertex. Finally, the collection of BDDs $\mathcal{P}_i(\boldsymbol{x})$ represents the set of vertices with priority i.

We first highlight the operations required for implementing Zielonka's algorithm using BDDs. Set comparison, intersection, union and complement are easily expressed as BDD operations using equivalence, conjunction, disjunction and negation. The main computational problem in Zielonka's algorithm is therefore the attractor set computation and computing a subgame. Since subgames are defined using set intersection, the operations involved when computing a subgame require conjunction of BDDs only; we omit further details. For the attractor set computation, which is also used in the PP algorithm, we first define the *pre-image* operation using BDDs; that is, given a set $U \subseteq V$ and a player α, we encode the operation $pre_\alpha(U)$ which is defined as follows:

$$pre_\alpha(U) = \{v \in V_\alpha \mid \exists u \in U : v \to u\} \cup \{v \in V_{\bar{\alpha}} \mid \forall u \in V : v \to u \Rightarrow u \in U\}$$

Observe that we can restate this as follows:

$$pre_\alpha(U) = \{v \in V_\alpha \mid \exists u \in U : v \to u\} \cup \{v \in V_{\bar{\alpha}} \mid \neg\exists u \in V \setminus U : v \to u\}$$

Representing the above operation using standard BDD operations such as disjunction, conjunction, negation and existential quantification is then as follows:

$$\begin{array}{l} pre_\alpha(\mathcal{U}(\boldsymbol{x})) \\ = \\ (\mathcal{V}_\alpha(\boldsymbol{x}) \wedge \exists \boldsymbol{x}'.(\mathcal{E}(\boldsymbol{x}, \boldsymbol{x}') \wedge \mathcal{U}(\boldsymbol{x}'))) \vee (\mathcal{V}_{\bar{\alpha}}(\boldsymbol{x}) \wedge \neg\exists \boldsymbol{x}'.(\mathcal{E}(\boldsymbol{x}, \boldsymbol{x}') \wedge \neg\mathcal{U}(\boldsymbol{x}'))) \end{array}$$

Using the pre-image, the α-attractor $Attr_\alpha(\mathcal{U}(\boldsymbol{x}))$ (*cf.* Page 2) is then effectively implemented using BDDs. The confined α-attractor $Attr_\alpha(\mathcal{T}(\boldsymbol{x}), \mathcal{U}(\boldsymbol{x}))$, used in the PP algorithm (see line 4) is defined analogously using an additional conjunction operator on BDDs. Note also that the operation $esc^\alpha_G(A)$, used in lines 5 and 10, can be implemented using the pre-image operation.

The operations in the PP algorithm that remain to be encoded involve the computation of the best escape priority and the computation of new priority mappings. For the best escape priority we require to compute minimal priority among a set of successor vertices of a given set of vertices U. We split this computation in two parts. First we identify the set of successor vertices using the operation $\mathit{post}(U)$ defined as follows:

$$\mathit{post}(U) = \{v \in V \mid \exists u \in U : u \to v\}$$

This can effectively be encoded as follows:

$$\mathit{post}(\mathcal{U}(\boldsymbol{x})) = (\exists \boldsymbol{x}.\mathcal{E}(\boldsymbol{x}, \boldsymbol{x}') \wedge \mathcal{U}(\boldsymbol{x}))[\boldsymbol{x}' \mapsto \boldsymbol{x}]$$

Computing $\mathit{bep}^{\alpha}_{p_g}(A)$, see line 16, then can be implemented by searching for the least value m in the domain of mapping p_g for which the BDD $\neg\mathcal{A}(\boldsymbol{x}) \wedge p_g(m) \wedge \mathit{post}(\mathcal{A}(\boldsymbol{x}) \wedge \mathcal{V}_\alpha(\boldsymbol{x}))$ is satisfiable. This can be done using a simple iteration. Finally, the computations of p^*_g in lines 12 and 17 are simple updates of the function p_g which can be done by iterating over the domain of the mapping p (resp. p_g) and computing new BDDs using conjunction and disjunction of the BDDs given by p and p_g. More specifically, for updates of the form $(p \uplus p_g^{\geq m})[A \mapsto m]$, we first compute the set of vertices in the domain of p_g that are above m, assuming we have BDDs $\mathcal{P}_{g,i}(\boldsymbol{x})$ representing the set of vertices $\{v \in V \mid p_g(v) = i\}$:

$$\mathcal{A}\mathit{bove}_{\geq m}(\boldsymbol{x}) = \bigvee\{\mathcal{P}_{g,i}(\boldsymbol{x}) \mid i \geq m\}$$

The updated priority function p^*_g coinciding with $(p \uplus p_g^{\geq m})[A \mapsto m]$, where A is encoded by BDD $\mathcal{A}(\boldsymbol{x})$ and the set of vertices $\{v \in V \mid p(v) = i\}$ is encoded by $\mathcal{P}_i(\boldsymbol{x})$ is then, for each value $i \neq m$, given by:

$$\neg\mathcal{A}(\boldsymbol{x}) \wedge ((\mathcal{P}_i(\boldsymbol{x}) \wedge \neg\mathcal{A}\mathit{bove}) \vee (\mathcal{P}_{g,i}(\boldsymbol{x}) \wedge \mathcal{A}\mathit{bove}))$$

whereas for value $i = m$ we have:

$$\mathcal{A}(\boldsymbol{x}) \vee (\mathcal{P}_i(\boldsymbol{x}) \wedge \neg\mathcal{A}\mathit{bove}) \vee (\mathcal{P}_{g,i}(\boldsymbol{x}) \wedge \mathcal{A}\mathit{bove})$$

5 Experimental Evaluation

We have implemented Zielonka's algorithm and the PP algorithm in Python, utilising a BDD package[1] which, next to a native Python BDD implementation, offers wrappers to C-based BDD implementations such as CUDD, Sylvan and BuDDy. For the timings we report on for our experiments we rely on the native Python implementation as our independent experiments indicate that its performance is comparable to that of CUDD. Note that the choice of programming language is secondary given that all essential time-consuming computations involve BDD creation and manipulation. We conducted all experiments on a Macbook Pro, 3.5 GHz Intel Core i7 (13-inch, 2017 model, *i.e.* dual core), with 16 Gb 2133 MHz LPDDR3 main memory, running macOS 10.13.2.

[1] See https://github.com/johnyf/dd, version 0.5.4, by Ioannis Filippidis

We compare the performance of our implementation of Zielonka's algorithm and the PP algorithm as described in Sections 3 and 4. To analyse the performance of both algorithms on parity games beyond those from a fixed benchmark set, we analyse their running times on randomly generated games. Such games, however, may be poor predictors for the performance of the algorithms on practical problems. Since one of our aims is to assess the speed at which BDD-based solvers can solve parity games that encode practical verification problems, we also compare their performance on some of the cases of the Keiren benchmark set [16].

5.1 Random Parity Games

The PGSolver [11] is a collection of parity game solving algorithms programmed in OCaml, which includes tools to generate self-loop-free random parity games with a fixed number of vertices. Since the tool generates explicit graphs in the PGSolver format, we automatically converted these games to BDDs using a binary encoding for the vertices.

Unfortunately (but not unexpected), our experiments indicate that for games that consist of more than 2000 vertices, the BDD approach starts to fail dramatically; the BDD representations of the games become overly complex and unstructured, leading to excessive running times of both our Zielonka and PP implementations. The main reason for the poor performance is the lack of structure provided by the binary encoding of vertices: typically, BDDs remain concise if two adjacent vertices (*i.e.* vertices related by E) only differ in their representation by a relatively small number of bits. The binary encoding, however, does not guarantee this.

In view of the above observation, we resort to another approach for generating random games and testing the scalability of both implementations. Instead of generating explicit games, we generate random BDDs representing games of a fixed size but with different characteristics. We do this as follows. We generate games consisting of exactly 2^N vertices using a vector $\boldsymbol{x}$ of Boolean variables of length N. The sets V_0 and V_1, the priority function p and the edges E are then generated by constructing a 'random' clause. For V_0 we generate a conjunctive clause by uniformly choosing, for each variable $\boldsymbol{x}_i$ whether the variable occurs positively, negatively, or not at all. The set V_1 is set to be the complement of V_0. For the priority mapping we use a similar approach ensuring that the priority mapping partitions the set of vertices V; we set an upper bound of K priorities, meaning that we generate K clauses. Since it may happen that some clauses become unsatisfiable, there may be fewer than K priorities in the games we generate. Finally, we construct the edge relation E. To this end, we first randomly select a subset $\boldsymbol{u}$ of the Boolean variables $\boldsymbol{x}$. Next, we generate M clauses where we again uniformly generate a clause based on a positive, negative or no occurrence of a variable in $\boldsymbol{u}$, and, similarly for their copies $\boldsymbol{u}'$. For the remaining variables we require that the primed copies $\boldsymbol{x}_i'$ are equivalent to the unprimed ones, *i.e.* $\boldsymbol{x}_i' \leftrightarrow \boldsymbol{x}_i$.

We measured the performance of both algorithms on a collection of randomly generated parity games where we varied N and K; we set M to 4. For each N, for $3 \leq N \leq 34$, we measured the total time to solve 2000 games. Figure 2 plots the size of the randomly generated parity games (each consisting of 2^N vertices) on the x-axis against the time (in seconds) on the y-axis per algorithm. The jumps in our plot are due to an increase in the number of priorities: for $N < 32$, we generated games with at most 7 priorities, whereas for $N \geq 32$ we generated games with at most 8 priorities.

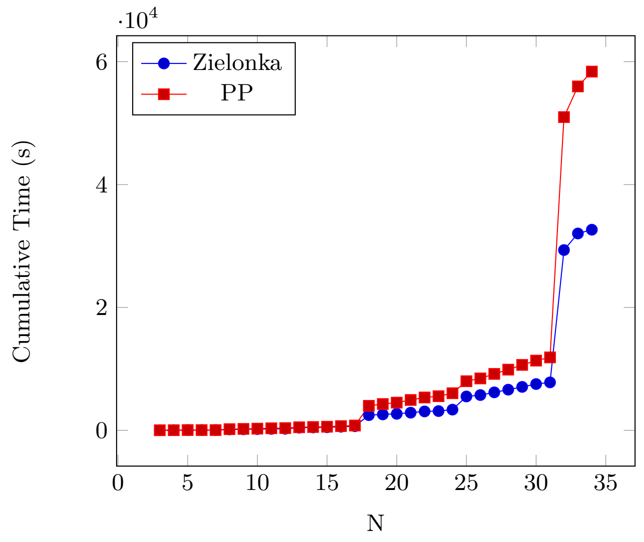

Fig. 2. Cumulative time (in seconds) required to solve 5000 random games (per instance of N, consisting of 2^N vertices each) using Zielonka and the PP algorithm.

We notice that Zielonka's algorithm consistently outperforms PP. While for small games the difference is not very significant, Zielonka seems to scale better as games get more complex, as witnessed for the larger values for N. We remark that the performance of both algorithms is still rather good: solving games consisting of 2^{34} vertices with 8 priorities takes, on average 5.7 seconds using Zielonka and 9.9 seconds using PP. We repeat the experiment, fixing N to 32, M again to 4 and varying K from 4 to 14, and generate 50 games for each configuration. The results are depicted in Figure 3.

These second set of experiments sketch a slightly different picture, showing that both Zielonka and PP have similar performance on average. Moreover, as the number of different priorities in the games increase, the performance of both algorithms degrades. This suggests that these algorithms will have more difficulty handling symbolic games that derive from, *e.g.* synthesis problems which typically have a large number of priorities.

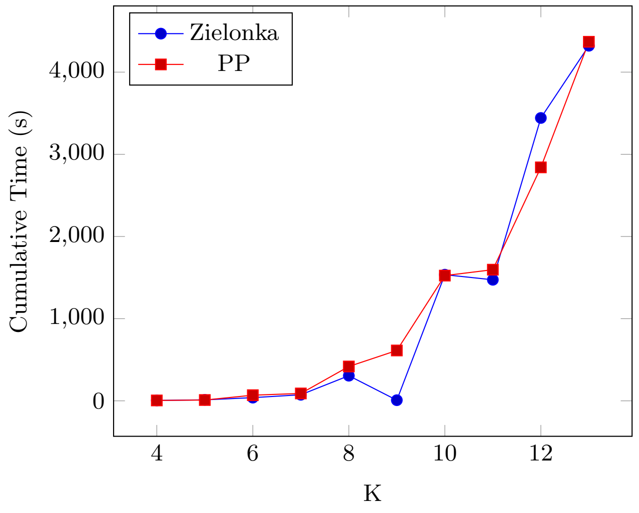

Fig. 3. Cumulative time (in seconds) required to solve 50 random games (per instance of K) with 2^{32} vertices using Zielonka and the PP algorithm.

5.2 The Keiren Parity Game Solving Benchmark Set

In [16], Keiren describes and provides a set of parity games that originate from over 300 model checking problems and more than 200 equivalence checking problems. The main obstacle in reusing the data set is that the games are encoded as explicit graphs in the PGSolver format. This means that most of the structure that a BDD solver can typically exploit for compactly representing the game graph is lost. As we concluded from the random games generated by PGSolver, a binary encoding of the sets involved leads to severe performance degradation of the solvers, and given the size of the graphs, there is little hope the running times of our algorithms on such encoded BDDs provide meaningful information. We have coped with this by generating BDDs from the original specifications for a few of the model checking games included in the Keiren benchmark set. Note that the conversion of the original specifications to BDDs is not straightforward: most games stem from model checking problems for system models that employ (unbounded) lists, natural numbers, *etcetera*, for which no trivial bounds can be established. The results can be found in Table 1.

We note that solving the explicit game for Onebit$_8$ (the onebit sliding window protocol with 8 different messages) requires over 140 seconds using the `pbespgsolve` tool of the mCRL2 [8] tool set and consumes 16Gb of memory. In contrast, our BDD algorithms both solve the resulting game in under a second and require a neglible amount of memory. We were unable to solve the Onebit$_{32}$ game using the `pbespgsolve` tool. Our experiments suggest that PP and Zielonka will perform comparably on games originating from verification problems.

Specification	Property	Zielonka	PP
Chatbox	It is possible to JOIN infinitely often	0.01	0.01
	Invariantly JOIN can be followed by LEAVE	0.01	0.01
Onebit$_2$	Infinitely often send and receive	0.06	0.09
	Absence of Deadlock	0.01	0.01
Onebit$_8$	Infinitely often send and receive	0.36	0.48
	Absence of Deadlock	0.10	0.11
Onebit$_{32}$	Infinitely often send and receive	0.80	1.13
	Absence of Deadlock	0.32	0.31

Table 1. Running times for selected model checking problems.

5.3 Discussion and Threats to Validity

The correctness of the tested algorithms was established in the respective papers in which they were published; the correctness of our implementation, however, cannot be guaranteed. Our code, however, closely matches the pseudocode given in the preceding sections. We cross-checked the answers of both solvers, comparing the computed winning sets for both players for each game we solved. Alternatively, one might compute winning strategies during game solving; such strategies provide the certificates that can be checked in polynomial time. While computing winning strategies can be done straightforwardly in an explicit setting, it is not clear how to do the same in a symbolic setting. Finally, we base some of our conclusions on randomly generated games; such games may not have the structure of games that originate from practical verification problems or synthesis problems.

Our results indicate that, overall, the performance of Zielonka's algorithm and the PP algorithm is comparable in most cases. For games with a small number of priorities, Zielonka seems to perform slightly better than PP, suggesting that in practice, the former is the solver of first choice. However, given that both solvers require at most seconds for even the largest models, containing up to 2^{34} vertices, either algorithm would work in practice. Comparing the performance of our symbolic solvers to the explicit solvers, however, indicates that there the BDD-based solvers offer huge performance gains.

6 Conclusions

We studied two algorithms for solving parity games, *viz.* Zielonka's recursive algorithm and the recently introduced Priority Promotion algorithm. Both algorithms work by repeatedly identifying dominions by decomposing a parity game, but both have a different way of doing so. We have described how these algorithms can be implemented using Binary Decision Diagrams (BDDs) and subsequently assessed the performance of our implementations. Since there are no available benchmark sets for experimenting on symbolically encoded BDDs, we have selected a few cases from the Keiren benchmark set for parity game solvers [16] and converted these manually to BDDs. In addition, we have assessed our implementations on random games of varying sizes and complexities. Our results show that the two algorithms perform remarkably similar on average, with the classic Zielonka algorithm slightly ourperforming the PP solver for games with few priorities. As soon as the number of priorities increases beyond the number typically found in games encoding verification problems, the performance of both algorithms is seriously compromised; this, however, is not specific to the use of BDDs.

References

1. A. Arnold, A. Vincent, and I. Walukiewicz. Games for synthesis of controllers with partial observation. *TCS*, 303(1):7–34, 2003.
2. M. Bakera, S. Edelkamp, P. Kissmann, and C.D. Renner. Solving μ-calculus parity games by symbolic planning. In *MoChArt*, volume 5348 of *Lecture Notes in Computer Science*, pages 15–33. Springer, 2008.
3. M. Benerecetti, D. Dell'Erba, and F. Mogavero. A delayed promotion policy for parity games. In *GandALF*, volume 226 of *EPTCS*, pages 30–45, 2016.
4. M. Benerecetti, D. Dell'Erba, and F. Mogavero. Improving priority promotion for parity games. In *Haifa Verification Conference*, volume 10028 of *Lecture Notes in Computer Science*, pages 117–133, 2016.
5. M. Benerecetti, D. Dell'Erba, and F. Mogavero. Solving parity games via priority promotion. In *CAV (2)*, volume 9780 of *Lecture Notes in Computer Science*, pages 270–290. Springer, 2016.
6. C.S. Calude, S. Jain, B. Khoussainov, W. Li, and F. Stephan. Deciding parity games in quasipolynomial time. In *STOC*, pages 252–263. ACM, 2017.
7. T. Chen, B. Ploeger, J. van de Pol, and T.A.C. Willemse. Equivalence Checking for Infinite Systems Using Parameterized Boolean Equation Systems. In *CONCUR'07*, pages 120–135, 2007.
8. S. Cranen, J.F. Groote, J.J.A. Keiren, F.P.M. Stappers, E.P. de Vink, W. Wesselink, and T.A.C. Willemse. An overview of the mcrl2 toolset and its recent advances. In *TACAS*, volume 7795 of *Lecture Notes in Computer Science*, pages 199–213. Springer, 2013.
9. R. Drechsler and B. Becker. *Binary Decision Diagrams - Theory and Implementation.* Springer, 1998.
10. E.A. Emerson and C.S. Jutla. Tree automata, mu-calculus and determinacy. In *FOCS'91*, pages 368–377, Washington, DC, USA, 1991. IEEE Computer Society.
11. O. Friedmann and M. Lange. Solving parity games in practice. In *ATVA*, volume 5799 of *Lecture Notes in Computer Science*, pages 182–196. Springer, 2009.
12. M. Gazda and T.A.C. Willemse. Zielonka's recursive algorithm: dull, weak and solitaire games and tighter bounds. In *GandALF*, volume 119 of *EPTCS*, pages 7–20, 2013.
13. M. Jurdziński. Deciding the winner in parity games is in UP $\cap$ co-UP. *IPL*, 68(3):119–124, 1998.
14. M. Jurdziński. Small progress measures for solving parity games. In *STACS'00*, volume 1770 of *LNCS*, pages 290–301. Springer, 2000.
15. M. Jurdzinski and R. Lazic. Succinct progress measures for solving parity games. In *LICS*, pages 1–9. IEEE Computer Society, 2017.

16. J.J.A. Keiren. Benchmarks for parity games. In *FSEN*, volume 9392 of *Lecture Notes in Computer Science*, pages 127–142. Springer, 2015.
17. Y. Liu, Z. Duan, and C. Tian. An improved recursive algorithm for parity games. In *TASE*, pages 154–161. IEEE Computer Society, 2014.
18. R. McNaughton. Infinite games played on finite graphs. *APAL*, 65(2):149–184, 1993.
19. T. van Dijk. Oink: an implementation and evaluation of modern parity game solvers. In *TACAS*, 2018. To appear; see https://arxiv.org/abs/1801.03859.
20. I. Wegener. *Branching Programs and Binary Decision Diagrams*. SIAM, 2000.
21. W. Zielonka. Infinite games on finitely coloured graphs with applications to automata on infinite trees. *TCS*, 200(1-2):135 – 183, 1998.