Fatal Attractors in Parity Games

Michael Huth[1], Jim Huan-Pu Kuo[1], and Nir Piterman[2]

[1] Department of Computing, Imperial College London
London, SW7 2AZ, United Kingdom
{m.huth, jimhkuo}@imperial.ac.uk
[2] Department of Computer Science, University of Leicester
Leicester, LE1 7RH, United Kingdom
nir.piterman@leicester.ac.uk

Abstract. We study a new form of attractor in parity games and use it to define solvers that run in PTIME and are *partial* in that they do not solve all games completely. Technically, for color c this new attractor determines whether player $c\%2$ can reach a set of nodes X of color c whilst avoiding any nodes of color less than c. Such an attractor is *fatal* if player $c\%2$ can attract all nodes in X back to X in this manner. Our partial solvers detect fixed-points of nodes based on fatal attractors and correctly classify such nodes as won by player $c\%2$. Experimental results show that our partial solvers completely solve benchmarks that were constructed to challenge existing full solvers. Our partial solvers also have encouraging run times in practice. For one partial solver we prove that its runtime is in $O(|V|^3)$, that its output game is independent of the order in which attractors are computed, and that it solves all Büchi games. [3]

1 Introduction

Parity games are an important foundational structure in formal verification (see e.g. [11]). Mathematically, they can be seen as a representation of the model checking problem for the modal mu-calculus [4], and its exact computational complexity has been an open problem for over twenty years now.

Parity games are infinite, 2-person, 0-sum, graph-based games that are hard to solve. Their nodes, controlled by different players, are colored with natural numbers and the winning condition of plays depends on the minimal color occurring in cycles. The condition for winning a node, therefore, is an alternation of existential and universal quantification. In practice, this means that the maximal color of its coloring function is the only exponential source for the worst-case complexity of most parity game solvers, e.g. for those in [11, 8, 10].

Research on solving parity games may be loosely grouped into the following approaches: design of algorithms that solve all parity games by construction and that so far all have exponential or sub-exponential worst-case complexity

[3] A preliminary version of the results reported in this paper was presented at the GAMES 2012 workshop in Naples, Italy, on 11 September 2012.

(e.g. [11, 8, 10, 9]), restriction of parity games to classes for which polynomial-time algorithms can be devised as complete solvers (e.g. [1, 3]), and practical improvements to solvers so that they perform well across benchmarks (e.g. [5]).

We here propose a new approach that relates to, and potentially impacts, all of these aforementioned activities. We want to design and evaluate a new form of "partial" parity game solver. These are solvers that are well defined for all parity games but that may not solve all games completely, i.e. for some parity games they may not decide the winning status of some nodes. For us, a partial solver has an arbitrary parity game as input and returns two things: a sub-game of the input game, and a classification of the winning status of all nodes of the input game that are not in that sub-game. In particular, the returned sub-game is empty if, and only if, the partial solver classified the winners for all input nodes.

The input/output type of our partial solvers clearly relates them to so called preprocessors that may decide the winner of nodes whose structure makes such a decision an easy static criterion (e.g. in the elimination of self-loops or dead ends [5]). But we here search for dynamic criteria that allow partial solvers to completely solve a range of benchmarks of parity games. This ambition sets our work apart from research on preprocessors but is consistent with it as one can always run a partial solver as preprocessor.

The motivation for the study reported in this paper is that we want to investigate what theoretical building blocks one may create and use for designing partial solvers that run in polynomial time and work well on many games, whether partial solvers can be components of more efficient complete solvers, and whether there are interesting subclasses of parity games for which partial solvers completely solve all games. In particular, one may study the class of output games of a PTIME partial solver in lieu of studying the aforementioned open problem for all parity games.

We summarize the main contributions made in this paper:

- We present a new form of attractor that can be used in fixed-point computations to detect winning nodes for a given player in parity games.
- We propose several designs of partial solvers for parity games by using this new attractor within fixed-point computations.
- We analyze these partial solvers and show, e.g., that they work in PTIME and that one of them is independent of the order of attractor computation.
- And we evaluate these partial solvers against known benchmarks and report that these experiments have very encouraging results.

Outline of paper. Section 2 contains needed formal background and fixes notation. Section 3 introduces the building block of our partial solvers, a new form of attractor. Some partial solvers based on this attractor are presented in Section 4, theoretical results about these partial solvers are proved in Section 5, and experimental results for these partial solvers run on benchmarks are reported and discussed in Section 6. We summarize and conclude the paper in Section 7.

2 Preliminaries

We write $\mathbb{N}$ for the set $\{0, 1, \dots\}$ of natural numbers. A parity game G is a tuple (V, V_0, V_1, E, c), where V is a set of nodes partitioned into possibly empty node sets V_0 and V_1, with an edge relation $E \subseteq V \times V$ (where for all v in V there is a w in V with (v, w) in E), and a coloring function $c\colon V \to \mathbb{N}$. In figures, $c(v)$ is written within nodes v, nodes in V_0 are depicted as circles and nodes in V_1 as squares. For v in V, we write $v.E$ for node set $\{w \in V \mid (v, w) \in E\}$ of successors of v. By abuse of language, we call a subset U of V a *sub-game* of G if the game graph $(U, E \cap (U \times U))$ is such that all nodes in U have some successor. We write $\mathcal{P}G$ for the class of all finite parity games G, which includes the parity game with empty node set for our convenience. We only consider games in $\mathcal{P}G$.

Throughout, we write p for one of 0 or 1 and $1 - p$ for the other player. In a parity game, player p owns the nodes in V_p. A play from some node v_0 results in an infinite play $r = v_0 v_1 \dots$ in (V, E) where the player who owns v_i chooses the successor v_{i+1} such that (v_i, v_{i+1}) is in E. Let $\mathsf{Inf}(r)$ be the set of colors that occur in r infinitely often: $\mathsf{Inf}(r) = \{k \in \mathbb{N} \mid \forall j \in \mathbb{N}\colon \exists i \in \mathbb{N}\colon i > j \text{ and } k = c(v_i)\}$. Player 0 wins play r iff $\min \mathsf{Inf}(P)$ is even; otherwise player 1 wins play r.

A strategy for player p is a total function $\tau\colon V_p \to V$ such that $(v, \tau(v))$ is in E for all $v \in V_p$. A play r is consistent with τ if each node v_i in r owned by player p satisfies $v_{i+1} = \tau(v_i)$. It is well known that each parity game is determined: node set V is the disjoint union of two, possibly empty, sets W_0 and W_1, the winning regions of players 0 and 1 (respectively). Moreover, strategies $\sigma\colon V_0 \to V$ and $\pi\colon V_1 \to V$ can be computed such that

- all plays beginning in W_0 and consistent with σ are won by player 0; and
- all plays beginning in W_1 and consistent with π are won by player 1.

Solving a parity game means computing such data (W_0, W_1, σ, π).

Example 1. In the parity game G depicted in Figure 1, the winning regions are $W_1 = \{v_3, v_5, v_7\}$ and $W_0 = \{v_0, v_1, v_2, v_4, v_6, v_8, v_9, v_{10}, v_{11}\}$. Let σ move from v_2 to v_4, from v_6 to v_8, from v_9 to v_8, and from v_{10} to v_9. Then σ is a winning strategy for player 0 on W_0. And every strategy π is winning for player 1 on W_1.

3 Fatal attractors

In this section we define a special type of attractor that is used for our partial solvers in the next section. We start by recalling the normal definition of attractor, and that of a trap, and then generalize the former to our purposes.

Definition 1. *Let X be a node set in parity game G. For player p in $\{0, 1\}$, set*

$$\mathsf{cpre}_p(X) = \{v \in V_p \mid v.E \cap X \neq \emptyset\} \cup \{v \in V_{1-p} \mid v.E \subseteq X\} \tag{1}$$

$$\mathsf{Attr}_p[G, X] = \mu Z.(X \cup \mathsf{cpre}_p(Z)) \tag{2}$$

where $\mu Z.F(Z)$ denotes the least fixed point of a monotone function $F\colon 2^V \to 2^V$.

The control predecessor of a node set X for p in (1) is the set of nodes from which player p can force to get to X in exactly one move. The attractor for

player p to a set X in (2) is computed via a least fixed-point as the set of nodes from which player p can force the game in zero or more moves to get to the set X. Dually, a *trap* for player p is a region from which player p cannot escape.

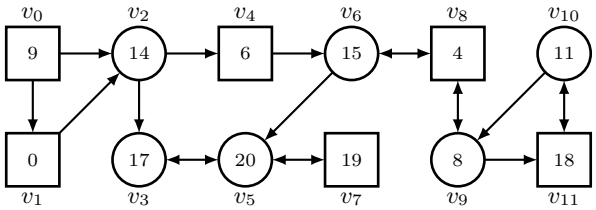

Fig. 1. A parity game: circles denote nodes in V_0, squares denote nodes in V_1.

Definition 2. *Node set X in parity game G is a trap for player p (p-trap) if for all $v \in V_p \cap X$ we have $v.E \subseteq X$ and for all $v \in V_{1-p} \cap X$ we have $v.E \cap X \neq \emptyset$.*

It is well known that the complement of an attractor for player p is a p-trap and that it is a sub-game. We state this here formally as a reference:

Theorem 1. *Given a node set X in a parity game G, the set $V \setminus \mathsf{Attr}_p[G, X]$ is a p-trap and a sub-game of G.*

We now define a new type of attractor, which will be a crucial ingredient in the definition of all our partial solvers.

Definition 3. *Let A and X be node sets in parity game G, let p in $\{0, 1\}$ be a player, and c a color in G. We set*

$$\begin{aligned} \mathsf{mpre}_p(A, X, c) &= \{v \in V_p \mid c(v) \geq c \wedge v.E \cap (A \cup X) \neq \emptyset\} \cup \\ &\quad \{v \in V_{1-p} \mid c(v) \geq c \wedge v.E \subseteq A \cup X\} \\ \mathsf{MAttr}_p(X, c) &= \mu Z.\mathsf{mpre}_p(Z, X, c) \end{aligned} \tag{3}$$

The *monotone* control predecessor $\mathsf{mpre}_p(A, X, c)$ of node set A for p with target X is the set of nodes of color at least c from which player p can force to get to either A or X in one move. The *monotone* attractor $\mathsf{MAttr}_p(X, c)$ for p with target X is the set of nodes from which player p can force the game in one or more moves to X by only meeting nodes whose color is at least c. Notice that the target set X is kept external to the attractor. Thus, if some node x in X is included in $\mathsf{MAttr}_p(X, c)$ it is so as it is attracted to X in at least one step.

Our control predecessor and attractor are different from the "normal" ones in a few ways. First, ours take into account the color c as a formal parameter. They add only nodes that have color at least c. Second, as discussed above, the target set X itself is not included in the computation by default. For example, $\mathsf{MAttr}_p(X, c)$ includes states from X only if they can be attracted to X.

We now show the main usage of this new operator by studying how specific instantiations thereof can compute so called *fatal attractors*.

Definition 4. *Let X be a set of nodes of color c, where $p = c\%2$.*

1. *For such an X we denote p by $p(X)$ and c by $c(X)$. We denote $\mathsf{MAttr}_p(X, c)$ by $\mathsf{MA}(X)$. If $X = \{x\}$ is a singleton, we denote $\mathsf{MA}(X)$ by $\mathsf{MA}(x)$.*
2. *We say that $\mathsf{MA}(X)$ is* a fatal attractor *if $X \subseteq \mathsf{MA}(X)$.*

We note that fatal attractors $\mathsf{MA}(X)$ are node sets that are won by player $p(X)$ in G. The winning strategy is the attractor strategy corresponding to the least fixed-point computation in $\mathsf{MAttr}_p(X, c)$. First of all, player $p(X)$ can force, from all nodes in $\mathsf{MA}(X)$, to reach some node in X in at least one move. Then, player $p(X)$ can do this again from this node in X as X is a subset of $\mathsf{MA}(X)$. At the same time, by definition of $\mathsf{MAttr}_p(X, c)$ and $\mathsf{mpre}_p(A, X, c)$, the attraction ensures that only colors of value at least c are encountered. So in plays starting in $\mathsf{MA}(X)$ and consistent with that strategy, every visit to a node of parity $1-p(X)$ is followed later by a visit to a node of color $c(X)$. It follows that in an infinite play consistent with this strategy and starting in $\mathsf{MA}(X)$, the minimal color to be visited infinitely often is c – which is of p's parity.

Theorem 2. *Let $\mathsf{MA}(X)$ be fatal in parity game G. Then the attractor strategy for player $p(X)$ on $\mathsf{MA}(X)$ is winning for $p(X)$ on $\mathsf{MA}(X)$ in G.*

Let us consider the case when X is a singleton $\{k\}$ and $\mathsf{MA}(k)$ is not fatal. Suppose that there is an edge (k, w) in E with w in $\mathsf{MA}(k)$. We show that this edge cannot be part of a winning strategy (of either player) in G. Since $\mathsf{MA}(k)$ is not fatal, k must be in $V_{1-p(k)}$ and so is controlled by player $1 - p(k)$. But if that player were to move from k to w in a memoryless strategy, player $p(k)$ could then attract the play from w back to k without visiting colors of parity $1-p(k)$ and smaller than $c(k)$, since w is in $\mathsf{MA}(k)$. And, by the existence of memoryless winning strategies [4], this would ensure that the play is won by player $p(k)$ as the minimal infinitely occurring color would have parity $p(k)$. We summarize:

Lemma 1. *Let $\mathsf{MA}(k)$ be not fatal for node k. Then we may remove edge (k, w) in E if w is in $\mathsf{MA}(k)$, without changing winning regions of parity game G.*

Example 2. For G in Figure 1, the only colors k for which $\mathsf{MA}(k)$ is fatal are 4 and 8: $\mathsf{MA}(4)$ equals $\{v_2, v_4, v_6, v_8, v_9, v_{10}, v_{11}\}$ and $\mathsf{MA}(8)$ equals $\{v_9, v_{10}, v_{11}\}$. In particular, $\mathsf{MA}(8)$ is contained in $\mathsf{MA}(4)$ and nodes v_1 and v_0 are attracted to $\mathsf{MA}(4)$ in G by player 0. And v_{11} is in $\mathsf{MA}(11)$ (but the node of color 11, v_{10}, is not), so edge (v_{10}, v_{11}) may be removed.

4 Partial solvers

We can use the above definitions and results to define partial solvers next. Their soundness will be shown in Section 5.

```
psol(G = (V, V_0, V_1, E, c)) {
  for (k ∈ V in descending color ordering c(k)) {
    if (k ∈ MA(k)) { return psol(G \ Attr_{p(k)}[G, MA(k)]) }
    if (∃(k, w) ∈ E: w ∈ MA(k))
    { G = G \ {(k, w) ∈ E | w ∈ MA(k)} }
  }
  return G
}
```

Fig. 2. Partial solver `psol` based on detection of fatal attractors $\mathsf{MA}(k)$ and fatal moves.

4.1 Partial solver `psol`

Figure 2 shows the pseudocode of a partial solver, named `psol`, based on $\mathsf{MA}(X)$ for singleton sets X. Solver `psol` explores the parity game G in descending color ordering. For each node k, it constructs $\mathsf{MA}(k)$, and aims to do one of two things:

- If node k is in $\mathsf{MA}(k)$, then $\mathsf{MA}(k)$ is fatal for player $1 - p(k)$, thus node set $\mathsf{Attr}_{p(k)}[G, \mathsf{MA}(k)]$ is a winning region of player $p(k)$, and removed from G.
- If node k is not in $\mathsf{MA}(k)$, and there is a (k, w) in E where w is in $\mathsf{MA}(k)$, all such edges (k, w) are removed from E and the iteration continues.

If for no k in V attractor $\mathsf{MA}(k)$ is fatal, game G is returned as is – empty if `psol` solves G completely. The accumulation of winning regions and computation of winning strategies are omitted from the pseudocode for improved readability.

Example 3. In a run of `psol` on G from Figure 1, there is no effect for colors larger than 11. For $c = 11$, `psol` removes edge (v_{10}, v_{11}) as v_{11} is in $\mathsf{MA}(11)$. The next effect is for $c = 8$, when the fatal attractor $\mathsf{MA}(8) = \{v_9, v_{10}, v_{11}\}$ is detected and removed from G (the previous edge removal did not cause the attractor to be fatal). On the remaining game, the next effect occurs when $c = 4$, and when the fatal attractor $\mathsf{MA}(4)$ is $\{v_2, v_4, v_6, v_8\}$ in that remaining game. As player 0 can attract v_0 and v_1 to this as well, all these nodes are removed and the remaining game has node set $\{v_3, v_5, v_7\}$. As there is no more effect of `psol` on that remaining game, it is returned as the output of `psol`'s run.

4.2 Partial solver `psolB`

Figure 3 shows the pseudocode of another partial solver, named `psolB` (the "B" suggests a relationship to "Büchi"), based on $\mathsf{MA}(X)$, where X is a set of nodes of the same color. This time, the operator $\mathsf{MA}(X)$ is used within a greatest fixed-point in order to discover the largest set of nodes of a certain color that can be (fatally) attracted to itself. Accordingly, the greatest fixed-point starts from all the nodes of a certain color and gradually removes those that cannot be attracted to the same color. When the fixed-point stabilizes, it includes the set of nodes of the given color that can be (fatally) attracted to itself. This node set can be removed (as a winning region for player $d\%2$) and the residual game analyzed recursively. As before, the colors are explored in descending order.

```
psolB(G = (V, V_0, V_1, E, c)) {
  for (colors d in descending ordering) {
    X = { v in V | c(v) = d };
    cache = {};
    while (X ≠ {} && X ≠ cache) {
      cache = X;
      if (X ⊆ MA(X)) { return psolB(G \ Attr_{d%2}[G, MA(X)])
      } else { X = X ∩ MA(X); }
    }
  }
  return G
}
```

Fig. 3. Partial solver `psolB`.

We make two observations. First, if we were to replace the recursive calls in `psolB` with the removal of the winning region from G and a continuation of the iteration, we would get an implementation that discovers less fatal attractors. Second, edge removal in `psol` relies on the set X being a singleton. A similar removal could be achieved in `psolB` when the size of X is reduced by one (in the operation $X = X \cap \mathsf{MA}(X)$). Indeed, in such a case the removed node would not be removed and the current value of X be realized as fatal. We have not tested this edge removal approach experimentally for this variant of `psolB`.

Example 4. A run of `psolB` on G from Figure 1 has the same effect as the one for `psol`, except that `psolB` does not remove edge (v_{10}, v_{11}) when $c = 11$.

A way of comparing partial solvers P_1 and P_2 is to say that $P_1 \leq P_2$ if, and only if, for all parity games G the set of nodes in the output sub-game $P_1(G)$ is a subset of the set of nodes of the output sub-game $P_2(G)$. We note that `psol` and `psolB` are incomparable for this intensional pre-order over partial solvers.

4.3 Partial solver `psolQ`

It seems that `psolB` is more general than `psol` in that if there is a singleton X with $X \subseteq \mathsf{MA}(X)$ then `psolB` will discover this as well. However, the requirement to attract to a single node seems too strong. Solver `psolB` removes this restriction and allows to attract to more than one node, albeit of the same color. Now we design a partial solver `psolQ` that can attract to a set of nodes of more than one color (the "Q" is our code name for this "Q"uantified layer of colors of the same parity). Solver `psolQ` allows to combine attraction to multiple colors by adding them gradually and taking care to "fix" visits to nodes of opposite parity.

We extend the definition of mpre and MAttr to allow inclusion of more (safe) nodes when collecting nodes in the attractor.

```
layeredAttr(G,p,X) { // PRE-CONDITION: all nodes in X have parity p
   A = {};
   b = max{c(v) | v ∈ X};
   for (d = p up to b in increments of 2) {
      Y = {v ∈ X | c(v) ≤ d};
      A = PMAttr_p(A ∪ Y, d);
   }
   return A;
}

psolQ(G = (V, V_0, V_1, E, c)) {
   for (colors b in ascending order) {
      X = {v ∈ V | c(v) ≤ b ∧ c(v)%2 = b%2};
      cache = {};
      while (X ≠ {} && X ≠ cache) {
         cache = X;
         W = layeredAttr(G,b%2,X);
         if (X ⊆ W) { return psolQ(G \ Attr_{b%2}[G, W]);
         } else { X = X ∩ W; }
      }
   }
   return G;
}
```

Fig. 4. Operator `layeredAttr`(G, p, X) and partial solver `psolQ`.

Definition 5. *Let A and X be node sets in parity game G, let p in $\{0, 1\}$ be a player, and c a color in G. We set*

$$\mathsf{pmpre}_p(A, X, c) = \{v \in V_p \mid (c(v) \geq c \vee v \in X) \wedge v.E \cap (A \cup X) \neq \emptyset\} \cup \{v \in V_{1-p} \mid (c(v) \geq c \vee v \in X) \wedge v.E \subseteq A \cup X\} \quad (4)$$

$$\mathsf{PMAttr}_p(X, c) = \mu Z.\mathsf{pmpre}_p(Z, X, c) \quad (5)$$

The *permissive monotone* predecessor in (4) adds to the monotone predecessor also nodes that are in X itself even if their color is lower than c, i.e., they violate the monotonicity requirement. The *permissive monotone* attractor in (5) then uses the permissive predecessor instead of the simpler predecessor. This is used for two purposes. First, when the set X includes nodes of multiple colors – some of them lower than c. Then, inclusion of nodes from X does not destroy the properties of fatal attraction. Second, increasing the set X of target nodes allows to include the previous target as set of "permissible" nodes. This creates a layered structure of attractors.

We use the permissive attractor to define `psolQ`. Figure 4 presents the pseudo code of operator `layeredAttr`(G, p, X). It is an attractor that combines attraction to nodes of multiple color. It takes a set X of colors of the same parity p. It considers increasing subsets of X with more and more colors and tries to attract fatally to them. It starts from a set Y_p of nodes of parity p with color p and computes $\mathsf{MA}(Y_p)$. At this stage, the difference between pmpre and mpre does not apply as Y_p contains nodes of only one color and A is empty. Then, instead of

stopping as before, it continues to accumulate more nodes. It creates the set Y_{p+2} of the nodes of parity p with color p or $p+2$. Then, $\mathsf{PMAttr}_p(A \cup Y_{p+2}, p+2)$ includes all the previous nodes in A (as all nodes in A are now permissible) and all nodes that can be attracted to them or to Y_{p+2} through nodes of color at least $p+2$. This way, even if nodes of a color lower than $p+2$ are included they will be ensured to be either in the previous attractor or of the right parity. Then Y is increased again to include some more nodes of p's parity. This process continues until it includes all nodes in X.

This layered attractor may also be fatal:

Definition 6. *We say that* `layeredAttr`(G, p, X) *is fatal if* X *is a subset of* `layeredAttr`(G, p, X).

As before, fatal layered attractors are won by player p in G. The winning strategy is more complicated as it has to take into account the number of iterations in the for loop in which a node was first discovered. Every node in `layeredAttr`(G, p, X) belongs to a layer corresponding to a maximal color d. From a node in layer d, player p can force to reach some node in $Y_d \subseteq X$ or some node in a lower layer d'. As the number of layers is finite, eventually some node in X is reached. When reaching X, player p can attract to X in the same layered fashion again as X is a subset of `layeredAttr`(G, p, X). Along the way, while attracting through layer d we are ensured that only colors at least d or of a lower layer are encountered. So in plays starting in `layeredAttr`(G, p, X) and consistent with that strategy, every visit to a node of parity $1-p$ is followed later by a visit to a node of parity p of lower color.

Theorem 3. *Let* `layeredAttr`(G, p, X) *be fatal in parity game* G*. Then the layered attractor strategy for player* p *on* `layeredAttr`(G, p, X) *is winning for* p *on* `layeredAttr`(G, p, X) *in* G.

Pseudo code of solver `psolQ` is also shown in Figure 4: `psolQ` prepares increasing sets of nodes X of the same color and calls `layeredAttr` within a greatest fixed-point. For a set X, the greatest fixed-point attempts to discover the largest set of nodes within X that can be fatally attracted to itself (in a layered fashion). Accordingly, the greatest fixed-point starts from all the nodes in X and gradually removes those that cannot be attracted to X. When the fixed-point stabilizes, it includes a set of nodes of the same parity that can be attracted to itself. These are removed (along with the normal attractor to them) and the residual game is analyzed recursively.

We note that the first two iterations of `psolQ` are equivalent to calling `psolB` on colors 0 and 1. Then, every iteration of `psolQ` extends the number of colors considered. In particular, in the last two iterations of `psolQ` the value of b is the maximal possible value of the appropriate parity. It follows that the sets X defined in these last two iterations include all nodes of the given parity. These last two computations of greatest fixed-points are the most general and subsume all previous greatest fixed-point computations. We discuss in Section 6 why we increase the bound b gradually and do not consider these two iterations alone.

Example 5. The run of `psolQ` on G from Figure 1 finds a fatal attractor for bound $b = 4$, which removes all nodes except v_3, v_5, and v_7. For $b = 19$, it realizes that these nodes are won by player 1, and outputs the empty game. That `psolQ` is a partial solver can be seen in Figure 5, which depicts a game that is not modified at all by `psolQ` and so is returned as is.

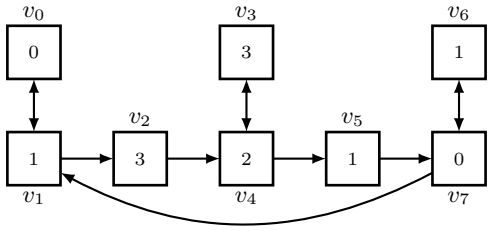

Fig. 5. A 1-player parity game modified by neither `psol`, `psolB` nor `psolQ`.

5 Properties of our partial solvers

We now discuss the properties of our partial solvers, looking first at their soundness and computational complexity.

5.1 Soundness and Computational Complexity

Theorem 4. *1. The partial solvers* `psol`*,* `psolB`*, and* `psolQ` *are sound.*
2. The running time for `psol` *and* `psolB` *is in* $O(|V|^2 \cdot |E|)$.
3. And `psol` *and* `psolB` *can be implemented to run in time* $O(|V|^3)$.
4. And `psolQ` *runs in time* $O(|V|^2 \cdot |E| \cdot |c|)$ *with* $|c|$ *the number of colors in* G.

If `psolQ` were to restrict attention to the last two iterations of the for loop, i.e., those that compute the greatest fixed-point with the maximal even color and the maximal odd color, the run time of `psolQ` would be bounded by $O(|V|^2 \cdot |E|)$. For such a version of `psolQ` we also ran experiments on our benchmarks and do not report these results, except to say that this version performs considerably worse than `psolQ` in practice. We believe that this is so since `psolQ` more quickly discovers small winning regions that "destabilize" the rest of the games.

5.2 Robustness of `psolB`

Our pseudo-code for `psolB` iterates through colors in descending order. A natural question is whether the computed output game depends on the order in which these colors are iterated. Below, we formally state that the outcome of `psolB` is indeed independent of the iteration order. This suggests that these solvers are a form of polynomial-time projection of parity games onto sub-games.

Let us formalize this. Let π be some sequence of colors in G, that may omit or repeat some colors from G. Let $\texttt{psolB}(\pi)$ be a version of `psolB` that checks for (and removes) fatal attractors according to the order in π (including any color repetitions in π). We say that $\texttt{psolB}(\pi)$ is *stable* if for every color c_1, the input/output behavior of $\texttt{psolB}(\pi)$ and $\texttt{psolB}(\pi \cdot c_1)$ are the same. That is, the sequence π leads `psolB` to stabilization in the sense that every extension of the version $\texttt{psolB}(\pi)$ with one color does not change the input/output behavior.

Theorem 5. *Let π_1 and π_2 be sequences of colors with $\texttt{psolB}(\pi_1)$ and $\texttt{psolB}(\pi_2)$ stable. Then G_1 equals G_2 if G_i is the output of $\texttt{psolB}(\pi_i)$ on G, for $1 \leq i \leq 2$.*

Next, we formally define classes of parity games, those that `psolB` solves completely and those that `psolB` does not modify.

Definition 7. *We define class $\mathcal{S}$ (for "Solved") to consist of those parity games G for which $\texttt{psolB}(G)$ outputs the empty game. And we define $\mathcal{K}$ (for "Kernel") as the class of those parity games G for which $\texttt{psolB}(G)$ outputs G again.*

The meaning of `psolB` is therefore a total, idempotent function of type $\mathcal{P}G \to \mathcal{K}$ that has $\mathcal{S}$ as inverse image of the empty parity game. By virtue of Theorem 5, classes $\mathcal{S}$ and $\mathcal{K}$ are *semantic* in nature.

We now show that $\mathcal{S}$ contains the class of Büchi games, which we identify with parity games G with color 0 and 1 and where nodes with color 0 are those that player 0 wants to reach infinitely often.

Theorem 6. *Let G be a parity game whose colors are only 0 and 1. Then G is in $\mathcal{S}$, i.e. `psolB` completely solves G.*

We point out that $\mathcal{S}$ does not contain some game types for which polynomial-time solvers are known. For example, not all 1-player parity games are in $\mathcal{S}$ (see Figure 5). Class $\mathcal{S}$ is also not closed under sub-games.

6 Experimental results

6.1 Experimental setup

We wrote Scala implementations of `psol`, `psolB`, and `psolQ`, and of Zielonka's solver (`zlka`) that rely on the same data structures and do not compute winning strategies – which has routine administrative overhead. The (parity) *Game* object has a map of *Node*s (objects) with node identifiers (integers) as the keys. Apart from colors and owner type (0 or 1), each *Node* has two lists of identifiers, one for successors and one for predecessors in the game graph (V, E). For attractor computation, the predecessor list is used to perform "backward" attraction.

This uniform use of data types allows for a first informed comparison. We chose `zlka` as a reference implementation since it seems to work well in practice on many games [5]. We then compared the performance of these implementations on all eight non-random, structured game types produced by the PGSolver tool [6]. Here is a list of brief descriptions of these game types.

– `Clique`: fully connected games with alternating colors and no self-loops.
– `Ladder`: layers of node pairs with connections between adjacent layers.
– `Recursive Ladder`: layers of 5-node blocks with loops.
– `Strategy Impr`: worst cases for strategy improvement solvers.
– `Model Checker Ladder`: layers of 4-node blocks.
– `Tower Of Hanoi`: captures well-known puzzle.
– `Elevator Verification`: a verification problem for an elevator model.
– `Jurdzinski`: worst cases for small progress measure solvers.

The first seven types take as game parameter a natural number n as input, whereas `Jurdzinski` takes a pair of such numbers n, m as game parameter.

For regression testing, we verified for all tested games that the winning regions of `psol`, `psolB`, `psolQ` and `zlka` are consistent with those computed by PGSolver. Runs of these algorithms that took longer than 20 minutes (i.e. 1200K milliseconds) or for which the machine exhausted the available memory during solver computation are recorded as aborts ("`abo`") – the most frequent reason for `abo` was that the used machine ran out of memory. All experiments were conducted on the same machine with an Intel® Core™ i5 (four cores) CPU at 3.20GHz and 8G of RAM, running on a Ubuntu 11.04 Linux operating system.

For most game types, we used *unbounded binary search* starting with 2 and then iteratively doubling that value, in order to determine the `abo` boundary value for parameter n within an accuracy of plus/minus 10. As the game type `Jurdzinski`$[n, m]$ has two parameters, we conducted three unbounded binary searches here: one where n is fixed at 10, another where m is fixed at 10, and a third one where n equals m. We used a larger parameter configuration ($10 \times$ power of two) for `Jurdzinski` games.

We report here only the last two powers of two for which one of the partial solvers didn't timeout, as well as the boundary values for each solver. For game types whose boundary value was less than 10 (`Tower Of Hanoi` and `Elevator Verification`), we didn't use binary search but incremented n by 1. Finally, if a partial solver didn't solve its input game completely, we ran `zlka` on the remaining game and added the observed running times for `zlka` to that of the partial solver. (This occurred for `Elevator Verification` for `psol` and `psolB`.)

6.2 Experiments on structured games

Our experimental results are depicted in Figures 6 and 7, colored green (respectively red) for the partial solver with best (respectively worst) result. Running times are reported in milliseconds. The most important outcome is that partial solvers `psol` and `psolB` solved seven of the eight game types *completely* for all runs that did not time out, the exception being `Elevator Verification`; and that `psolQ` solved all eight game types completely. This suggests that partial solvers can actually be used as solvers on a range of structured game types.

We now compare the performance of these partial solvers and of `zlka`. There were ten experiments, three for `Jurdzinski` and one for each of the remaining seven game types. For seven out of these ten experiments, `psolB` had the largest boundary value of the parameter and so seems to perform best overall. The solver

`Clique`$[n]$

n	psol	psolB	psolQ	zlka
2**11	6016.68	48691.72	3281.57	12862.92
2**12	abo	164126.06	28122.96	76427.44
20min	$n = 3680$	$n = 5232$	$n = 4608$	$n = 5104$

`Ladder`$[n]$

n	psol	psolB	psolQ	zlka
2**19	abo	22440.57	26759.85	24406.79
2**20	abo	47139.96	59238.77	75270.74
20min	$n = 14712$	$n = 1596624$	$n = 1415776$	$n = 1242376$

`Model Checker Ladder`$[n]$

n	psol	psolB	psolQ	zlka
2**12	119291.99	90366.80	117006.17	79284.72
2**13	560002.68	457049.22	644225.37	398592.74
20min	$n = 11528$	$n = 12288$	$n = 10928$	$n = 13248$

`Recursive Ladder`$[n]$

n	psol	psolB	psolQ	zlka
2**12	abo	abo	138956.08	abo
2**13	abo	abo	606868.31	abo
20min	$n = 1560$	$n = 2064$	$n = 11352$	$n = 32$

`Strategy Impr`$[n]$

n	psol	psolB	psolQ	zlka
2**10	174913.85	134795.46	abo	abo
2**11	909401.03	631963.68	abo	abo
20min	$n = 2368$	$n = 2672$	$n = 40$	$n = 24$

`Tower Of Hanoi`$[n]$

n	psol	psolB	psolQ	zlka
9	272095.32	54543.31	610264.18	56780.41
10	abo	397728.33	abo	390407.41
20min	$n = 9$	$n = 10$	$n = 9$	$n = 10$

`Elevator Verification`$[n]$

n	psol	psolB	psolQ	zlka
1	171.63	120.59	147.32	125.41
2	646.18	248.56	385.56	237.51
3	2707.09	584.83	806.28	512.72
4	223829.69	1389.10	2882.14	1116.85
5	abo	11681.02	22532.75	3671.04
6	abo	168217.65	373568.85	41344.03
7	abo	abo	abo	458938.13
20min	$n = 4$	$n = 6$	$n = 6$	$n = 7$

Fig. 6. First experimental results for partial solvers run over benchmarks

`Jurdzinski`$[10, m]$

m	psol	psolB	psolQ	zlka
10*2**7	abo	179097.35	abo	abo
10*2**8	abo	833509.48	abo	abo
20min	$n = 560$	$n = 2890$	$n = 1120$	$n = 480$

`Jurdzinski`$[n, 10]$

n	psol	psolB	psolQ	zlka
10*2**7	308033.94	106453.86	abo	abo
10*2**8	abo	406621.65	abo	abo
20min	$n = 2420$	$n = 4380$	$n = 1240$	$n = 140$

`Jurdzinski`$[n, n]$

n	psol	psolB	psolQ	zlka
10*2**3	215118.70	23045.37	310665.53	abo
10*2**4	abo	403844.56	abo	abo
20min	$n = 110$	$n = 200$	$n = 100$	$n = 50$

Fig. 7. Second experimental results run over `Jurdzinski` benchmarks

`zlka` was best for `Model Checker Ladder` and `Elevator Verification`, and about as good as `psolB` for `Tower Of Hanoi`. And `psolQ` was best for `Recursive Ladder`. Thus `psol` appears to perform worst across these benchmarks.

Solvers `psolB` and `zlka` seem to do about equally well for game types `Clique`, `Ladder`, `Model Checker Ladder`, and `Tower Of Hanoi`. But solver `psolB` appears to outperform `zlka` dramatically for game types `Recursive Ladder`, and `Strategy Impr` and is considerably better than `zlka` for `Jurdzinski`.

We think these results are encouraging and corroborate that partial solvers based on fatal attractors may be components of faster solvers for parity games.

6.3 Number of detected fatal attractors

We also recorded the number of fatal attractors that were detected in runs of our partial solvers. One reason for doing this is to see whether game types have a typical number of dynamically detected fatal attractors that result in the complete solving of these games.

We report these findings for `psol` and `psolB` first: for `Clique`, `Ladder`, and `Strategy Impr` these games are solved by detecting two fatal attractors only; `Model Checker Ladder` was solved by detecting one fatal attractor. For the other game types `psol` and `psolB` behaved differently. For `Recursive Ladder`$[n]$, `psolB` requires $n = 2^k$ fatal attractors whereas `psolQ` needs only 2^{k-2} fatal attractors. For `Jurdzinski`$[n, m]$, `psolB` detects $mn + 1$ many fatal attractors, and `psol` removes x edges where x is about $nm/2 \leq x \leq nm$, and detects slightly more than these x fatal attractors. Finally, for `Tower Of Hanoi`$[n]$, `psol` requires the detection of 3^n fatal attractors whereas `psolB` solves these games with detecting two fatal attractors only.

We also counted the number of recursive calls for `psolQ`: it equals the number of fatal attractors detected by `psolB` for all game types except `Recursive Ladder`, where it is 2^{k-1} when n equals 2^k.

6.4 Experiments on variants of partial solvers

We performed additional experiments on variants of these partial solvers. Here, we report results and insights on two such variants. The first variant is one that modifies the definition of the monotone control predecessor to

$$\begin{aligned}\mathsf{mpre}_p(A,X,c) = \{v \in V_p \mid ((c(v)\%2 = p) \vee c(v) \geq c) \wedge v.E \cap (A \cup X) \neq \emptyset\} \cup \\ \{v \in V_{1-p} \mid ((c(v)\%2 = p) \vee c(v) \geq c) \wedge v.E \subseteq A \cup X\}\end{aligned}$$

The change is that the constraint $c(v) \geq c$ is weakened to a disjunction $(c(v)\%2 = p) \vee (c(v) \geq c)$ so that it suffices if the color at node v has parity p even though it may be smaller than c. This implicitly changes the definition of the monotone attractor and so of all partial solvers that make use of this attractor; and it also impacts the computation of A within `psolQ`. Yet, this change did not have a dramatic effect on our partial solvers. On our benchmarks, the change improved things slightly for `psol` and made it slightly worse for `psolB` and `psolQ`.

A second variant we studied was a version of `psol` that removes at most one edge in each iteration (as opposed to all edges as stated in Fig. 2). For games of type `Ladder`, e.g., this variant did much worse. But for game types `Model Checker Ladder` and `Strategy Impr`, this variant did much better. The partial solvers based on such variants and their combination are such that `psolB` (as defined in Figure 3) is still better across all benchmarks.

6.5 Experiments on random games

It is our belief that comparing the behavior of parity game solvers on random games does not give an impression of how these solvers perform on parity games in practice. However, evaluating our partial solvers over random games gives an indication of how often partial solvers completely solve random games, and of whether partial solvers can speed up complete solvers as preprocessors. So we generated $130,000$ random games with the `randomgame` command of PGSolver.

Each game had between 10 and 500 nodes (average of 255). Each node v had out-degree (i.e. the size of $v.E$) at least 1, and at most 2, 3, 4, or 5 – where this number was determined at random. These games contained no self-loops and no bound on the number of different colors. Then `psolB` solved 82% of these $130,000$ random games completely. The average run-time over these $130,000$ games was 319ms for `psolB` (which includes run-time of `zlka` on the residual game where applicable), whereas the full solver `zlka` took 505ms on average. And only about $22,000$ of these games (less than 17%) were such that `zlka` solved them faster than the variant of `zlka` that used `psolB` as preprocessor.

7 Conclusions

We proposed a new approach to studying the problem of solving parity games: partial solvers as polynomial algorithms that correctly decide the winning status of some nodes and return a sub-game of nodes for which such status cannot be decided. We demonstrated the feasibility of this approach both in theory and

in practice. Theoretically, we developed a new form of attractor that naturally lends itself to the design of such partial solvers; and we proved results about the computational complexity and semantic properties of these partial solvers. Practically, we showed through extensive experiments that these partial solvers can compete with extant solvers on benchmarks – both in terms of practical running times and in terms of precision in that our partial solvers completely solve such benchmark games.

In future work, we mean to study the descriptive complexity of the class of output games of a partial solver, for example of `psolQ`. We also want to research whether such output classes can be solved by algorithms that exploit invariants satisfied by these output classes. Furthermore, we mean to investigate whether classes of games characterized by structural properties of their game graphs can be solved completely by partial solvers. Such insights may connect our work to that of [3], where it is shown that certain classes of parity games that can be solved in PTIME are closed under operations such as the join of game graphs. Finally, we want to investigate whether and how partial solvers can be integrated into solver design patterns such as the one proposed in [5].

A technical report [7] accompanies this paper and contains – amongst other things – selected proofs, the pseudo-code of our version of Zielonka's algorithm, and further details on experimental results and their discussion.

References

1. Dietmar Berwanger, Anuj Dawar, Paul Hunter, and Stephan Kreutzer. DAG-width and parity games. In *STACS*, LNCS 3884, pages 524–436. Springer-Verlag, 2006.
2. Krishnendu Chatterjee and Monika Henzinger. An $O(n^2)$ time algorithm for alternating Büchi games. In *SODA*, pages 1386–1399, 2012.
3. Christoph Dittmann, Stephan Kreutzer, and Alexandru I. Tomescu. Graph operations on parity games and polynomial-time algorithms. arXiv:1208.1640, 2012.
4. E.A. Emerson and C. Jutla. Tree automata, μ-calculus and determinacy. In *FOCS*, pages 368–377, 1991.
5. Oliver Friedmann and Martin Lange. Solving parity games in practice. In *ATVA*, LNCS 5799, pages 182–196. Springer, 2009.
6. Oliver Friedmann and Martin Lange. The PGSolver Collection of Parity Game Solvers. Tech report, Institut für Informatik, LMU Munich, Feb 2010. Version 3.
7. Michael Huth, Jim Huan-Pu Kuo, and Nir Piterman. Fatal attractors in parity games. Tech report, Dep. of Computing, Imperial College London, Jan 2013.
8. M. Jurdziński. Small progress measures for solving parity games. In *STACS*, LNCS 1770, pages 290–301. Springer-Verlag, 2000.
9. M. Jurdziński, M. Paterson, and U. Zwick. A deterministic subexponential algorithm for solving parity games. In *SODA*, pages 117–123. ACM/SIAM, 2006.
10. J. Vöge and M. Jurdziński. A discrete strategy improvement algorithm for solving parity games. In *CAV*, LNCS 1855, pages 202–215. Springer-Verlag, 2000.
11. W. Zielonka. Infinite games on finitely coloured graphs with applications to automata on infinite trees. *Theoretical Computer Science*, 200(1–2):135–183, 1998.