Fatal Attractors in Parity Games

January 21, 2013

Michael Huth and Jim Huan-Pu Kuo
Department of Computing
Imperial College London
London, SW7 2AZ, United Kingdom
{m.huth, jimhkuo}@imperial.ac.uk

Nir Piterman
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. 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. [1]

1 Introduction

Parity games are an important foundational structure in formal verification (see e.g. [10]). 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.

[1]Please cite this technical report as "Michael Huth, Jim Huan-Pu Kuo, and Nir Piterman. Fatal attractors in parity games. Technical report, Department of Computing, Imperial College London, January 2013". A preliminary version of the results reported in this paper was presented at the GAMES 2012 workshop in Naples, Italy, on 11 September 2012.

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 [10, 7, 9].

Research on solving parity games may be losely grouped into the following approaches: design of algorithms that solve all parity games by construction and that so far all have exponential or subexponential worst-case complexity (e.g. [10, 7, 9, 8]), 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 subgame of the input game, and a classification of the winning status of all nodes of the input game that are not in that subgame. In particular, the returned subgame 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 intesting 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. This technical report also contains an appendix that 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.

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 *subgame* 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).

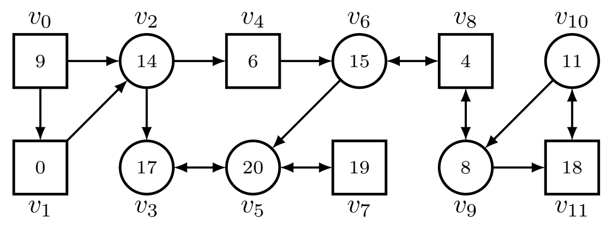

Figure 1: A parity game: circles denote nodes in V_0, squares denote nodes in V_1.

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*

$$\begin{aligned} \mathsf{cpre}_p(X) &= \{v \in V_p \mid v.E \cap X \neq \emptyset\} \cup \{v \in V_{1-p} \mid v.E \subseteq X\} && (1) \\ \mathsf{Attr}_p[G, X] &= \mu Z.(X \cup \mathsf{cpre}_p(Z)) && (2) \end{aligned}$$

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.

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 subgame. 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 subgame of G.*

We now define a more general 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*

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

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 $\mathsf{MA}(k)$

```
psol($G = (V, V_0, V_1, E, c)$) {
  for ($k \in V$ in descending color ordering $c(k)$) {
    if ($k \in \mathsf{MA}(k)$) { return psol($G \setminus \mathsf{Attr}_{p(k)}[G, \mathsf{MA}(k)]$) }
    if ($\exists\,(k, w) \in E$: $w \in \mathsf{MA}(k)$)
    { $G = G \setminus \{(k, w) \in E \mid w \in \mathsf{MA}(k)\}$ }
  }
  return $G$
}
```

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

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 colour 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.

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.

```
psolB($G = (V, V_0, V_1, E, c)$) {
  for (colors $d$ in descending ordering) {
    $X$ = { $v$ in $V$ | $c(v)$ = $d$ };
    cache = {};
    while ($X \neq$ {} && $X \neq$ cache) {
      cache = $X$;
      if ($X \subseteq$ $\mathsf{MA}(X)$) { return psolB($G$ \ $\mathsf{Attr}_{d\%2}[G, \mathsf{MA}(X)]$)
      } else { $X$ = $X \cap$ $\mathsf{MA}(X)$; }
    }
  }
  return $G$
}
```

Figure 3: Partial solver `psolB`.

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.

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 subgame $P_1(G)$ is a subset of the set of nodes of the output subgame $P_2(G)$. We note that `psol` and `psolB` are incomparable for this intensional preorder 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.

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*

$$\begin{aligned}
\mathsf{pmpre}_p(A, X, c) &= \{v \in V_p \mid (c(v) \geq c \vee v \in X) \wedge v.E \cap (A \cup X) \neq \emptyset\} \cup \\
&\quad\ \{v \in V_{1-p} \mid (c(v) \geq c \vee v \in X) \wedge v.E \subseteq A \cup X\} \qquad (4) \\
\mathsf{PMAttr}_p(X, c) &= \mu Z.\mathsf{pmpre}_p(Z, X, c) \qquad (5)
\end{aligned}$$

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 allowes 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}

```
layeredAttr($G$,$p$,$X$) { // PRE-CONDITION: all nodes in $X$ have parity $p$
   $A$ = {};
   $b$ = max{$c(v)$ | $v \in X$};
   for ($d$ = $p$ up to $b$ in increments of 2) {
      $Y$ = {$v \in X$ | $c(v) \leq d$};
      $A$ = $\mathsf{PMAttr}_p(A \cup Y, d)$;
   }
   return $A$;
}

psolQ($G = (V, V_0, V_1, E, c)$) {
   for (colors $b$ in ascending order) {
      $X$ = { $v \in V$ | $c(v) \leq b \wedge c(v)\%2 = b$};
      cache = {};
      while ($X \neq$ {} && $X \neq$ cache) {
         cache = $X$;
         $W$ = layeredAttr($G$,$b\%2$,$X$);
         if ($X \subseteq W$) { return psolQ($G \setminus \mathsf{Attr}_{b\%2}[G, W]$);
         } else { $X$=$X \cap W$; }
      }
   }
   return $G$;
}
```

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

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*

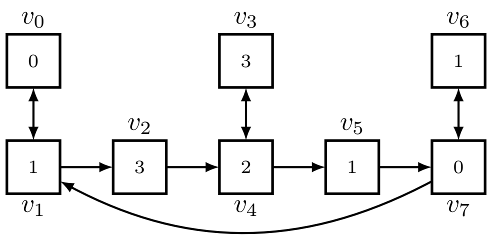

Figure 5: A 1-player parity game modified by neither `psol`, `psolB` nor `psolQ`.

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.*

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 decending 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 subgames.

Let us formalize this. Let π be some sequence of colors in G, that may omit or repeat some colors from G. Let `psolB`(π) 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 `psolB`(π) is *stable* if for every color c_1, the input/output behavior of `psolB`(π) and `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 `psolB`(π) with one color does not change the input/output behavior.

Theorem 5 *Let* π_1 *and* π_2 *be sequences of colors with* `psolB`(π_1) *and* `psolB`(π_2) *stable. Then* G_1 *equals* G_2 *if* G_i *is the output of* `psolB`(π_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* `psolB`(G) *outputs the empty game. And we define* $\mathcal{K}$ *(for "Kernel") as the class of those parity games* G *for which* `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 `Jurdzkinski` 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 `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`.

Some of these improvements can even be seen by comparing running times of our partial solvers with those of the PGSolver, which we will do below. We stress that this does compare proof of concept implementations of our partial solvers running in JVM with a highly optimized PGSolver running in native code – which is why we omitted the timing information for PGSolver in Figures 6 and 7. We ran PGSolver version 3.2 in configuration `pgsolver -global recursive`, meaning that it is solving parity games using Zielonka's algorithm and that all

`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	$n = 4$	$n = 6$	$n = 6$	$n = 7$

Figure 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	$n = 110$	$n = 200$	$n = 100$	$n = 50$

Figure 7: Second experimental results run over `Jurdzinski` benchmarks

other features are in default mode.

For each game type we compare the running time of PGSolver for the largest power of two for which it does not time out to the running time of our best partial solver for this game type. For `Jurdzinski`$[10, 2^6]$, `psolB` runs about 9 times faster than PGSolver. For `Jurdzinski`$[2^6, 10]$, `psolB` runs about 11 times faster than PGSolver. For `Jurdzinski`$[2^3, 2^3]$, `psolB` runs about 5 times faster than PGSolver. For `Clique`$[2^{12}]$, `psolB` runs about 2 times faster than PGSolver. And for `Recursive Ladder`$[2^5]$, `psolQ` runs about 1706 times faster than PGSolver.

For `Ladder`$[2^{20}]$, PGSolver runs about as fast as `psolB`. For game `Tower Of Hanoi`$[10]$, PGSolver runs about 169 times faster than `psolB`. For `Model Checker Ladder`$[2^{13}]$, PGSolver runs about 1660 times faster than `psolB`. For `Strategy Impr`$[2^{11}]$, PGSolver runs about 47 times faster than `psolB`. And for `Elevator Verification`$[6]$, PGSolver is about 89 times faster than the composition of `psolB` and `zlka` (applied to the output of `psolB`).

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

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

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 outdegree (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 subgame 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].

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 $\mathrm{O}(n^2)$ 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] M. Jurdziński. Small progress measures for solving parity games. In *STACS*, LNCS 1770, pages 290–301. Springer-Verlag, 2000.

[8] M. Jurdziński, M. Paterson, and U. Zwick. A deterministic subexponential algorithm for solving parity games. In *SODA*, pages 117–123. ACM/SIAM, 2006.

[9] J. Vöge and M. Jurdziński. A discrete strategy improvement algorithm for solving parity games. In *CAV*, LNCS 1855, pages 202–215. Springer-Verlag, 2000.

[10] W. Zielonka. Infinite games on finitely coloured graphs with applications to automata on infinite trees. *Theoretical Computer Science*, 200(1–2):135–183, 1998.

A Zielonka's recursive solver

Zielonka's recursive solver is essentially constructed from the steps taken in the determinacy proof of parity games [10]. In order to emphasise the design essence and assist comprehension, the strategy extraction logic has been omitted in our fatal attractor based partial solvers. Therefore, we implement the simplified Zielonka's solver that excludes the logic for winning strategy extraction for fairer performance comparison. The pseudocode of our Zielonka implementation (`zlka`) is shown in Figure 8.

```
zielonka($G = (V, E, c)$) {
  $n$ = max$\{c(v) \mid v \in V\}$
  if ($n$ == 0) { return ($V \setminus Attr_1(G, \emptyset)$, $Attr_1(G, \emptyset)$) }
  $\sigma$ = $n\%2$
  $W_{\overline{\sigma}}$ = win-opponent($G, \sigma, n$)
  $W_\sigma$ = $V \setminus W_{\overline{\sigma}}$
  return ($W_0, W_1$)
}

win-opponent($G, \sigma, n$) {
  $W$ = $\emptyset$
  repeat {
    $W'$ = $W$
    $X$ = $Attr_{\overline{\sigma}}(G, W)$
    $Y$ = $V \setminus X$
    $N$ = $\{v \in Y \mid c(v)$ == $n\}$
    $Z$ = $Y \setminus Attr_\sigma(G[Y], N)$    // $G[Y]$ is a subgame of $G$ restricted to edges and nodes in $Y$
    $(Z_0, Z_1)$ = zielonka($G[Z]$)
    $W$ = $X \cup Z_{\overline{\sigma}}$
  } until ($W'$ == $W$)
  return $W$
}
```

Figure 8: Pseudocode of Zielonka's solver without resolution of the winning strategies

B Analysis of partial solver behaviours on games tested

We discuss the behaviours of our partial solvers on the eight non-random games tested in our experiments.

B.1 `Clique`

For `Clique`$[n]$, `psolB` performs the best in terms of termination boundary, followed by `zlka`. However, in terms of running time, `psolQ` does the best amongst all solvers towards larger game sizes, outperforms `zlka` by around 3 times as much for the largest game (`Clique`$[2^{12}]$) tested before all solvers aborted. `Clique`$[n]$ games are fully connected parity games without self-loops. The input parameter n specifies the number of nodes. The node set is divided into two equal number of V_0 and V_1 nodes (when $|V|$ is even, otherwise $|V_0| = |V_1|$

+ 1). For $p = 0$ and 1, V_p nodes are owned entirely by player p, every node has an unique colour, and for all $v \in V_p$, $c(v)\%2 = p$.

Apart from for n=2, `psol` and `psolB` solve all `Clique`[n] games by finding two fatal attractors. Let c_m be the largest colour in `Clique` game G, then the first fatal attractor A_1 detected consists of two nodes of the largest colours in $c_m\%2$ parity, followed by computing $Attr_{c_m\%2}(G, A_1)$ which attracts all $c_m\%2$ parity nodes. This makes up the winning region of player $c_m\%2$. The second fatal attractor A_2 consists of two nodes with the largest $1-c_m\%2$ parity colours, then the winning region for player $1 - c_m\%2$ is $Attr_{1-c_m\%2}(G, A_2)$. Therefore, the winning regions are divided such that W_p (for $p = 0$ and 1) consists of all V_p nodes.

Partial solver `psolQ` detects two fatal attractors, for colours 0 and 1, to solve the game.

B.2 `Ladder`

In `Ladder`[n] game, all V_0 nodes have colour 2, and all V_1 nodes have colour 1. Each node $v \in V$ has two successors, one V_0 and one V_1 node, which form a node pair. Every node pair is connected to the next pair to form a “ladder” of node pairs. Finally, the bottom pair is connected to the top to close the loop. The parameter n configures the number of such node pairs.

In Figure 6, we see that `psolB` performs better than `zlka` in terms of running time and terminating boundary, 37% time reduction and 22% boundary increase for `Ladder`[2^{20}]. We do not observe significant performance difference between the partial solvers `psolB`, and `psolQ`. Due to its extremely small index (i.e., 2), regardless the colour exploration ordering, these three partial solvers detect the two fatal attractors consecutively, each consists of all nodes in V_1 and V_0, for colour 1 and 2, respectively.

However, in order to make `Ladder` compatible with `psol`, we need to convert the game to have unique colours, hence, inflating the index from 2 to size of the game. This process unnecessarily increases the computational complexity of the game, and the low performance for `psolB` exhibited in the experimental results backs up this intuition.

B.3 `Model Checker Ladder`

A `Model Checker Ladder`[n] game consists of overlapping blocks of four nodes where n specifies the number of blocks. All nodes are owned by player 1. However, the nodes are connected in such a way that every cycle in the game passes through a single “choke point” node of colour 0. The partial solver iterates through all colours, and eventually detects a colour with a fatal attractor, in the case of `Model Checker Ladder`, this colour is 0. The accumulation process then goes through iterations to eventually attract all of the nodes in the game.

As a result, all partial solvers detect only one fatal attractor (fatal for player 1) in this game structure. In Figure 6, all partial solvers seem fairly close in terms of running times and terminating boundary for `Model Checker Ladder`.

B.4 `Recursive Ladder`

The `Recursive Ladder`$[n]$ game consists of n blocks, each of the blocks consists of four layers of five nodes in total. The node in the bottom layer has connections to two nodes in the next block, whereas the top layer node has a connection back to the top layer node of the previous block, hence forming the "recursive" structure. Different to `Ladder`, the last block in `Recursive Ladder` does not loop back to the first block.

All our partial solvers outperform the recursive solver `zlka` for this game in the experiment (although this is not obvious in Figure 6 as we only show the last two entries). Partial solver `psolQ` is the best performer here among all solvers. It solved `Recursive Ladder`$[2^{13}]$ in around 10 minutes (when all other solvers timed out after 20 minutes) and reached a terminating boundary 355 times larger than what `zlka` achieved.

The experiment results also show that `psolQ` requires fewer fatal attractors than `psol` and `psolB` to solve the game, `psolB` solves `Recursive Ladder`$[n]$ through n $(= 2^k)$ fatal attractors; `psol` through between 2^{k-1} and 2^k fatal attractors and removes about that many edges; `psolQ` detects 2^{k-1} fatal attractors.

B.5 `Strategy Impr`

The structure of this game is more complicated than games introduced previously. A `Strategy Impr`$[n]$ game has $25n + 10$ many nodes. One key property of `Strategy Impr` is that it causes the strategy improvement algorithm [9] to exhibit an exponential running time [6]. In our experiment, the partial solvers can reach higher termination boundaries than `zlka` against this game type. Partial solver `psolB` is the best performer (followed by `psol`), and it is able to solve `Strategy Impr`$[2^{11}]$ in around 10.5 minutes when other solvers, `psolQ`, and `zlka`, timed out after 20 minutes. The termination boundary reached by `psolB` is 111 times greater than that for `zlka`'s.

Although this game has a more complicated structure, it contains a few properties which `psolB` and `psol` exploit. `Strategy Impr` (min-parity) games contain a V_1 node v_s that has the maximum colour, which is odd, and has a self-loop. As well as a pair of high colour odd parity V_0 nodes, from which player 0 can either go to v_s (which is bad for player 0), or stay with them (also bad for player 0). For descending colour ordering, `psolB`/`psol` immediately identifies the fatal attractor $\mathsf{MA}(\{v_s\})/\mathsf{MA}(v_s)$, followed by normal attractor computation for player 1 to it. The next fatal attractor is found after 3 colour iterations, the normal attractor computed for that fatal attractor includes the whole of the remaining game. As a result, the entire game is won by player 1.

Partial solver `psolQ` solves this game in a similar manner. The difference is that it finds the required fatal attractors at very late stages of the process due to its colour exploration ordering. The wasted iterations cause significant inefficiency, this is reflected by the poorer experimental results.

We note that the number of fatal attractors detected by all partial solvers to solve `Strategy Impr`$[n]$ are identical (2) regardless of the size of n.

B.6 `Towers Of Hanoi`

In `Towers Of Hanoi`$[n]$ game, all nodes belong to player 0, and have colour 1 or 2. This game has a intertwined tower structure that captures a well-known puzzle. Each tower in the game consists of four nodes, with outgoing edges to other towers. Notably, every tower, except one, has an odd colour (i.e., 1) node with self-loop. The node with self-loop in the exception tower has even colour. Regardless of the value of n in a `Towers Of Hanoi`$[n]$ game, the number of such exceptions is 1.

The result for this game in Figure 6 accounts for the worse performance of `psolQ` whereas `psolB` performs similarly as `zlka`. Let G be a `Towers Of Hanoi` game of any size, `psolB` solves G in two steps. From the maximum colour (i.e., 2), it first detects fatal attractor A_0, consisting of that even parity node with self-loop (denoted as v) and the node that has outgoing edge to v. The attractor $Attr_0(G, A_0)$ makes up all of W_0, and is removed from the game. The remaining game G' after the previous step, only has nodes of colour 1. The next step simply identifies all nodes with self-loops as the fatal attractor A_1. The normal attractor $Attr_1(G', A_1)$ is the W_1 region. Due to the unique colour conversion, `psol` needs many more steps to solve `Towers Of Hanoi`.

Partial solver `psolQ` solves this game in this manner. Starting from the minimum colour (i.e., 1), it detects and removes fatal attractor A_1 and its normal attractor, $Attr_1(G, A_1)$ (which does not contain all nodes with colour 1). Removal of a region causes a recursive call to `psolQ` which resets the boundary colour b to 1. In the next step, `psolQ` tries to find a fatal attractor for colour 1 again, but fails. Then for colour 2, it detects fatal attractor A_0.

The above descriptions of solver steps show that `Towers Of Hanoi` is solved with exactly two fatal attractors by `psolB`, and `psolQ`, regardless of the game size. While `psol` requires 3^n fatal attractors, due to the inflated number of colours caused by the unique colour conversion process.

B.7 `Elevator Verification`

`Elevator Verification` represents a fairness verification problem for an elevator model [6]. An `Elevator Verification`$[n]$ game represents a model with n floors, and has roughly $5^{n-1} \times 37$ number of nodes. This game has three colours (0, 1, and 2) regardless of the size of n.

All games we have seen previously can be solved completely by all partial solvers. Partial solver `psol` does not solve this type of game completely, and `psolB` does not do so for $n > 1$. We therefore add to their running time that of `zlka` on the game that remains unsolved by these partial solvers. Partial solver `psolQ` does solve this type of games completely.

In Figure 6, we see that `zlka` alone performs better then first running any of these partial solver and then `zlka` for `Elevator Verification`. Again, `psolB` performs much better than psol. We note that for larger games the vast proportion of time for the compositions psol;zlka and psolB;zlka is spent in the partial solvers. Although psolQ is able to solve this game completely, it does not seem to perform better than zlka.

Partial solver psolQ solves this game in this manner. For games with n = 1, Elevator Verification[n] is solved by two fatal attractors (and their corresponding normal attractors). For $n > 1$, it finds three fatal attractors this way. Given a Elevator Verification game G, psolQ first attempts to find a fatal attractor for colour variable $d = 0$, but fails to do so. Followed by detection of fatal attractor A_1, and removal of $Attr_1(G, A_1)$ for $d = 1$. The removal of a region causes a recursive call which resets d to 0. In this iteration, psolQ attempts and fails to accumulate a fatal attractor for $d = 0$, and 1, then it succeeds in finding fatal attractor A_2 for $d = 2$. Eventually, psolQ finds fatal attractor A_0 for $d = 0$ in the next iteration when d is reset.

Amongst the three partial solvers, psolB appears to have the best/least running time in the experimental results for Elevator Verification. However, the truth of this observation is polluted by the fact that psolB is unable to solve Elevator Verification completely and its results include the running time of zlka (which performs favourably) on the remaining game. We note that psol has poor performance here and is unable to solve Elevator Verification completely.

B.8 Jurdzinski

The Jurdzinski[n, m] game is designed to cause the worst-case behaviour for the SPM solver [7]. This game consists of n number of layers, while each layer consists of m number of repeating blocks of three nodes. The layers and blocks are inter-connected in the manner described in [7].

As this game has two input parameters n and m, we ran three binary search experiments: one where n is fixed to 10 and binary search is done over k, where $m = 10 \times k$, one in which these roles of n and m are swapped, and a third one where n equals m – and so we treat this third experiment like the other experiments with one parameter only.

The initial configuration is $(n, m) = (10, 10)$ in all three experiments. The results in Figure 7 show that in runs of all three forms, psolB clearly outperforms zlka and does better throughout than other partial solvers. It reaches 4 to 31 times larger terminating boundary than zlka. Additonally, psolB solves (the three largest games) Jurdzinski[$10, 10 \times 2^8$] in around 14 minutes, Jurdzinski[$10 \times 2^8, 10$], and Jurdzinski[$10 \times 2^4, 10 \times 2^4$] in around 7 minutes when all other solvers timed out.

The results also show that greater number of fatal attractors are detected for Jurdzinski. Partial solvers psolB and psolQ detect $nm + 1$ many fatal attractors. Partial solver psol detects slightly more than it removes edges, $nm/2 \leq x \leq n \cdot m$ many edges are removed for Jurdzinski[n, m].

Although psolB finds the same fatal attractors as psolQ (but in different ordering), their performances differ significantly. In psolB solving, the first few fatal attractors detected consist of all nodes of some colours. Hence, the steps taken rapidly reduce the indices, and the computational complexity of the remaining games. However, in `psolQ` solving, a prefix of fatal attractors found consist of only a subset of nodes of some colours. After removing the normal attractors to these fatal attractors, the indices of the remaining games remain the same. This means `psolQ` is required to attempt many "unfruitful" fatal attractor detections on some colours before "real" progress can be made.

The process of `psol` solving is the same as `psolB` in spirit. However, due to the unique colour conversion process, the inflated colour space means that `psol` has to process these "artificial" colours which yield no fatal attractor.

C Proof of Theorem 3

Consider the computation of $\texttt{layeredAttr}(G, p, X)$. Let A_d be an enumeration of the sets A computed by the for loop. Here d is the index of the for loop ranging over $b\%2, \ldots, b$, where b is the bound on the priorities in X. It follows that $A_{i+2} = \mathsf{PMAttr}_p(A_i \cup Y_{i+2}, i+2)$, where Y_{i+2} is the subset of X of nodes of priority at most $i+2$. By definition of $\mathsf{PMAttr}_p(A_i \cup Y_{i+2}, i+2)$ it follows that $A_i \subseteq A_{i+2}$. Indeed, if A_i is attracted to Y_i then the same attractor computation includes all the nodes in A_i in the computation of A_{i+2} as they are now permissible. Let A denote A_b, i.e., the result returned by $\texttt{layeredAttr}(G, p, X)$. For every node $v \in A$, let $r(v) = (d, i)$ where d is the minimal such that $v \in A_d$ and i is the distance to attract to $A_{d-2} \cup Y_d$ in the computation of $\mathsf{PMAttr}_p(A_{d-2} \cup Y_d, d)$. Consider the strategy for player $b\%2$ that from v minimizes the rank $r(v')$ according to the lexicographic order on the rank.

We show that every infinite play conforming to this strategy remains forever in $\texttt{layeredAttr}(G, p, X)$. Indeed, if $r(v) = (d, i)$, then all successors of v (if $v \in V_{1-b\%2}$) or some successor of v (if $v \in V_{b\%2}$) are/is either in X, which is a subset of A, or in $A_{d',i'}$ for some $(d', i') < r(v)$ (successors of v are in $Y_d \subseteq X$, A_{d-2}, or closer to $Y_d \cup A_{d-2}$). When reaching X, the same strategy can be applied again as $X \subseteq \texttt{layeredAttr}(G, p, X)$.

Second, we show that the play is winning for player $b\%2$. Consider node v_0 whose color has parity $1 - b\%2$ appearing in the play. Let $v_0, v_1, \ldots$ be an enumeration of the nodes in the play starting from v_0. By definition, $r(v_0) = (d_0, i_0)$ for some (d_0, i_0), and clearly, $c(v_0) > d_0$. We show that this play visits color of parity $b\%2$ that is at most d_0. By construction, v_1 is either in $\{v \in X \mid c(v) \leq d_0\}$, which implies that its color is of parity $b\%2$ and smaller than $c(v_0)$, or $r(v_1) = (d_1, i_1)$ for $(d_1, i_1) < (d_0, i_0)$. In this case, we change the obligation to visit a $b\%2$-parity color that is at most d_0 to visit a $b\%2$-parity color at most d_1 and pass it to v_1. Continuing this way, the play must reach X with a lower color than that of v_0 by well-founded induction.

D Proof of Item 1 of Theorem 4, `psol` Soundness

In Figure 2, `psol` only returns (not explicitly shown) $\mathsf{Attr}_{p(k)}[G, \mathsf{MA}(k)]$ as a node set classified to be won by player $p(k)$ whenever $\mathsf{MA}(k)$ is fatal. Theorem 2 shows that these regions are winning for player $p(k)$. Lemma 1 shows edge removal does not alter the winning strategies. Since these are the only two code locations where G is modified, the winning regions detected in `psol` are correct. □

E Proof of Item 1 of Theorem 4, `psolB` Soundness

In Theorem 2, we have proved that $\mathsf{MA}(X)$ is winning for player $p(X)$ if X is a subset of $\mathsf{MA}(X)$. For every color d in G, the for-loop in `psolB` constructs $\mathsf{MA}(X)$ where all nodes in X have color d. If X is a subset of $\mathsf{MA}(X)$, then $\mathsf{MA}(X)$ is identified as a winning region (for player $d\%2$) and its normal $d\%2$ attractor in G is therefore removed from G, and this is the only code location where G is modified. □

F Proof of Item 1 of Theorem 4, `psolQ` Soundness

By Theorem 3, we have proved that $\texttt{layeredAttr}(G, p, X)$ is winning for player $p(X)$ if X is a subset of $\texttt{layeredAttr}(G, p, X)$. For every color b in G, the for loop in `psolQ` constructs $\texttt{layeredAttr}(G, p, X)$, where X is the set of nodes of parity $b\%2$ with color at most b. If X is a subset of $\texttt{layeredAttr}(G, p, X)$, then $\texttt{layeredAttr}(G, p, X)$ is identified as a winning region (for player $b\%2$) and its normal $b\%2$ attractor in G is therefore removed from G, and this is the only code location where G is modified. □

G Proof of Items 2, 3, and 4 of Theorem 4

2. To see that the running time for `psol` is in $O(|V|^2 \cdot |E|)$, note that all nodes have at least one successor in G and so $|V| \leq |E|$. The computation of the attractor $\mathsf{MA}(k)$ in linear in the number of edges and so in $O(|E|)$. Each call of `psol` will compute at most $|V|$ many such attractors. In the worst case, there are $|V|$ many recursive calls. In summary, the running time is bound by $O(|E| \cdot |V| \cdot |V|)$ as claimed.

   To see that `psolB` also has running time in $O(|V|^2 \cdot |E|)$, recall that we may compute $\mathsf{MA}(X)$ in time linear in $|E|$. Second, node set V is partitioned into sets of nodes of a specific color, and so `psolB` can do at most $|V|$ many computations within the body of `psolB` before and if a recursive call happens.

3. The claim that `psol` and `psolB` can be implemented to run in $O(|V|^3)$ essentially reduces to showing that we can, in linear time, transform and reduce each computation of $\mathsf{MA}(X)$ to the solution of a Buchi game. This

is so since such games can be solved in time $O(|V|^2)$ [2]. Indeed, let c denote $c(X)$, p denote $p(X)$, and let $G[\geq c]$ denote the game obtained from G by doing the following in the prescribed order.

(a) Remove from G all nodes of color less than c, as well as all of their incoming and outgoing edges.

(b) Add to G a sink node that has a self loop.

(c) Every node in V_p not removed in the first step but where all of its successors were removed gets an edge to the new sink node.

(d) Every node in V_{1-p} not removed in the first step but that had one of its successors removed gets an edge to the new sink node as well.

(e) If $p = 1$, then we swap ownership of all remaining nodes: player 0 nodes become player 1 nodes, and vice versa.

(f) Finally, we color every node in X by p and all other nodes (including the new sink state) by $1 - p$.

It is possible to show that the winning region in $G[\geq c]$ is $\mathsf{MA}(X)$. Indeed, every node in the winning region of $G[\geq c]$ can be attracted to X without passing through colors smaller than c infinitely often. In the other direction, the attractor strategy to X induced by $\mathsf{MA}(X)$ can be converted to a winning strategy in $G[\geq c]$. The size of $G[\geq c]$ is bounded by the size of G: there is at most one more node (the sink state), and each edge added to $G[\geq c]$ has a corresponding edge that is removed from G.

4. As before, the computation of `layeredAttr`(G, p, b) can be completed in $O(|V| \cdot |E|)$. Denote $A_{b\%2-2} = \emptyset$ and $A_d = \mathsf{PMAttr}_p(A_{d-2} \cup Y_d, d)$ for $d = b\%2, \ldots, b$. As noted previously, $A_{d-2} \subseteq A_d$. Hence, the entire run of the for loop can be implemented so that each edge is crossed at most once in all the permissive monotone control predecessor computations.

   Then, the loop on X in `psolQ` can run at most $|V|$ times. And the extenal for loop runs at most $|c|$ times. It follows that `layeredAttr` is called at most $|V| \cdot |c|$. □

H Proof of robustness of `psolB`

In order to prove Theorem 5 we first prove a few auxiliary lemmas. Below, we write $G[U]$ for the subgame identified by node set U.

Lemma 2 *For every game G, for every set of nodes K and for every trap U for player p, the following holds:* $\mathsf{Attr}_p[G, K] \cap U \subseteq \mathsf{Attr}_p[G[U], K \cap U]$

Proof: The proof proceeds by induction on the distance from K in $\mathsf{Attr}_p[G, K]$. For every node v of G let $d(v)$ denote the distance of v from K in the attraction to K in G.

- Suppose that $K \cap U = \emptyset$. Then, $\mathsf{Attr}_p[G[U], K \cap U] = \emptyset$ and we have to show that $\mathsf{Attr}_p[G, K] \cap U = \emptyset$.

  Assume otherwise, then $v \in \mathsf{Attr}_p[G, K] \cap U \neq \emptyset$. Let v be the node of minimal distance to K in $\mathsf{Attr}_p[G, K] \cap U$. If $v \in V_p$, then there is some successor w of v such that $d(v) = d(w) + 1$. However, w cannot be in $\mathsf{Attr}_p[G, K] \cap U$ by minimality of v. Thus, there is an edge from v that leads to a node not in U contradicting that U is a trap for player p. Similarly, if $v \in V_{1-p}$, then for all successors w of v we have $d(v) > d(w)$ and it follows that all succssors w of v are not in $\mathsf{Attr}_p[G, K] \cap U$. So all successors of v are not in U and U cannot be a trap for player p.

  It follows that $\mathsf{Attr}_p[G, K] \cap U = \emptyset$ as required.

- Suppose that $K \cap U \neq \emptyset$. We prove that for every node $v \in \mathsf{Attr}_p[G, K] \cap U$ we have $d_G(v, K) \geq d_{G[U]}(v, K \cap U)$, where $d_G(v, K)$ and $d_{G[U]}(v, K \cap U)$ are the distances of v from K (respectively $K \cap U$) in the computation of the corresponding attractor.

  Again, the proof proceeds by induction on $d_G(v, K)$. Consider a node v in $\mathsf{Attr}_p[G, K] \cap U$ such that $d_G(v, K) = 0$. Then v is in K and from $v \in U$ we conclude that v is in $K \cap U$ and $d_{G[U]}(v, K \cap U) = 0$.

  Consider a node v in $\mathsf{Attr}_p[G, K] \cap U$ such that $d_G(v, K) > 0$. If v is in V_p, then there is a node w such that $d_G(v, K) = d_G(w, K) + 1$. Since U is a trap, it must be the case that w is in U as well and hence w is in $\mathsf{Attr}_p[G, K] \cap U$. By induction $d_G(w, K) \geq d_{G[U]}(w, K \cap U)$.

  If v is in V_{1-p}, then for all successors w of v we have $d_G(v, K) \geq d_G(w, K) + 1$. Furthermore by U being a trap, there is some successor w of v such that w is in U. It follows that w is in $\mathsf{Attr}_p[G, K] \cap U$.

  As U is a subset of the nodes of G we have $succ(v, G) \supseteq succ(v, G[U])$, where $succ(v, G)$ is the set of successors of v in G and $succ(v, G[U])$ is the set of successors of v in $G[U]$. But then, for every w in $succ(v, G[U])$ we have $d_{G[U]}(w, K \cap U) \leq d_G(w, K)$. Hence, $d_{G[U]}(v, K \cap U) \leq d_G(v, K)$. $\square$

We now specialize the above to the case of monotone attractors. We narrow the scope in this context to match its usage in `psolB`. A more general claim talking about general sets in the spirit of Lemma 2 requires quite cumbersome notations and we skip it here (as it is not needed below).

Lemma 3 *Consider a game G and a set of nodes K of color c such that $p = c\%2$. For every trap U for player p, the following holds:* $\mathsf{MAttr}_p(K, c) \cap U$ *computed in G is a subset of* $\mathsf{MAttr}_p(K \cap U, c)$ *computed in $G[U]$.*

The proof is very similar to the proof of Lemma 2.

Proof: The proof proceeds by induction on the distance from K in $\mathsf{MAttr}_p(K, c)$. For every node v of G let $d(v)$ denote the distance of v from K in the monotone attraction to target K in G.

- Suppose that $K \cap U = \emptyset$. Then, $\mathsf{MAttr}_p(K \cap U, c)$ in $G[U]$ is empty and we have to show that $\mathsf{MAttr}_p(K, c)$ in G has empty intersection with U.

  Assume otherwise, then there is some v such that v is in $\mathsf{MAttr}_p(K, c)$ in G and $v \in U$. Let v in U be the node of minimal distance to K in $\mathsf{MAttr}_p(K, c)$ computed in G. If $d(v) = 1$ and $v \in V_p$, then v has some node in K as successor. But $K \cap U = \emptyset$ and v has a successor outside U contradicting that U is a trap. If $d(v) = 1$ and v is in V_{1-p}, then all successors of v are in K. As $K \cap U = \emptyset$ all successors of v are outside U contradicting that U is a trap. If $d(v) > 1$, the case is similar. If v is in V_p, then there is some successor w of v such that $d(v) = d(w) + 1$. However, w cannot be in $\mathsf{MAttr}_p(K, c) \cap U$ computed in G, by the minimality of v. Thus, there is an edge from v that leads to a node not in U contradicting that U is a trap for player p. Similarly, if v is in V_{1-p}, then for all successors w of v we have $d(v) > d(w)$ and it follows that all succssors w of v are not in $\mathsf{MAttr}_p(K, c) \cap U$ in G. So all successors of v are not in U and U cannot be a trap for player p.

  It follows that $\mathsf{MAttr}_p(K, c)$ computed in G does not intesect U as required.

- Suppose that $K \cap U \neq \emptyset$. We prove that for every node v in $\mathsf{MAttr}_p(K, c) \cap U$ computed in G we have $d_G(v, K) \geq d_{G[U]}(v, K \cap U)$, where $d_G(v, K)$ and $d_{G[U]}(v, K \cap U)$ are the distances of v from K (respectively $K \cap U$) in the computation of the corresponding monotone attractors.

  Again, the proof proceeds by induction on $d_G(v, K)$. Consider a node v in $\mathsf{MAttr}_p(K, c)$ computed in G such that v is in U and $d_G(v, K) = 1$. Then, if v is in V_p, then v has a successor in K. As U is a trap, it must be the case that this successor is also in U showing that $d_{G[U]}(v, K \cap U) = 1$. If v is in V_{1-p}, then all of v's successors are in K. As U is a trap, v must have some successors in $G[U]$. It follows that $d_{G[U]}(v, K \cap U) = 1$.

  Consider a node in $\mathsf{MAttr}_p(K, c)$ such that v is in U and $d_G(v, K) > 1$. If v is in V_p then there is a node w such that $d_G(v, K) = d_G(w, K) + 1$. By U being a trap, it must be the case that w is in U as well and hence w is in $\mathsf{MAttr}_p(K, c) \cap U$ computed in G. By induction $d_G(w, K) \geq d_{G[U]}(w, K \cap U)$.

  If v is in V_{1-p}, then for all successors w of v we have $d_G(v, K) \geq d_G(w, K) + 1$. Furthermore by U being a trap, there is some w successor of v such that w is in U. It follows that all such w are in $\mathsf{MAttr}_p(K, c) \cap U$ computed in G.

  As U is a subset of the nodes of G, we have $\mathit{succ}(v, G) \supseteq \mathit{succ}(v, G[U])$, where $\mathit{succ}(v, G)$ is the set of successors of v in G and $\mathit{succ}(v, G[U])$ is the set of successors of v in $G[U]$. But then, for every w in $\mathit{succ}(v, G[U])$ we have $d_{G[U]}(w, K \cap U) \leq d_G(w, K)$. Hence, $d_{G[U]}(v, K \cap U) \leq d_G(v, K)$. □

We now show that the order of removal of attractors for even and odd colors are interchangeable.

Lemma 4 *Removal of fatal attractors for even colors and for odd colors are interchangeable.*

Proof: Let c_1 be some odd color and c_0 be some even color. Let X_1 be the set of nodes of color c_1 such that $X_1 \subseteq \mathsf{MAttr}_1(X_1, c_1)$ and X_1 is the maximal node set with this property. (That is to say, X_1 is the set computed by a call to `psolB` with the color c_1.) Similarly, let X_0 be the set of nodes of color c_0 such that $X_0 \subseteq \mathsf{MAttr}_0(X_0, c_0)$ and X_0 is the maximal with this property. We assume that both $\mathsf{MAttr}_1(X_1, c_1)$ and $\mathsf{MAttr}_0(X_1, c_1)$ are not empty.

By soundness, $\mathsf{MAttr}_1(X_1, c_1)$ is part of the winning region for player 1. Let U be the residual game $G \backslash \mathsf{Attr}_1[G, \mathsf{MAttr}_1(X_1, c_1)]$. We note that Lemma 2 does not help us directly. Indeed, node set $\mathsf{Attr}_1[G, \mathsf{MAttr}_1(X_1, c_1)]$ is an attractor for player 1. Hence, U is a trap for player 1 but not necessarily for player 0.

By soundness, $\mathsf{MAttr}_0(X_0, c_0)$ is a subset of U. Indeed, all the nodes that are removed from G are winning for player 1 but $\mathsf{MAttr}_0(X_0, c_0)$ is part of the winning region for player 0. It follows that X_0 is a subset of U.

Furthermore, $\mathsf{MAttr}_0(X_0 \cap U, c_0)$ is a superset of $\mathsf{MAttr}_0(X_0, c_0)$, where this follows from an argument similar to the one made in the proof of Lemma 2 above.

But from the construction of $\mathsf{MAttr}_0(X_0 \cap U, c_0)$ it follows that node set $\mathsf{MAttr}_0(X_0 \cap U, c_0)$ is also a subset of $\mathsf{MAttr}_0(X_0, c_0)$. Indeed, if we consider the entire doubly nested fixpoint, then the computation of $\mathsf{MAttr}_0(X_0 \cap U, c_0)$ starts from a subset of the nodes of color c_0 and $\mathsf{MAttr}_0(X_0, c_0)$ starts from the entire set of nodes of color c_0. $\square$

It follows that we may think about the removal of (attractors of) fatal attractors separately for all the even colors and all the odd colors. We now restate and then prove Theorem 5:

Theorem 7 *Let π_1 and π_2 be sequences of colors with* `psolB`(π_1) *and* `psolB`(π_2) *stable. Then G_1 equals G_2 if G_i is the output of* `psolB`(π_i) *on G, for $1 \leq i \leq 2$.*

Proof: By Lemma 4, we may assume that in both π_1 and π_2 all even colors occur before odd colors. We show that the node set of the output of version `psolB`$(\pi_1 \cdot \pi_2)$ is a subset of the node set of the output of version `psolB`(π_2). As π_1 is stable, it follows that actually `psolB`$(\pi_1) \subseteq$ `psolB`(π_2). The same argument works in the other direction and it follows that the two residul games are actually equivalent.

Let $\pi_1 = c^1_1 \cdots c^1_n$, where $c^1_1, \ldots, c^1_m$ are even and $c^1_{m+1}, \ldots, c^1_n$ are odd. Let G^1_0, G^1_1, $\ldots$, G^1_n be the sequence of games after the different applications of the colors in π_1. That is, $G^1_0 = G$, and G^1_i is the result of applying `psolB` with color c^1_i on G^1_{i-1}. It follows that $G^1_n = G_1$. Similarly, let $\pi_2 = c^2_1 \cdots c^2_p$, where $c^2_1, \ldots, c^2_q$ are even and $c^2_{q+1}, \ldots, c^2_p$ are odd. Let $G^2_0 = G$ and let G^2_i be the result of applying `psolB` with color c^2_i on G^2_{i-1}. Let $G^{1,2}_0 = G^1_n$ and $G^{1,2}_i$ is the result of applying `psolB` with color c^2_i on $G^{1,2}_{i-1}$. We show that $G^{1,2}_j$ is a subset of G^2_j.

By Lemma 4 it is clear that we can consider the application of $c^2_1, \ldots, c^2_q$ right after the application of $c^1_1, \ldots, c^1_m$. Indeed, in the sequence $c^1_{m+1}, \ldots, c^1_n$ is interchangeable with $c^2_1, \ldots, c^2_q$.

Consider the application of c^2_j to $G^{1,2}_{j-1}$ and to G^2_{j-1}. By induction $G^{1,2}_{j-1}$ is a subset of G^2_{j-1}. Furthermore, $G^{1,2}_{j-1}$ is obtained from G by removing a sequence of attractors for player 0. It follows that $G^{1,2}_{j-1}$ is G^2_{j-1} restricted to a trap for player 0.

It follows from Lemmas 3 and 2 that the computation of the attractor removes a larger part of $G^{1,2}_{j-1}$ than that of G^2_{j-1}. Hence $G^{1,2}_j$ is a subset of G^2_j. $\square$

I Proof of Theorem 6

We recall one way of solving a Büchi game will take the perspective of player 0. First we inductively define, for $n \geq 0$, and $X = \{v \in V \mid c(v) = 0\}$ the sets

$$\begin{aligned} Z^0 &= V \\ U^n &= \mathsf{Attr}_0[G, Z^n] \\ Y^n &= \mathsf{cpre}_0(U^n) \\ Z^{n+1} &= Y^n \cap X \end{aligned} \qquad (6)$$

Let n_0 be minimal such that $Z^{n_0} = Z^{n_0+1}$. The winning region for W_0 for player 0 in game G with colors 0 and 1 only is then equal to

$$W_0 = \mathsf{Attr}_0[G, Z^{n_0}] \qquad (7)$$

Since the order of processing colors in `psolB` does not impact its output game (by Theorem 5), we may assume that color $d = 0$ gets processed first (this is just for convenience of presentation).

When the first iteration of `psolB` does process $d = 0$, the computation essentially captures the process defined in the equations (6): the interplay of U^n and Y^n achieves the effect that player 0 can move from Y^n into U^n, which models that player 0 can reach the target set again from any node in the target set. The computation of Z^n corresponds to the `else` branch of the iteration within `psolB`. The constraint of our monotone attractor, that $c(v) \geq d$, is vacuously true here as d equals 0. So the first iteration will effectively compute set Z^{n_0} as fixed-point. Then `psolB` will be called recursively on $G \setminus W_0$ by the definition of W_0 in (7).

In that remaining game, player 1 can secure that all plays visit nodes of color 0 only finitely often. This follows from the fact that W_0 was removed from game G and that Büchi games are determined. In particular, `psolB` will not detect a fatal attractor for $d = 0$ in that remaining game. But when its iteration runs with $d = 1$ we argue as follows.

The following algorithm computes the winning region for player 1 in a Büchi game. Let $X = \{v \in V \mid c(v) = 1\}$.

$$\begin{aligned} Z^0 &= \emptyset \\ Y^{n,0} &= X \\ Y^{n,m} &= X \cap \mathsf{cpre}_1(Z^{n-1} \cup Y^{n,m-1}) \\ Z^n &= \mathsf{Attr}_1[G, Y^{n,m_0^n}] \end{aligned} \tag{8}$$

where m_0^n is the minimal natural number such that Y^{n,m_0^n} equals Y^{n,m_0^n+1}. Let n_0 be the minimal natural number such that Z^{n_0} equals Z^{n_0+1}. Let $X^{i,j}$ denote the sequence of values computed for the variable X in `psolB`, where i is the number of recursive invocations of `psolB`, and j is the value of X computed after running in the loop j times.

It is simple to see that $X^{n,m}$ is a superset of $Y^{n,m}$ restricted to the residual game in the nth call to `psolB`. Indeed, both start from the set X and the computation of $X \cap \mathsf{cpre}_1(Z^{n-1} \cup Y^{n,m-1})$ is contained in the computation of $\mathsf{MA}(X^{n,m-1})$. The intersection with X in the algorithm above is included in the definition of $\mathsf{MA}(X)$. Furthermore, every recursive call to `psolB` computes the exact attractor $\mathsf{Attr}_1[G, \mathsf{MA}(X)]$ just as above. And the removal of nodes in `psolB` is equivalent to the inclusion of Z^{n-1} in the computation of $\mathsf{cpre}_1(Z^{n-1} \cup Y^{n,m-1})$. $\square$