Preprocessing parity games with partial Gauss elimination

Vincent Kusters
Eindhoven University of Technology

Abstract We investigate the use of partial Gauss elimination as a preprocessing step to parity game solving. Gauss elimination is a well-known technique for solving Boolean equation systems. We present an $\Theta(V^3)$ time algorithm for exhaustive partial Gauss elimination on parity games. Experimental validation shows that this approach is probably not feasible for speeding up the solving of real-world parity games.

1 Introduction

One approach to verifying concurrent systems is model checking. Given a model of a system and a specification, the *model checking problem* is the problem of deciding if the system satisfies the specification. Such specifications typically include safety properties, which state that nothing bad happens, and liveness properties, which state that good things eventually happen. Temporal logic formulae can be used to describe these specifications.

The modal μ-calculus is one of the most expressive languages to express such properties. The model checking problem on a model and a modal μ-calculus formula is known is to be in NP $\cap$ co-NP [7] and even in UP $\cap$ co-UP [3], so it is still an open problem whether the model checking problem can be solved in polynomial time. All known algorithms run in exponential time. An interesting question, therefore, is whether models can be preprocessed with a polynomial time algorithm to improve the performance of existing solvers.

Existing methods of solving the model checking problem include reductions to the problem of solving a Boolean equation system [7] or to the problem of determining the winners of the nodes in a parity game [9].

In this report, we investigate the use of Gauss elimination, a technique commonly used to solve Boolean equations systems, to preprocess a parity game.

2 Preliminaries

In this section we give an overview of the syntax and semantics of Boolean equation systems and parity games, as well as the relation between Boolean equation systems and parity games.

2.1 Boolean equation systems

A Boolean equation system is a finite sequence of least and greatest fixed point equations, where each right-hand side of an equation is a proposition formula [6]. For a more detailed account on Boolean equation systems than is presented here, we refer to [7].

Definition 1. *A* Boolean equation system (BES) $\mathcal{E}$ *is defined by the following grammar:*

$$\begin{array}{lll} \mathcal{E} & ::= & \varepsilon \mid (\nu X = f)\ \mathcal{E} \mid (\mu X = f)\ \mathcal{E} \\ f, g & ::= & \mathit{true} \mid \mathit{false} \mid X \mid f \wedge g \mid f \vee g \end{array}$$

where ε is the empty BES; $X \in \mathcal{X}$ is a proposition variable; and f, g are proposition formulae.

We only consider Boolean equations systems where every propositional variable occurs in the left-hand side of at most one equation.

The set of *bound* variables of a Boolean equation system $\mathcal{E}$, denoted $\mathsf{bnd}(\mathcal{E})$, is the set of variables that occur on the left-hand side of the equations in $\mathcal{E}$. Similarly, the set of *occurring* variables, denoted $\mathsf{occ}(\mathcal{E})$, is the set of variables that occur on the right-hand side of some equation in $\mathcal{E}$.

We will only consider *closed* Boolean equation systems, i.e., systems $\mathcal{E}$ for which $\mathsf{occ}(\mathcal{E}) \subseteq \mathsf{bnd}(\mathcal{E})$. These systems have unique solutions, as we will see shortly. If a Boolean equation system contains no *true* or *false* constants, and if it does not have an equation whose right-hand side contains both a $\wedge$- and a $\vee$-operator, it is it is said to be in *standard recursive form* [5]. Any Boolean equation system can be transformed to a Boolean equation system in SRF by introducing new variables for subexpressions that violate the SRF property. The *operator* of a propositional variable X in a Boolean equation system in SRF $\mathcal{E}$ is defined by $\mathsf{op}(X) = \Phi$, where $\sigma X = \Phi F$ is the defining equation of X in $\mathcal{E}$.

A Boolean equation system can be partitioned into largest subsequences with the same fixed point sign. Each such largest subsequence is called a *block*. The blocks are numbered from left to right, starting at 0 if the block contains greatest fixpoint equations and starting at 1 otherwise. The *rank* of a variable X, denoted $\mathsf{rank}(X)$, is the number of the block that contains X's defining equation. For example, in the Boolean equation system $(\mu X = f)(\mu Y = g)(\nu Z = h)$, the rank of Z is 2.

The *solution* of a Boolean equation system is characterized as follows [6]:

Definition 2. *Let $\eta : \mathcal{X} \to \mathbb{B}$ be an environment. The* interpretation $[\![f]\!]\eta$ *maps a proposition formula f to true or false:*

$$\begin{array}{ll} [\![X]\!]\eta \overset{def}{=} \eta(X) & \\ [\![\mathit{true}]\!]\eta \overset{def}{=} \mathit{true} & [\![f \wedge g]\!]\eta \overset{def}{=} [\![f]\!]\eta \wedge [\![g]\!]\eta \\ [\![\mathit{false}]\!]\eta \overset{def}{=} \mathit{false} & [\![f \vee g]\!]\eta \overset{def}{=} [\![f]\!]\eta \vee [\![g]\!]\eta \end{array}$$

The solution of a BES*, given an environment η, is inductively defined as follows:*

$$\begin{array}{lcl} [\![\varepsilon]\!]\eta & \overset{def}{=} & \eta \\ [\![(\sigma X = f)\ \mathcal{E}]\!]\eta & \overset{def}{=} & \begin{cases} [\![\mathcal{E}]\!](\eta[X := [\![f]\!]([\![\mathcal{E}]\!]\eta[X := \mathit{false}])]) & \mathit{if}\ \sigma = \mu \\ [\![\mathcal{E}]\!](\eta[X := [\![f]\!]([\![\mathcal{E}]\!]\eta[X := \mathit{true}])]) & \mathit{if}\ \sigma = \nu \end{cases} \end{array}$$

We refer to [7, Section 3.2] for an explanation of this definition. For our purposes, it is sufficient to know that the solution is unique, respects the Boolean equation system in the sense that for each equation, the left-hand side evaluates to the same value as the right-hand side, and that changing the order of the equations may change the solution.

2.2 Parity games

A parity game is played by players *Even* and *Odd* on a total finite directed graph. Solving a parity game amounts to finding the partitioning of the vertices of the graph into a set of vertices won by player Even and a set of vertices won by player Odd. A more detailed description of parity games than we give here can be found in [1].

Definition 3. *A parity game Γ is a four-tuple $(V, E, p, (V_{Even}, V_{Odd}))$, where (V,E) is a directed graph with vertices V and total edge relation E, $p : V \to \mathbb{N}$ is a priority function, and (V_{Even}, V_{Odd}) is a partitioning of V.*

We say that player Even *owns* all vertices in V_{Even} and player Odd owns all vertices in V_{Odd}. We define *owner*$[v]$ to be 0 if the owner of v is Even and 1 otherwise. The *out-neighbours* of a vertex v, or alternatively, the vertices *adjacent* to v are given by $adj[v] = \{v' \mid (v, v') \in E\}$.

Parity games are played as follows. Firstly, the token is placed on a vertex $s \in V$. The token is moved along edges, yielding an infinite list π of vertices the token passes through, which we will refer to as a *play.* If the token is on a vertex v, then the owner of v decides along which edge to pass the token. The winner of s is determined by looking at the *parity* of the lowest priority that occurs infinitely often in π: if it is even, then player Even wins s, and otherwise player Odd wins s.

A *strategy* for a player i is a partial function $\phi : V_i \to V$ which gives for every vertex owned by i, the vertex that i moves the token to next. A path π is *consistent* with a strategy ϕ for player i if and only if for every suffix $[v_k, v_{k+1}, \dots]$ of π where v_k is owned by player i, it holds that $\phi(v_k) = v_{k+1}$.

A strategy ϕ for player i is said to be a *winning strategy* from a vertex v if and only if i is the winner of every path that starts in v and that is consistent with ϕ. It is known from the literature [9] that each vertex in the game is won by exactly one player. This effectively partitions the set of vertices V into a set won by player Even and a set won by player Odd.

Finally, we consider the following useful equivalence relation between parity games:

Definition 4. *Let $\Gamma = (V, E, p, (V_{Even}, V_{Odd}))$ and $\Gamma' = (V', E', p', (V'_{Even}, V'_{Odd}))$. Then Γ and Γ' are* equivalent *if and only if:*

- *$V = V'$, and*
- *$V_{Even} = V'_{Even}$, and*
- *$V_{Odd} = V'_{Odd}$, and*
- *the set of vertices won by player Odd and the set of vertices won by player Even are the same for Γ and Γ'.*

Note that this definition does not require that the priorities of the nodes are the same in both games.

2.3 Correspondence between Boolean equation systems and parity games

Parity games correspond exactly to the fragment of Boolean equation systems in Standard Recursive Form. Recall that any Boolean equation system can be converted to an equivalent Boolean equation system in SRF.

Converting a parity game to a Boolean equation system is straightforward:

Definition 5. *Let $(V, E, p, (V_{Even}, V_{Odd}))$ be a parity game. Construct the corresponding closed BES in SRF by converting every $v \in V$ as follows:*

$$\begin{cases} \sigma X_v = \bigwedge\{X_{v'} \mid (v, v') \in E\} & \text{if } v \in V_{Odd} \\ \sigma X_v = \bigvee\{X_{v'} \mid (v, v') \in E\} & \text{if } v \in V_{Even} \end{cases},$$

where $\sigma = \mu$ if $p(v)$ is odd, and $\sigma = \nu$ otherwise. The ordering of the equations is such that for vertices u and v with $p(u) < p(v)$, it holds that X_u occurs before *X_v in the BES.*

Converting a Boolean equation system to a parity game is more involved. We first define the *dependency graph* of a Boolean equation system. The nodes in this graph are the bound propositional variables and there is an edge from X to Y if Y occurs in X's defining equation.

Definition 6. *Let $\mathcal{E}$ be a Boolean equation system. The* dependency graph *$\mathcal{G}_{\mathcal{E}}$ of $\mathcal{E}$ is defined as (V, E), where:*

- *$V = \mathsf{bnd}(\mathcal{E})$*
- *$(X, Y) \in E$ if and only if there is a $\sigma X = f$ in $\mathcal{E}$ with $Y \in \mathsf{occ}(f)$*

We can now give the translation from Boolean equation systems in SRF to parity games:

Definition 7. *Let $\mathcal{E}$ be a closed Boolean equation system in SRF. Then $\mathcal{E}$ corresponds to the parity game $\Gamma_{\mathcal{E}} = (V, E, p, (V_{Even}, V_{Odd}))$, where*

- *(V, E) is the dependency graph $\mathcal{G}_{\mathcal{E}}$ of $\mathcal{E}$,*
- *$p(X) = \mathsf{rank}(X)$ for all variables $X \in \mathsf{bnd}(E)$,*
- *$V_{Odd} = \{X \mid \mathsf{op}(X) = \bigwedge\}$, so all conjunctive variables are assigned to V_{Odd}, and*
- *$V_{Even} = V \setminus V_{Odd}$, all other equations are assigned to V_{Even}.*

The following theorem gives us the equivalence between the solution of a Boolean equation system and the partitioning of vertices into those won by player Even and those won by player Odd [4]:

Theorem 1. *Player Even has a winning strategy from a vertex X_i in $\Gamma_{\mathcal{E}}$ if and only if $([\![\mathcal{E}]\!]\eta)(X_i) = true$.*

2.4 Solving a Boolean equation system

A well-known method for solving Boolean equation systems is Gauss elimination [7]. Before giving the algorithm, we present two equality-preserving transformations on Boolean equation systems.

Lemma 1. *The* elimination rule*:*

- *Let $\mathcal{E}_1, \mathcal{E}_2$ be Boolean Equation Systems,*
- *$\sigma X = f, \sigma X = f'$ be Boolean Equations, where $f' = f[X := false]$ if $\sigma = \mu$ and $f' = f[X := true]$ if $\sigma = \nu$.*

Then $[\![\mathcal{E}_1(\sigma X = f)\mathcal{E}_2]\!]\theta = [\![\mathcal{E}_1(\sigma X = f')\mathcal{E}_2]\!]\theta$.

The elimination rule states that we are allowed to eliminate X in its defining equation. More precisely, if X is defined as $\mu X = f$, then we are allowed to replace X by *false* in f. Similarly, if X if defined as $\nu X = g$, then we may replace X by *true* in g.

Lemma 2. *The* substitution rule*:*

- *Let $\mathcal{E}_1, \mathcal{E}_2, \mathcal{E}_3$ be Boolean Equation Systems,*
- *let $\sigma X_1 = f$, $\sigma X_1 = f', \sigma_2 X_2 = g$ be Boolean Equations, where $f' = f[X_2 := g]$.*
- *$[\![\mathcal{E}(\sigma_1 X_1 = f)\mathcal{E}_2(\sigma_2 X_2 = g)\mathcal{E}_3]\!]\theta \overset{def}{=} \theta_1$*
- *$[\![\mathcal{E}(\sigma_1 X_1 = f')\mathcal{E}_2(\sigma_2 X_2 = g)\mathcal{E}_3]\!]\theta \overset{def}{=} \theta_2$*

Then $\theta_1 = \theta_2$.

The substitution rule states that we are allowed to substitute X for the right-hand side of X's defining equation in the equations *left* of X.

The following rule is not used by Gauss elimination but will prove to be useful in the next section.

Lemma 3. *The* intra-block reordering rule*:*

- *Let $\mathcal{E}_1, \mathcal{E}_2$ be Boolean Equation Systems,*
- $[\![\mathcal{E}_1(\sigma X_1 = f)(\sigma X_2 = g)\mathcal{E}_2]\!]\theta \overset{def}{=} \theta_1$
- $[\![\mathcal{E}_1(\sigma X_2 = g)(\sigma X_1 = f)\mathcal{E}_2]\!]\theta \overset{def}{=} \theta_2$

Then $\theta_1 = \theta_2$.

The intra-block reordering rule states the order of equations inside a block may be changed freely.

Using these elimination and substitution rules we can now present the Gauss elimination algorithm [7]:

```
Algorithm GAUSS-ELIMINATION($\mathcal{E}$)
Input. A Boolean equation system $(\sigma_1 X_1 = f_1) \ldots (\sigma_n X_n = f_n)$.
Output. The solution for $X_1$.
1.  for $i \leftarrow n$ downto 1 do
2.      if $\sigma_i = \mu$ then $f_i \leftarrow f_i[X_i := false]$
3.                 else $f_i \leftarrow f_i[X_i := true]$
4.      for $j \leftarrow 1$ to $i - 1$ do
5.          $f_j \leftarrow f_j[X_i := f_i]$
```

This procedure can be extended to compute the solution for all variables in the system instead of just the first.

Although this procedure appears to have a running time of $O(V^2)$ at first sight, the intermediate formulae can blow up exponentially, yielding an exponential running time and memory usage [7]. Indeed, the exponential memory blowup is an important argument for using a parity game solver on the corresponding parity game instead of using Gauss elimination.

3 Partial Gauss elimination on parity games

Note that steps in Gauss elimination algorithm do not preserve the SRF property in general: the BES in SRF $\mathcal{E}_1(\mu X = Y \vee Z)(\mu Z = W \wedge U)$ is reduced to $\mathcal{E}_1'(\mu X = Y \vee (W \wedge U))(\mu Z = W \wedge U)$, which is not in SRF. Since parity games correspond to Boolean equation systems in SRF, we must consider restricted versions of the elimination and substitution rules which maintain the SRF property to do our preprocessing.

The first restriction is that the elimination rule may only be applied on X if this does not introduce Boolean constants after simplification. Eliminating X in $(\nu X = X \wedge Y)$ is allowed, as *true* $\wedge\, Y$ simplifies to Y. The second restriction is that a conjunctive equation may not be substituted into a disjunctive equation or vice versa. We formally define what it means to apply the elimination rule and the substitution rule exhaustively under these restrictions.

Definition 8. *We define a Boolean equation system $\mathcal{E}$ in SRF to be* final *if and only if:*

- *we cannot decompose $\mathcal{E}$ (even after intra-block reordering) into $\mathcal{E}_1(\sigma_1 X_1 = \Phi F_1)\mathcal{E}_2(\sigma_2 X_2 = \Phi F_2)\mathcal{E}_3$, for some $\Phi \in \{\bigwedge, \bigvee\}$ and $X_2 \in F_1$, in such a way that $((F_1 \setminus \{X_2\}) \cup F_2) \neq F_1$, and*
- *$\mathcal{E}$ contains no equation $(\sigma X = \Phi F_X)$ where $X \in F_X$ and $|F_X| > 1$.*

The first criterion expresses that no application of the substitution rule (even after intra-block reordering) on $\mathcal{E}$ can add a new proposition variable to the right hand side of a Boolean equation without destroying the SRF property. The second criterion expresses that X should either be the only variable on the right-hand side of its defining equation, or it should be eliminated.

Translating this definition into parity game terms, we obtain the following equivalent definition:

Definition 9. *We define a parity game to be* final *if and only if:*

- *for all nodes $v \in V$ and all $v' \in adj[v]$, where $owner[v] = owner[v']$ and $p(v) \leq p(v')$, it holds that $adj[v] = (adj[v] \setminus \{v'\}) \cup adj[v']$, and*
- *there is no node $v \in V$ such that $v \in adj[v]$ and $|adj[v]| > 1$.*

Our preprocessing algorithm will transform a parity game Γ into an equivalent, final parity game.

3.1 Removing uniform strongly connected components

As we will see later on in this section, our algorithm requires that the input parity game contains no nontrivial strongly connected components that are completely owned by one player and where all priorities are the same.

Definition 10. *Let $\Gamma = (V, E, p, (V_{Even}, V_{Odd}))$ be a parity game. We define $G^i_P = (V, E^i_P)$, where $(u, v) \in E'$ if and only if $(u, v) \in E$ and player i owns u and v and $p(u) \in P$ and $p(v) \in P$.*

Let $S^i_P = (V^*, E^*)$, where $V^* \subseteq V^i_P$ and $E^* \subseteq E^i_P$, be a strongly connected component of G^k_P.

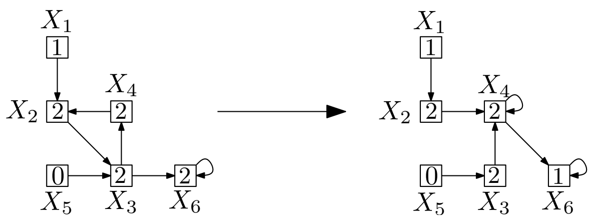

Figure 1: Removal of a nontrivial uniform even strongly connected component.

If i is Even, then we say S_P^i is an even strongly connected component *of* Γ. *If i is Odd, then we say S_P^i is an* odd strongly connected component *of* Γ.

S_P^i is called uniform *if P contains exactly one priority. S_p^i is called* trivial *if it consists of exactly one node, which may or may not have a selfloop.*

Fortunately, we can remove nontrivial even and odd uniform SCCs from a parity game without changing its solution. Given a set of vertices V^*, let $N(V^*)$ be the out-neighbourhood of V^*, i.e., $N(V^*) = \bigcup_{v \in V^*} adj[v] \setminus V^*$. Let $C(V^*)$ be the edges between V^* and $N(V^*)$, i.e., $C(V^*) = E \cap (V^* \times N(V^*))$.

Lemma 4. *Let $\Gamma = (V, E, p, (V_{Even}, V_{Odd}))$ be a parity game and let $G^* = (V^*, E^*)$ be an even or odd uniform strongly connected component of Γ. Choose any $r \in V^*$. Let $E' = ((E \setminus E^*) \setminus C(V^*)) \cup \{(v, r) \mid v \in V^*\} \cup \{(r, v) \mid v \in N(V^*)\}$.*

Then $\Gamma' = (V, E', p, (V_{Even}, V_{Odd}))$ has fewer nontrivial strongly connected components than Γ and is equivalent to Γ.

Proof. Note that this transformation removes G^* from the graph and introduces no new nontrivial SCCs. See Figure 1. In this example, we have $V^* = \{X_2, X_3, X_4\}$ and $E^* = \{(X_2, X_3), (X_3, X_4), (X_4, X_2)\}$ and $C(V^*) = \{X_6\}$. Note that the edges from E^* and $C(V^*)$ are removed, that edges from all nodes in V^* to $r = X_4$ are added and that an edge from r to every vertex in $N(V^*)$ is added.

To show that this transformation does not change the solution of the parity game, we show that whenever a player i has a winning strategy from a node v in Γ, she also has a winning strategy from v in Γ'.

Let (W_{Even}, W_{Odd}) be the partitioning of V into the nodes won by player Even and player Odd. Let ϕ_i be a winning strategy for player i from all nodes in W_i on Γ. Let player j be the player who owns all nodes in V^*. Let p^* denote the priority of the nodes in V^*. We now define a strategy ϕ_i' on Γ for every $v \in V_i$.

If $i \neq j$, then simply $\phi_i' = \phi_i$. Since, as we will see below, the player who wins V^* in Γ still wins V^* in Γ', the strategy ϕ_i' is still winning from all nodes in W_i on Γ'.

If $i = j$, and i and p^* are both even or both odd, then we define ϕ_i' as follows:

$$\phi_i'(v) = \begin{cases} \phi_i(v) & \text{if } v \notin V^* \\ r & \text{if } v \in V^*. \end{cases}$$

Since it is player i who owns the vertices in V^*, and it is also player i who wins if the lowest priority that occurs infinitely often on a path is p^*, it is optimal for player i to keep the token in r. The strategy on the rest of the graph does not have to change.

If $i = j$, and i and p^* are *not* both even or both odd, then we define ϕ_i' as follows:

$$\phi_i'(v) = \begin{cases} \phi_i(v) & \text{if } v \notin V^* \\ r & \text{if } v \in V^* \setminus \{r\} \\ v_e & \text{if } v = r, \end{cases}$$

where v_e is the *escape vertex* of V^*. If there is a $v \in V^*$ such that $\phi_i(v) \in N(V^*) \cap W_i$, then let v_e be this $\phi_i(v)$. Otherwise, $v_e = r$. In other words, if there is some vertex in $v \in V^*$ which moves the token to a vertex in $N(V^*)$ that is won by i, then clearly it is also optimal to set $\phi_i'(r) = \phi_i(v)$. If no such vertex exists, then player i cannot win V^* in Γ, so it does not help to escape the SCC.

We conclude that ϕ_i' is a winning strategy for W_i on Γ' and hence that Γ and Γ' are equivalent. Since Γ' contains strictly fewer nontrivial strongly connected components than Γ, the lemma follows. □

Using Lemma 4, we can give an algorithm to perform this transformation. See SCC-Reduce on the next page. We use the algorithm by Tarjan[8] to compute the strongly connected components of our graph. We assume a modified version of this algorithm Tarjan-Restricted which only traverses an edge (u, v) if $p(u) = p(v)$ and $owner[u] = owner[v]$. The effect of this change is that Tarjan-Restricted returns the even or odd uniform strongly connected components of a parity game.

Theorem 2. SCC-Reduce(Γ) *returns a parity game* Γ' *which is equivalent to* Γ *and contains no nontrivial uniform even or odd strongly connected components in* $O(V + E)$ *time.*

Proof. The correctness follows directly from Lemma 4 and the fact that the transformation done in Lemma 4 does not affect other uniform even or odd strongly connected components in the parity game. Note that we use linked lists to implement the adjacency lists. The running time of Tarjan-Restricted is $O(V + E)$. For each SCC V^* we perform the transformation given in Lemma 4. Lines 6 and 7 take constant time. Lines 8-9 take $O(V^*)$ time. Line 10 takes constant time. Lines 13-15 get executed $|E|$ times over the execution of the entire algorithm. One execution takes constant time, so the total time over the entire algorithm is $O(E)$. Lines 16 and 17 takes constant time. Summarizing, the body of the loop in lines 5-16 takes $O(V^*)$ time, excluding lines 13-15. Since S is a partition of V, the total running time of the algorithm becomes $O(V + E)$. □

Algorithm SCC-REDUCE(Γ)
Input. A parity game Γ.
Output. A parity game without nontrivial uniform even or odd SCCs equivalent to Γ.

```
1.  $(V, E, p, (V_{Even}, V_{Odd})) \leftarrow \Gamma$
2.  $S \leftarrow$ TARJAN-RESTRICTED($V$)
3.  **for each** $v \in V$ **do**
4.      $mark[v] \leftarrow$ **nil**
5.  **for each** $V^* \in S$ **do**
6.      **if** $|V^*| > 1$ **then**
7.          Choose any $v^* \in V^*$.
8.          **for each** $v \in V^*$ **do**
9.              $mark[v] \leftarrow v^*$
10.         $N \leftarrow [v^*]$
11.         **for each** $v \in V^*$ **do**
12.             **for each** $v' \in adj[v]$ **do**
13.                 **if** $mark[v'] \neq v^*$ **then**
14.                     Append $v'$ to $N$.
15.                     $mark[v'] \leftarrow v^*$
16.             $adj[v] \leftarrow [v^*]$
17.         $adj[v^*] \leftarrow N$
```

3.2 Computing a final parity game

We now present the full preprocessing algorithm and proofs of its correctness and running time.

Algorithm PARITY-GAME-FINAL(Γ)
Input. A parity game Γ with no nontrivial uniform even or odd strongly connected components.
Output. A final parity game equivalent to Γ.

```
1.  $(V, E, p, (V_{Even}, V_{Odd})) \leftarrow \Gamma$
2.  **for each** $v \in V$ **do**
3.      $visited[v] \leftarrow$ **false**
4.  **for each** $v \in V$ **do**
5.      **if not** $visited[v]$ **then** DAG-DFS-VISIT(v)
6.  **return** $(V, E, p, (V_{Even}, V_{Odd}))$
```

Algorithm DAG-DFS-Visit(v)

```
1.  $visited(v) \leftarrow$ true
2.  for each $v' \in adj[v]$ do
3.      if $v \neq v'$ and $owner[v] = owner[v']$ and $p(v) \leq p(v')$ and not $visited[v']$ then
4.          DAG-DFS-Visit($v'$)
5.      for each $v' \in adj[v]$ do
6.          if $v \neq v'$ and $owner[v] = owner[v']$ and $p(v) \leq p(v')$ then
7.              $adj[v] \leftarrow (adj[v] \setminus \{v'\}) \cup adj[v']$
8.      if $v \in adj[v]$ then
9.          if $p(v)$ mod $2 = owner[v]$ then
10.             $adj[v] \leftarrow \{v\}$
11.         else if $p(v)$ mod $2 \neq owner[v]$ and $|adj[v]| > 1$
12.             $adj[v] \leftarrow adj[v] \setminus \{v\}$
```

We first show that the output parity game is equivalent to the input parity game. Afterwards, we will show that the output parity game is final.

Lemma 5. Parity-Game-Substitute(Γ) *returns a parity game which is equivalent to* Γ.

Proof. We must show that the assignments in line 7, 10 and 12 of DAG-DFS-Visit do not change the winning sets of $\Gamma = (V, E, p, (V_{Even}, V_{Odd}))$.

Let X and Y be the nodes in line 7. Then a corresponding BES $\mathcal{E}$ of Γ can be written as follows using Definition 5:

$$\mathcal{E}_1 \ (\sigma_1 X = \Phi\left(\{Y\} \cup F_X\right)) \ \mathcal{E}_2 \ (\sigma_2 Y = \Phi F_Y) \ \mathcal{E}_3,$$

for some $\Phi \in \{\bigvee, \bigwedge\}$ and where $Y \notin F_X$.

Note that the constraint that $p(X) \leq p(Y)$ guarantees that we can place X before Y in $\mathcal{E}$. Applying the substitution rule, we obtain the BES $\mathcal{E}'$:

$$\mathcal{E}_1 \ (\sigma_1 X = \Phi\left(F_Y \cup F_X\right)) \ \mathcal{E}_2 \ (\sigma_2 Y = \Phi F_Y) \ \mathcal{E}_3,$$

Note that $\mathcal{E}'$ is a closed BES in SRF and $\mathcal{E}$ and $\mathcal{E}'$ are equivalent. Let $\mathcal{G}_{\mathcal{E}} = (V_1, E_1)$ be the dependency graph of $\mathcal{E}$ and let $\mathcal{G}_{\mathcal{E}'} = (V_2, E_2)$ be the dependency graph of $\mathcal{E}'$. It follows from Definition 6 that $V_2 = V_1$ and $E_2 = (E_1 \setminus \{(X, Y)\}) \cup \{(X, Z) \mid Z \in F_Y\}$.

Since none of the ranks of the variables were changed in the conversion of $\mathcal{E}$ to $\mathcal{E}'$ and no conjunctive equations were changed to disjunctive equations or vice versa, we can now apply Definition 7 on $\mathcal{E}'$ to obtain a parity game $\Gamma' = (V, E', p, (V_{Even}, V_{Odd}))$, where $E' = (E \setminus \{(X, Y)\}) \cup \{(X, Z) \mid Z \in adj[Y]\}$. This is exactly the change which is performed in line 7 of the algorithm.

Since Γ is equivalent to $\mathcal{E}$ and $\mathcal{E}$ is equivalent to $\mathcal{E}'$ and $\mathcal{E}'$ is equivalent to Γ', the substitution performed at line 7 does not affect the solution of the parity game.

In line 10, it holds that vertex v has a selfloop and that its owner and priority are both even or both odd. Consider the case where the owner and priority are both even. Now the corresponding BES $\mathcal{E}$ can be written as follows (writing X instead of v):

$$\mathcal{E}_1\ (\nu X = \bigvee \{X\} \cup F_X)\ \mathcal{E}_2.$$

When we apply the elimination rule, X is replaced by *true*. After Boolean simplification, we obtain $\mathcal{E}_1$ $(\nu X = \mathit{true})$ $\mathcal{E}_2$. Applying the elimination rule in reverse, we obtain $\mathcal{E}_1$ $(\nu X = X)$ $\mathcal{E}_2$, which is exactly the result of the transformation done in line 10. The case where the owner and priority are both odd is symmetrical. We conclude that the modification done at line 10 does not affect the solution of the parity game.

If the owner and priority of v are *not* both even or both odd, then line 12 is executed. Consider the case where the owner is Odd and the priority is even. In that case, we can write the corresponding BES $\mathcal{E}$ as follows:

$$\mathcal{E}_1\ (\nu X = \bigwedge \{X\} \cup F_X),$$

where F_X is nonempty. When we apply the elimination rule, X is replaced by *true*. After Boolean simplification, we obtain $(\nu X = \bigwedge F_X)$, which is exactly the result of the transformation in line 12. The case where the owner is Even and the priority is odd is symmetrical. It follows that the modification done at line 12 does not affect the solution of the parity game.

Since none of the modifications done by the algorithm affect the solution of the parity game, the lemma follows. □

The *constrained graph* of a parity game $\Gamma = (V, E, p, (V_{Even}, V_{Odd}))$ is the graph (V, E') where $(u, v) \in E'$ if and only if $(u, v) \in E$ and $u \neq v$ and $\mathit{owner}[u] = \mathit{owner}[v]$ and $p(u) \leq p(v)$. Note that the graph traversed by Parity-Game-Final(Γ) is a subgraph of the constrained graph of Γ.

The *traversable subgraph* at a vertex v of a parity game Γ is the subgraph of the constrained graph of Γ that is reachable from v. The subgraph visited by a call to DAG-DFS-Visit(v) is a subgraph of the traversable subgraph at v.

Lemma 6. *Let $\Gamma = (V, E, p, (V_{Even}, V_{Odd}))$ be a parity game with no nontrivial uniform even or odd strongly connected components and let $v \in V$. Then the traversable subgraph at v is a directed acyclic graph.*

Proof. Proof by contradiction. Suppose there is a path π in the traversable subgraph at v that contains a cycle, i.e., $\pi = [v_1, \ldots, v_k, v_\ell, \ldots, v_k]$. Note that $v_\ell \neq v_k$, since the constrained graph of Γ does not contain selfloops. Since for each edge (v_i, v_{i+1}) in π we have $p(v_i) \leq p(v_{i+1})$, it must hold that $p(v_k) \leq p(v_\ell) \leq \cdots \leq p(v_k)$. Hence, $p(v_k) = p(v_\ell) = \ldots p(v_k)$. But then $\{v_k, v_\ell, \ldots, v_k\}$ forms a nontrivial uniform even or odd strongly connected component. Contradiction. The lemma follows. □

We say that a vertex v is being *processed* if the execution of DAG-DFS-VISIT(v) has reached line 5.

Lemma 7. *Let (V_t, E_t) be subgraph of the traversable subgraph at v that is traversed during the call* DAG-DFS-VISIT(v)*. Then the order in which vertices are processed is a reverse topological order of (V_t, E_t).*

Proof. By Lemma 6, (V_t, E_t) is a directed acyclic graph. Since a vertex is processed strictly after all its out-neighbours have been processed, the vertices are processed in a reverse topological order of (V_t, E_t). □

Lemma 8. PARITY-GAME-FINAL(Γ) *returns a final parity game for any parity game Γ with no nontrivial uniform even or odd strongly connected components.*

Proof. Let us consider the subgraph (V_t, E_t) traversed by DAG-DFS-VISIT(v). We will show that after completion of DAG-DFS-VISIT(v), the modified subgraph (V_t, E_t) is final.

We define a vertex v to be *final* if v satisfies criterion 2 of Definition 9 and if the subgraph induced by $\{v\} \cup N(\{v\})$ satisfies criterion 1 of Definition 9.

By Lemma 7, the nodes in V_t are processed in a reversed topological order. It follows that the leaves are processed first. For each leaf with selfloops, the algorithm will remove either the selfloop or the edges to the leaf's other neighbours, establishing criterion 2 of Definition 9. The first criterion is trivially satisfied for leaves. It follows that after the algorithm completes, all leaves in V_t are final.

Suppose we start processing a nonleaf vertex v. Since the vertices are processed in reverse topological order, we can assume the children of v are already final. The algorithm now modifies $adj[v]$ in a way that establishes criterion 1 and 2 of Definition 9. Hence, v is also final after the algorithm completes. It follows from induction that the whole subgraph (V_t, E_t) becomes final.

By the definition of traversable subgraphs, the entire parity game will be final after all calls to DAG-DFS-VISIT. □

The following theorem now establishes the correctness and the running time of the algorithm:

Theorem 3. PARITY-GAME-FINAL(Γ) *returns a final parity game equivalent to Γ in $O(V^3)$ time.*

Proof. The correctness of the algorithm follows directly from Lemma 5 and Lemma 8. The running time of the normal DFS algorithm is $O(V + E)$. The only thing our algorithm adds is the time needed to do the substitutions and the eliminations. The eliminations in lines 8-12 of DAG-DFS-VISIT can be done in constant time per vertex.

If a vertex v has $O(V)$ children, each of which also has $O(V)$ children, then updating the adjacency list for v takes $O(V^2 \log V)$ time, assuming we use a balanced tree for

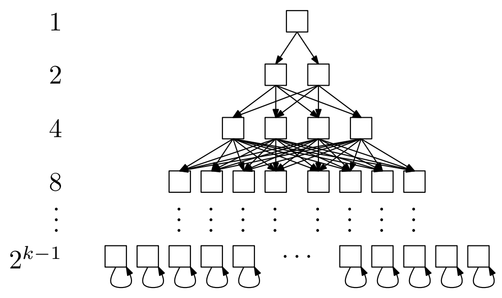

Figure 2: A worst case for Parity-Game-Final.

the adjacency lists. If we use an adjacency matrix instead, doing the updates will take $O(V^2)$ time. Since there are $|V|$ nodes, the total amount of time to do the updates is $O(V^3)$. Since $E = O(V^2)$, the final running time for our algorithm is $O(V^3)$. □

3.3 Worst-case construction

The previous section shows that the running time of the algorithm is $O(V^3)$. In this section, we show that we can always construct a parity game where the algorithm takes $\Omega(V^3)$ time.

Let us consider a tree (V, E) of size n. The tree is constructed from several levels $V = \bigcup\{\mathcal{L}_0, \ldots, \mathcal{L}_{k-1}\}$, where $2^k - 1 = n$ and each $\mathcal{L}_i$ contains 2^i nodes. The edge relation is constructed by adding all edges from a level $\mathcal{L}_i$ to the next level $\mathcal{L}_{i+1}$. In the last level, every vertex has only a selfloop. See Figure 2.

The algorithm first processes the vertices in $\mathcal{L}_{k-1}$, changing nothing. It then moves up to $\mathcal{L}_{k-2}$ and again changes nothing. Next, it proceeds to the vertices in $\mathcal{L}_{k-3}$. Each of these vertices has 2^{k-2} out-neighbours, and each of these out-neighbours have 2^{k-1} out-neighbours. Performing the substitutions therefore takes $2^{k-2} \cdot 2^{k-1}$ operations for each of the 2^{k-3} vertices in $\mathcal{L}_{k-3}$. After the substitutions, each vertex in $\mathcal{L}_{k-3}$ has an edge to each of the vertices in $\mathcal{L}_{k-1}$.

In general, the algorithm performs $2^i \cdot 2^{i+1} \cdot 2^{k-1}$ operations to do the substitutions for level i. This yields the following total number of operations for the entire algorithm:

$$\sum_{i=0}^{k-3} 2^i \cdot 2^{i+1} \cdot 2^{k-1} = \Theta(2^{3k}) = \Theta(n^3).$$

Combined with Theorem 3, this result gives us the following theorem:

Theorem 4. Parity-Game-Final(Γ) *returns a final parity game equivalent to* Γ *in* $\Theta(V^3)$ *time.*

4 Experimental analysis

We did a small experimental study on the effectiveness of the algorithm. We tested the algorithm on some of the parity games used in the experimental study of [1]. The parity game solver algorithm used is the recursive algorithm [9] as implemented in the PGSolver toolkit [2]. The `pgsolver` tool was run with all optimizations disabled.

The running times of each experiment are listed in Appendix 1. It should be noted that the running times which are very close to zero suffer strongly from measurement errors. They were included for the sake of completeness.

A direct implementation of the Parity-Game-Final algorithm from the previous section yields disappointing results. Not only does executing the preprocessor take longer than executing the solver directly, the solver is also often *slower* instead of faster on the result of the preprocessor. See Table 1 for details. An important reason for this is that the preprocessor causes a blow-up of the size of some of the parity games. As we can see in Table 3, the size of the parity game after preprocessing can be over 100 times as large as the size of the original parity game.

One way to reduce the blow-up is to make the following modification to the algorithm: *if a vertex $u \in V$ is currently being processed, and u has exactly one out-neighbour v, then do not substitute v into u.* In the parity games we considered, there is a substantial amount of vertices with a single out-neighbour. Also note that the removal of a strongly connected component S introduces $|S| - 1$ vertices with a single out-neighbour. The running times of the modified algorithm are listed in Table 2. Although this modification improves the running times of the preprocessor, as well as the running times of the solver, it is still faster to run the solver directly.

Another potential improvement is to apply the elimination rule immediately after a strongly connected component has been removed. If it is beneficial for the owner of the strongly connected component to keep the token in the strongly connected component, then an application of the elimination rule will remove all edges to the neighbourhood of the strongly connected component. Unfortunately, experiments with this modification reveal that it little effect on the running times.

5 Conclusions

We have presented a $\Theta(V^3)$ algorithm for preprocessing parity games with partial Gauss elimination. Experiments show that the algorithm in its current form is not a viable preprocessing step for parity game solvers.

A reader interested in evaluating variations of the algorithm may consider applying the substitution rule in reverse where possible. This may reduce the number of edges in a parity game, which may speed up some parity game solvers.

References

[1] Sjoerd Cranen, Jeroen J.A. Keiren, and Tim A.C. Willemse. “Stuttering Mostly Speeds Up Solving Parity Games”. In: (2011). Submitted for publication.

[2] O. Friedmann and M. Lange. *The PGSolver collection of parity game solvers.* Tech. rep. Institut für Informatik, Ludwig-Maximilians-Universität München, Germany, 2008.

[3] M. Jurdziński. “Deciding the Winner in Parity Games is in UP $\cap$ co-UP”. In: *Inf. Process. Lett.* 68.3 (1998), pp. 119–124.

[4] M.K. Keinänen. “Solving Boolean Equation Systems”. PhD thesis. Helsinki University of Technology, 2006.

[5] J. J. A. Keiren and T. A. C. Willemse. “Bisimulation Minimisations for Boolean Equation Systems”. In: *Proceedings HVC’09, Haifa, Israel.* Lecture Notes in Computer Science. Oct. 2009.

[6] J.J.A. Keiren, M.A. Reniers, and T.A.C. Willemse. “Structural Analysis of Boolean Equation Systems”. In: *ACM Transactions on Computational Logic* (2010). Accepted for publication.

[7] AH Mader. “Verification of modal properties using boolean equation systems”. In: (1997).

[8] R. Tarjan. “Depth-first search and linear graph algorithms”. In: *Conference Record 1971 Twelfth Annual Symposium on Switching and Automata Theory.* IEEE. 1971, pp. 114–121.

[9] W. Zielonka. “Infinite games on finitely coloured graphs with applications to automata on infinite trees”. In: *Theoretical Computer Science* 200.1-2 (1998), pp. 135–183.

1 Appendix: experimental results

file name	solve time	prep time	solve after prep time
`1394-fin.1394_property2.bisim.gm`	0.05s	0.07s	0.07s
`1394-fin.1394_property2.gm`	4.73s	8.67s	12.82s
`1394-fin.1394_property5.gm`	6.82s	13.14s	17.03s
`german_linear.german_consistency.gm`	0.39s	56.44s	57.71s
`hes98_spec1.hes98_impl3.eq1.bisim.gm`	0.02s	0.06s	0.07s
`swp_lists2_2.infinitely_often_enabled_then_infinitely_often_taken.gm`	0.36s	14.42s	27.39s
`swp_lists2_2.read_then_eventually_send_if_fair.gm`	0.43s	43.76s	20m 51s
`swp_lists2_3.infinitely_often_lost.gm`	6.12s	>10m	-
`swp_lists2_4.no_generation_of_messages.gm`	4.28s	1m 51s	2.48s
`swp_lists3_2.infinitely_often_receive_d1.gm`	0.11s	1.77s	1.80s
`swp_lists2_4.read_then_eventually_send_if_fair.gm`	2.41s	>10m	-
`swp_lists3_2.infinitely_often_receive_for_all_d.gm`	0.24s	3.98s	6.64s
`swp_lists3_2.nodeadlock.gm`	0.09s	0.13s	0.04s
`swp_lists3_2.read_then_eventually_send.gm`	6.78s	>10m	-
`swp_lists3_2.read_then_eventually_send_if_fair.gm`	1.29s	6m 35s	>10m
`swp_lists3_3.infinitely_often_enabled_then_infinitely_often_taken.gm`	3.45s	>10m	-
`swp_lists3_3.nodeadlock.gm`	0.62s	1.49s	0.15s
`swp_lists3_3.no_duplication_of_messages.bisim.gm`	0.03s	0.03s	0.04s
`swp_lists3_3.read_then_eventually_send_if_fair.bisim.gm`	0.03s	0.04s	0.04s

Table 1: The speed of the algorithm with and without preprocessing. The time required by the parity game solver on the original parity game is listed in the “solve time”. The time needed to do the preprocessing is listed in the “prep time” column. The time the solver requires to solve the preprocessed parity game is listed in the “solve after prep time” column. An experiment was aborted when a algorithm took more than 10 minutes to execute.

filename	solve time	prep time	solve after prep time
1394-fin.1394_property2.bisim.gm	0.05s	0.06s	0.09s
1394-fin.1394_property2.gm	4.73s	9.89s	10.97s
1394-fin.1394_property5.gm	6.82s	16.63s	16.42s
german_linear.german_consistency.gm	0.39s	9.12s	0.33s
hes98_spec1.hes98_impl3.eq1.bisim.gm	0.02s	0.04s	0.01s
swp_lists2_2.infinitely_often_enabled_then_infinitely_often_taken.gm	0.36s	0.75s	0.25s
swp_lists2_2.read_then_eventually_send_if_fair.gm	0.43s	1.49s	0.56s
swp_lists2_3.infinitely_often_lost.gm	6.12s	29.33s	2.91s
swp_lists2_4.no_generation_of_messages.gm	4.28s	1m 51s	3.88s
swp_lists3_2.infinitely_often_receive_d1.gm	0.11s	0.19s	0.10s
swp_lists2_4.read_then_eventually_send_if_fair.gm	2.41s	34.04s	8.97s
swp_lists3_2.infinitely_often_receive_for_all_d.gm	0.24s	0.43s	0.16s
swp_lists3_2.nodeadlock.gm	0.09s	0.12s	0.05s
swp_lists3_2.read_then_eventually_send.gm	6.78s	41.20s	4.66s
swp_lists3_2.read_then_eventually_send_if_fair.gm	1.29s	8.55s	2.17s
swp_lists3_3.infinitely_often_enabled_then_infinitely_often_taken.gm	3.45s	8.41s	1.60s
swp_lists3_3.nodeadlock.gm	0.62s	1.53s	0.15s
swp_lists3_3.no_duplication_of_messages.bisim.gm	0.03s	0.05s	0.03s
swp_lists3_3.read_then_eventually_send_if_fair.bisim.gm	0.03s	0.03s	0.04s

Table 2: The speed of the algorithm with and without preprocessing using the exception for vertices with a single out-neighbour. The column names are the same as in Table 1

filename	original size	prep size	modified prep size
`1394-fin.1394_property2.bisim.gm`	39K	83K	83K
`1394-fin.1394_property2.gm`	3.8M	22M	21M
`1394-fin.1394_property5.gm`	7.8M	46M	46M
`german_linear.german_consistency.gm`	1.1M	162M	565K
`hes98_spec1.hes98_impl3.eq1.bisim.gm`	205	214	213
`swp_lists2_2.infinitely_often_enabled_then_infinitely_often_taken.gm`	430K	67M	264K
`swp_lists2_2.read_then_eventually_send_if_fair.gm`	518K	164M	404K
`swp_lists2_3.infinitely_often_lost.gm`	7.9M	-	4.3M
`swp_lists2_4.no_generation_of_messages.gm`	8.4M	3.4M	3.4M
`swp_lists2_4.read_then_eventually_send_if_fair.gm`	3.6M	-	3.2M
`swp_lists3_2.infinitely_often_receive_d1.gm`	70K	6.0M	38K
`swp_lists3_2.infinitely_often_receive_for_all_d.gm`	216K	19M	121K
`swp_lists3_2.nodeadlock.gm`	54K	25K	25K
`swp_lists3_2.read_then_eventually_send.gm`	9.8M	-	5.0M
`swp_lists3_2.read_then_eventually_send_if_fair.gm`	1.6M	1.4G	1.4M
`swp_lists3_3.infinitely_often_enabled_then_infinitely_often_taken.gm`	4.3M	-	2.5M
`swp_lists3_3.nodeadlock.gm`	347K	145K	145K
`swp_lists3_3.no_duplication_of_messages.bisim.gm`	73	66	66
`swp_lists3_3.read_then_eventually_send_if_fair.bisim.gm`	128	123	121

Table 3: The original file size of the parity games, alongside the size after preprocessing (with and without the exception for vertices with a single out-neighbour).