Local Strategy Improvement for Parity Game Solving

Oliver Friedmann

Dept. of Computer Science
University of Munich, Germany

Martin Lange

Dept. of Computer Science
University of Kassel, Germany

The problem of solving a parity game is at the core of many problems in model checking, satisfiability checking and program synthesis. Some of the best algorithms for solving parity game are strategy improvement algorithms. These are global in nature since they require the entire parity game to be present at the beginning. This is a distinct disadvantage because in many applications one only needs to know which winning region a particular node belongs to, and a witnessing winning strategy may cover only a fractional part of the entire game graph.

We present a local strategy improvement algorithm which explores the game graph on-the-fly whilst performing the improvement steps. We also compare it empirically with existing global strategy improvement algorithms and the currently only other local algorithm for solving parity games. It turns out that local strategy improvement can outperform these others by several orders of magnitude.

1   Introduction

Parity games are two-player games of perfect information played on labelled, directed graphs. They are at the core of various problems in computer-aided verification, namely model checking [12], satisfiability checking for branching-time logics [5] and controller synthesis [1]. They are also closely related to other games of infinite duration, in particular mean and discounted payoff as well as stochastic games [6, 12].

A variety of algorithms for solving parity games has been invented so far. Among these, strategy improvement (SI) algorithms have proved to be very successful both because of their theoretical elegance as well as their good performance in practice. The name hints at the general method according to which these algorithms work. Starting with an initial strategy, they compute an evaluation which either witnesses that the current strategy is winning, or it offers ways to improve the current strategy. SI is therefore in fact an algorithm scheme parametrised by a policy which determines how to improve a non-winning strategy. The first SI algorithm was given by Jurdziński and Vöge [13] which was inspired by and improves over Puri's SI algorithms for parity and discounted payoff games [9]. Later, Schewe presented another SI algorithm for parity games and payoff games [10].

A disadvantage common to all these algorithms is their global nature: they require the entire game graph to be present at the start of the algorithm's execution. They then compute for a given player his entire winning region and a corresponding winning strategy. This is a drawback because in many applications of parity games taken from program verification one only needs to know for a specific node in the game which winning region it belongs to, together with a winning strategy witnessing this.

As an example, consider the model checking problem for branching-time temporal logics like CTL, CTL*, or the modal μ-calculus on finite-state systems for instance. These can easily be regarded as instances of the parity game solving problem [12, 8] in which the nodes are pairs of transition system states and (sets of) subformulas of the input formula. Solving the model checking problem means solving the parity game locally, i.e. deciding which player wins the node composed of the transition system's initial state and the input formula. A global algorithm would compute this information for every (reachable)

pair of nodes and (set of) subformulas. In model checking, local solving is usually considered to be more desirable for explicitly represented state spaces [2]. Similarly, the satisfiability problem for branching-time logics reduces to the problem of solving parity games [5] and, again, in order to decide whether or not the input formula is satisfiable one needs to know whether or not player 0 wins a particular node of the resulting parity game. The winners of the other nodes are irrelevant.

It is known that the problem of solving a parity game is equivalent under linear-time reductions to the model checking problem for the modal μ-calculus. There are local model checkers, in particular the one by Stevens and Stirling [11] which can therefore be rephrased as a parity game solver. It is generally outperformed in practice by the global algorithms – when the task is to solve an entire game – in particular the SI algorithms [4]. It is therefore fair to say that local parity game solvers with reasonable efficiency in practice are desirable but do not exist yet.

Here we present a local SI algorithm scheme for solving parity games. It solves a parity game locally, i.e. starting with a given initial node it expands the game graph in an on-the-fly fashion and intertwines this expansion with evaluations needed in order to improve current strategies. It terminates as soon as it has determined which winning region the initial node belongs to.

The rest of the paper is organised as follows. Sect. 2 recalls parity games and global SI. Sect. 3 presents local SI. Sect. 4 reports on empirical comparisons between local SI, global SI, and the local model checking algorithm by Stevens/Stirling when rephrased as a parity game solver. We show that local SI can outperform the others significantly.

The contribution of this paper therefore lies in the practical use of parity game solvers. The presented algorithm is not meant to improve other algorithms in asymptotic worst-case complexity. Its relevance is justified by the improvement in solving parity games in practice which this algorithm can bring along.

2   Preliminaries

2.1   Parity Games

A *parity game* is a tuple $G = (V, V_0, V_1, E, \Omega)$ where (V, E) forms a directed graph in which each node has at least one successor. The set of nodes is partitioned into $V = V_0 \cup V_1$ with $V_0 \cap V_1 = \emptyset$, and $\Omega : V \to \mathbb{N}$ is the *priority function* that assigns to each node a natural number called the *priority* of the node. We write $|\Omega|$ for the index of the parity game, that is the number of different priorities assigned to its nodes. The graph is not required to be total. Here we only consider games based on finite graphs. W.l.o.g. we will assume that games are *turn-based*, i.e. for all $v \in V$ and all $i \in \{0, 1\}$ we have: if $v \in V_i$ then $vE \subseteq V_{1-i}$.

We also use infix notation vEw instead of $(v, w) \in E$ and define the set of all *successors* of v as $vE := \{w \mid vEw\}$, as well as the set of all *predecessors* of w as $Ew := \{v \mid vEw\}$.

The game is played between two players called 0 and 1 in the following way. Starting in a node $v_0 \in V$ they construct a path through the graph as follows. If the construction so far has yielded a finite sequence $v_0 \ldots v_n$ and $v_n \in V_i$ then player i selects a $w \in v_n E$ and the play continues with the sequence $v_0 \ldots v_n w$. Otherwise, the play ends in a sink v_n.

Every infinite play has a unique winner given by the *parity* of the greatest priority that occurs infinitely often in a play. The winner of the play $v_0 v_1 v_2 \ldots$ is player i iff $\max\{p \mid \forall j \in \mathbb{N} \, \exists k \geq j : \Omega(v_k) = p\} \equiv_2 i$ (where $i \equiv_2 j$ holds iff $|i - j| \mod 2 = 0$). That is, player 0 tries to make an even priority occur infinitely often without any greater odd priorities occurring infinitely often, player 1 attempts the converse. For finite plays $v_0 v_1 v_2 \ldots v_n$ that end in a sink, the winner is given by the *chooser* of v_n. Player i wins the play iff $v_n \in V_{1-i}$.

A *positional strategy* for player i in G is a — possibly partial — function $\sigma : V_i \rightarrow V$. A play $v_0 v_1 \ldots$ *conforms* to a strategy σ for player i if for all $j \in \mathbb{N}$ we have: if $v_j \in V_i$ then $v_{j+1} = \sigma(v_j)$. Intuitively, conforming to a strategy means to always make those choices that are prescribed by the strategy. A strategy σ for player i is a *winning strategy* in node v if player i wins every play that begins in v and conforms to σ. We say that player i *wins* the game G starting in v iff he/she has a winning strategy for G starting in v. Let $\bot$ denote the empty strategy.

With G we associate two sets $W_0, W_1 \subseteq V$; W_i is the set of all nodes v s.t. player i wins the game G starting in v. Here we restrict ourselves to positional strategies because it is well-known that a player has a (general) winning strategy iff she has a positional winning strategy for a given game. In fact, parity games enjoy positional determinacy meaning that for every node v in the game either $v \in W_0$ or $v \in W_1$ [3]. Furthermore, it is not difficult to show that, whenever player i has winning strategies σ_v for all $v \in U$ for some $U \subseteq V$, then there is also a single strategy σ that is winning for player i from every node in U.

The problem of *(globally) solving a parity game* is to decide for every node $v \in V$ whether ot not it belongs to W_0. The problem of *locally solving a parity game* is to decide, for a given node v, whether or not v belongs to W_0. Note that it may not be necessary to visit all nodes of a game in order to answer this question. Consider the following trivial example. The two nodes under consideration are connected to each other and have no further outgoing edges. Hence, this simple cycle is clearly won by one of the two players. A depth-first search can find this loop whilst leaving the rest of the game unexplored.

A strategy σ for player i induces a *strategy subgame* $G|_\sigma := (V, V_0, V_1, E|_\sigma, \Omega)$ where $E|_\sigma := \{(u,v) \in E \mid u \in V_i \Rightarrow (u \in dom(\sigma) \wedge \sigma(u) = v)\}$. The set of strategies for player i is denoted by $\mathscr{S}_i(G)$.

Let $U \subseteq V$, $i \in \{0,1\}$ and σ be an i-winning strategy on U. The i-attractor of U is the least set W s.t. $U \subseteq W$ and whenever $v \in V_i$ and $vE \cap W \neq \emptyset$, or $v \in V_{1-i}$ and $vE \subseteq W$ then $v \in W$. Hence, the i-attractor of U contains all nodes from which player i can move "towards" U and player $1 - i$ must move "towards" U. At the same time, it is possible to construct an *attractor strategy* which is a positional strategy in a reachability game. Following this strategy guarantees player i to reach a node in U eventually, regardless of the opponent's choices. Let $\text{ATTR}_{G,i}(U, \sigma) = (W, \tau)$ denote the i-attractor of U s.t. τ is an i-winning strategy on W. Note that the i-attractor can be computed in $\mathscr{O}(|W \cup \bigcup_{v \in W} Ev|)$.

An important property that has been noted before [14, 12] is that removing the i-attractor of any set of nodes from a game will still result in a total game graph.

2.2 Global Strategy Improvement

We recap global SI [13, 10]. It defines a valuation of strategies and a partial order on them. The algorithm then starts with an arbitrary strategy, computes its valuation, improves it accordingly and iterates this until no improvement is possible anymore, which means that winning regions and strategies have been found.

Strategy valuations can be defined in different ways, but usually through *node valuations* for each node v. The latter are totally ordered and induced by a path starting in v leading to a cycle. The valuation process then selects for each node v the node valuation associated with a path π ending in a cycle starting in v conforming to the strategy σ that yields the worst valuation.

Schewe [10] uses a simple valuation by considering an *escape sink* transformation of the given parity game: all priorities are increased by 2, two new self-cycle nodes with priorities 0 and 1 are added to the graph and each player 0 node is connected to the 1 and each player 1 node is connected to the 0. The transformed game is equivalent to the original game, but now the only occurring cycle nodes in valuations are 0 and 1 (assuming that one starts with a reasonable initial strategy). Schewe's node valuation therefore is either the set of nodes leading to the unprofitable self-cycle or ∞, indicating that the node is already won by the σ-player following σ.

We follow this approach, but instead of adding additional self-cycles, we handle partial strategies σ: whenever the strategy is undefined for a certain node v, we identify this with the decision to move to the imaginary self-cycle.

Let G be a parity game. The *relevance ordering* $<$ on V is induced by Ω: $v < u : \iff \Omega(v) < \Omega(u)$; we extend $<$ w.l.o.g. to an arbitrary but fixed total ordering. Additionally one defines the total *reward ordering w.r.t. player q* $\prec_q$ on V: $v \prec_q u$ holds iff one of the following conditions applies.

1. either $\Omega(u) \equiv_2 q$ and $v < u$

2. or $\Omega(u) \equiv_2 q$ and $\Omega(v) \not\equiv_2 q$

3. or $\Omega(u) \not\equiv_2 q$, $\Omega(v) \not\equiv_2 q$ and $u < v$

The length of a loopless path π is denoted by $|\pi|$. The set of loopless paths π in G starting in v is denoted by $\Pi_G(v)$.

The set of *node valuations w.r.t. G*, $\mathcal{V}_G$, is defined as $\mathcal{V}_G := \{\infty\} \cup 2^V$, i.e. a node valuation is either ∞ or a set of nodes. Given a player q, we extend the reward ordering to a total ordering on node valuations. Let $M, N \subseteq V$ with $M \neq N$.

$$M \prec_q N : \iff \begin{cases} (max_<(M \triangle N) \in N \text{ and } max_<(M \triangle N) \equiv_2 q) \text{ or} \\ (max_<(M \triangle N) \in M \text{ and } max_<(M \triangle N) \not\equiv_2 q) \end{cases}$$

where $M \triangle N$ denotes the symmetric difference of both sets. Additionally $M \prec_q \infty$ holds for every $M \subseteq V$.

A loopless non-empty path $\pi = v_0 \ldots v_{k-1}$ induces a *q-node valuation for the node v_0* as follows.

$$\vartheta_q(\pi) := \begin{cases} \infty & \text{if } v_{k-1} \in V_{1-q} \\ \{v_i \mid 0 \leq i < k\} & \text{otherwise} \end{cases}$$

Let ϑ be a node valuation and v be a node. We define an addition operation $\vartheta \oplus v$ as follows.

$$\vartheta \oplus v := \begin{cases} \infty & \text{if } \vartheta = \infty \text{ or } v \in \vartheta \\ \vartheta \cup \{v\} & \text{otherwise} \end{cases}$$

A *game valuation* is a map $\Xi : V \to \mathcal{V}_G$ assigning to each $v \in V$ a node valuation. A partial ordering on game valuations is defined as follows:

$$\Xi \lhd_q \Xi' : \iff \left(\Xi(v) \preceq_q \Xi'(v) \text{ for all } v \in V\right) \text{ and } \left(\Xi \neq \Xi'\right)$$

A strategy σ of player q therefore can be evaluated as follows:

$$\Xi_{q,\sigma} : v \mapsto \min_{\prec_q}\{\vartheta_q(\pi) \mid \pi \in \Pi_{G|_\sigma}(v)\}$$

A game valuation Ξ induces a total counter-strategy τ_Ξ of player $1-q$ by selecting the least profitable strategy decision with respect to Ξ:

$$\tau_\Xi : v \in V_{1-q} \mapsto \min_{\prec_q} U_\Xi(v)$$

where $U_\Xi(v) = \{u \in vE \mid \forall w \in vE : \Xi(u) \preceq_q \Xi(w)\}$.

If Ξ originates from a strategy σ, τ_Ξ can be seen as the best counter-strategy against σ; we also write τ_σ for τ_{Ξ_σ}.

Given a game valuation Ξ and a strategy σ for player q, the *set of Ξ-consistent nodes*, $\mathrm{Con}_{G,q}(\sigma,\Xi)$ is defined as follows.

$$
\begin{aligned}
\mathrm{Con}_{G,q}(\sigma,\Xi) \; := \; & \{v \in V_0 \cap \mathrm{dom}(\sigma) \mid \Xi(v) = \Xi(\sigma(v)) \oplus v\} \cup \\
& \{v \in V_0 \setminus \mathrm{dom}(\sigma) \mid \Xi(v) = \{v\}\} \cup \\
& \{v \in V_1 \mid \Xi(v) = \min_{\prec_q}\{\Xi(u) \oplus v \mid u \in vE\}\}
\end{aligned}
$$

Note that for every strategy σ it holds that $\mathrm{Con}_{G,q}(\sigma,\Xi_\sigma) = V$.

A valuation Ξ originating from a strategy σ can be used to create a new strategy of player q. The strategy improvement algorithm only allows to select new strategy decisions for player q occurring in the *improvement arena* $\mathscr{A}_{G,q,\sigma} := (V, V_0, V_1, E', \Omega)$ where

$$vE'u : \iff vEu \text{ and } (v \in V_{1-q} \text{ or } (v \in V_q \text{ and } \Xi(\sigma(v)) \preceq_q \Xi(u)))$$

Thus all edges performing worse than the current strategy are removed from the game. A player q strategy σ is *q-improvable* iff there is a node $v \in V_q$, a node $u \in V$ with vEu and $\sigma(v) \neq u$ s.t. $\Xi(\sigma(v)) \prec_q \Xi(u)$. The *set of improvable nodes* w.r.t. a valuation Ξ is defined as follows.

$$\mathrm{Imp}_{G,q}(\Xi) := \{v \in V_0 \mid \Xi(v) < \max_{\prec_q}(\{\{v\}\} \cup \{\Xi(u) \oplus v \mid u \in vE\})$$

An *improvement policy* now selects a strategy for player q in a given improvement arena w.r.t. a valuation originating from a strategy. More formally: an improvement policy is a map $\mathscr{I}_{G,q} : \mathscr{S}_q(G) \to \mathscr{S}_q(G)$ fulfilling the following two conditions for every player q strategy σ.

1. For every node $v \in V_q$ it holds that $(v, \mathscr{I}_{G,q}(\sigma)(v))$ is an edge in $\mathscr{A}_{G,q,\sigma}$.

2. If σ is q-improvable then there is a node $v \in V_q$ s.t. $\Xi_\sigma(\sigma(v)) \prec_q \Xi_\sigma(\mathscr{I}_G(\sigma)(v))$.

Updating a strategy according to an improvement policy can only result in strategies with better valuations (w.r.t. $\lhd_q$) [13]. If a strategy σ is not q-improvable, the strategy iteration comes to an end. Winning regions and strategies have then been found in the following sense. $W_q = \{v \mid \Xi_{G,q,\sigma}(v) = \infty\}$; $W_{1-q} = \{v \mid \Xi_{G,q,\sigma}(v) \neq \infty\}$; σ is a winning strategy for player q on W_q; $\tau := \tau_\sigma$ is a winning strategy for player $1-q$ on W_{1-q}.

SI w.r.t. player q starts with the empty initial strategy ι_G and updates it according to a given improvement policy $\mathscr{I}_G$ until it becomes non-improvabable.

Algorithm 1 Strategy Iteration

1: $\sigma \leftarrow \iota_G$
2: **while** σ is q-improvable **do**
3:      $\sigma \leftarrow \mathscr{I}_G(\sigma)$
4: **end while**
5: **return** W_q, W_{1-q}, σ and some τ as described above

---

3   A Local Strategy Improvement Algorithm

Note that global SI is formulated w.r.t. one player q: starting with a strategy for player q it iteratively improves it until it becomes a winning strategy on some part of the game graph. A winning strategy for

the opponent on the complementary part is then derived from this. This is counterproductive for local solving: if local SI was done for one player only, and the opponent is the winner of the designated input node, then the algorithm would have to expand the entire reachable part of the game graph and it would therefore be at most as good as global SI. Consequently, local SI is parametrised by an improvement policy.

We fix a parity game $G = (V, V_0, V_1, E, \Omega)$ and an initial position $v^* \in V$ for which the winner and an associated winning strategy is to be computed. The algorithm maintains two sets W_0 and W_1 in which it collects the winning nodes for players 0 and 1, resp., as well as two winning strategies ρ_0 and ρ_1 on these sets. Initially, all of them are empty.

Local SI is carried out for both players alternatingly on potentially different subgraphs of G. Although it would be possible to run the strategy iteration only for one player, say 0, it would require the algorithm to expand the complete graph that is reachable from v^* if player 1 wins the initial position v^*. Therefore, the algorithm basically alternates the strategy iteration w.r.t. both players indepedently.

For each player q, it maintains initially empty node sets U_q — called the q-subgraph — and E_q initialised to $\{v^*\}$ — called the q-expansion set — with the following properties. The q-subgraph represents a local view onto the game G for player q. The q-expansion set contains nodes which lie just outside of the q-subgraph and which can be used in order to expand the latter. These components satisfy the following invariants.

1. *Subgraph-Closure* (SC): for every $v \in U_q \cap V_{1-q}$ it holds that $vE \setminus (W_0 \cup W_1) \subseteq U_q$; this ensures that for every opponent node in the q-subgraph all possible choices are also included in the set.

2. *Winning-Closure* (WC): $\text{ATTR}_{G|_{U_0 \cup U_1}, q}(W_q, \rho_q) = (W_q, \rho_q)$; this ensures that the set of known winning nodes is closed under attractor strategies w.r.t. the known subgraphs.

3. *Winning-Exclusion* (WE): $U_q \cap (W_0 \cup W_1) = \emptyset$; this ensures that both winning sets cannot be reached by a trivial strategy by one of the two players.

4. *Border-Expansion* (BE): $\{u \in vE \setminus (W_0 \cup W_1 \cup U_q) \mid v \in U_q\} \subseteq E_q \subseteq V \setminus (W_0 \cup W_1 \cup U_q)$; this ensures that all nodes which are not in the q-subgraph but can be reached from it via one edge are in E_q.

Additionally, for each player q we maintain a set I_q — called the q-improvement set —, a valuation Ξ_q for $G|_{U_q}$, a q-strategy σ_q and another set C_q — called the q-change set. They satisfy the following invariants.

1. *Subgame-Strategy* (SS): $\text{dom}(\sigma_q) \cup \text{ran}(\sigma_q) \subseteq U_q$; this ensures that the strategy associated with the subgame is not leading out of the subgame itself.

2. *Valuation-Consistency* (VC): $U_q \setminus \text{Con}_{G,q}(\sigma_q, \Xi_q) \subseteq C_q \subseteq U_q$; this ensures that the q-change set contains at least all these nodes of the q-subgraph that have an inconsistent valuation.

3. *Improvement-Covering* (IC): $I_q \setminus C_q = \text{Imp}_{G,q}(\Xi_q) \setminus C_q$; this ensures that the q-improvement set contains all improvement nodes of the q-subgraph disregarding the q-change set.

Initially, I_q, C_q as well as Ξ_q and σ_q are empty. Finally, the algorithm maintains a *current player* q^* initialised arbitrarily.

It is parametrized by the following policy that is allowed to access all data that is maintained by the algorithm.

1. *Improvement Policy*: a function IMPRPOL that returns a non-empty set of pairs (v, u) s.t. $v \in U_{q^*} \cap V_{q^*}$, $u \in U_{q^*}$ and (v, u) is a proper improvement edge w.r.t. Ξ_{q^*}.

Whenever the algorithm encounters a new set of winning nodes, it calls the function WINNING with parameters q, W and ρ where q is the winning player, W is the new set of nodes won by q and ρ is an

---

Algorithm 2 Update of Winning Set

1: **procedure** WINNING(q, W, ρ)
2:     $(W', \rho') \leftarrow \text{ATTR}_{G|_{U_0 \cup U_1}, q}(W, \rho)$
3:     $W_q \leftarrow W_q \cup W'$
4:     $\rho_q \leftarrow \rho_q \cup \rho'$
5:     $B \leftarrow \bigcup_{v \in W'} Ev$
6:     $(I_0, I_1) \leftarrow (I_0 \setminus W', I_1 \setminus W')$
7:     $(E_0, E_1) \leftarrow (E_0 \setminus W', E_1 \setminus W')$
8:     $(C_0, C_1) \leftarrow ((C_0 \setminus W') \cup (U_0 \cap B), (C_1 \setminus W') \cup (U_1 \cap B))$
9:     $(U_0, U_1) \leftarrow (U_0 \setminus W', U_1 \setminus W')$
10:    $(\Xi_0, \Xi_1) \leftarrow (\Xi_0 \setminus W', \Xi_1 \setminus W')$
11:    $(\sigma_0, \sigma_1) \leftarrow (\sigma_0 \setminus W', \sigma_1 \setminus W')$
12: **end procedure**

---

associated winning strategy on W. It computes the q-attractor of the given winning set in the subgraph $U_0 \cup U_1$, adds it to the winning set of q and updates all other sets accordingly. In particular, the change sets are updated to contain the border nodes in the respective subgraph.

In order to refer to the data before and after the call, we refer to the unchanged name of a data variable for its value before the call and to the †-superscript for its value after the call.

Lemma 3.1 *Let (SC), (WC), (WE), (BE), (VC) and (IC) hold for both players. Let q be a player, $W \subseteq U_q$ be a set and ρ be a winning strategy for q on W. Then, calling* WINNING(q, W, ρ) *preserves (SC), (WC), (WE), (BE), (VC) and (IC) for both players. Additionally, $C_{q'}^\dagger = (C_{q'} \setminus W_q^\dagger) \cup (U_{q'} \cap \bigcup_{v \in W_q^\dagger \setminus W_q} Ev)$ for both players q' and $W \subseteq W_q^\dagger$.*

Also, $W_{q'} \subseteq W_{q'}^\dagger$ and $U_{q'} \subseteq U_{q'}^\dagger \cup W_{q'}^\dagger$ for both players q'.

Local SI expands the subgraph U_{q^*} of the current player q^* if there are no more improvement edges in U_{q^*} and the winner of v^* has not been decided yet. Then, it calls a function EXPAND with parameters q and N, where q is a player with a non-empty expansion set and N is a non-empty subset of the expansion set. This essentially adds the minimal set N' to U_q that contains N s.t. the new U_q fulfills the Subgraph-Closure property again. Additionally, the q-change set is increased by the set of new nodes; it may happen, that expanding U_q reveals new edges that lead to nodes in the winning sets W_0 or W_1. The EXPAND routine puts these nodes aside and calls the WINNING procedure in the end to update the winning sets. Therefore, it is possible that calling EXPAND actually decreases the size of U_q.

Lemma 3.2 *Let (SC), (WC), (WE), (BE), (VC) and (IC) hold for both players. Let q be a player and $N \subseteq E_q$ be a set. Then, calling* EXPAND(q, N) *preserves (SC), (WC), (WE), (BE), (VC) and (IC) for both players. Additionally, $N \subseteq U_q^\dagger \cup W_0^\dagger \cup W_1^\dagger$ and $C_{q'}^\dagger = (C_{q'} \setminus W_q^\dagger) \cup (U_{q'} \cap \bigcup_{v \in W_q^\dagger \setminus W_q} Ev) \cup (U_{q'}^\dagger \setminus U_{q'})$ for both players q'.*

Also, $W_{q'} \subseteq W_{q'}^\dagger$ and $U_{q'} \subseteq U_{q'}^\dagger \cup W_{q'}^\dagger$ for both players q'.

Whenever the change set C_q of a player q is changed, a re-evaluation process needs to take place that updates the valuation Ξ_q accordingly. The local evaluation for player q can be invoked by calling EVALUATE. During this process, it is possible that some nodes of U_q are identified as being won by player q. The function therefore returns a set of nodes newly won by player q as well as a winning strategy on this set. It updates all dependent node valuations as long as their valuation is not locally

---

Algorithm 3 Expansion of Nodes

1: **procedure** EXPAND(q, N)
2:     $(W'_0, \rho'_0) \leftarrow (\emptyset, \bot)$
3:     $(W'_1, \rho'_1) \leftarrow (\emptyset, \bot)$
4:     **while** N is not empty **do**
5:         $v \leftarrow$ pick from N
6:         $N \leftarrow N \setminus \{v\}$
7:         $U_q \leftarrow U_q \cup \{v\}$
8:         $\Xi_q(v) \leftarrow \emptyset$
9:         $E_q \leftarrow E_q \setminus \{v\}$
10:         $C_q \leftarrow C_q \cup \{v\}$
11:         $c \leftarrow$ chooser of v
12:         **if** there is some $u \in W_c \cup W'_c$ s.t. vEu **then**
13:             $W'_c \leftarrow W'_c \cup \{v\}$
14:             $\rho'_c \leftarrow \rho'_c[v \mapsto u]$
15:         **else if** $u \in W_{1-c} \cup W'_{1-c}$ for every u with vEu **then**
16:             $W'_{1-c} \leftarrow W'_{1-c} \cup \{v\}$
17:         **else if** $c = q$ **then**
18:             $E_q \leftarrow E_q \cup (vE \setminus (U_q \cup W_0 \cup W'_0 \cup W_1 \cup W'_1))$
19:         **else**
20:             $N \leftarrow N \cup (vE \setminus (U_q \cup W_0 \cup W'_0 \cup W_1 \cup W'_1))$
21:         **end if**
22:     **end while**
23:     WINNING$(0, W'_0, \rho'_0)$
24:     WINNING$(1, W'_1, \rho'_1)$
25: **end procedure**

---

consistent. The improvement set is then updated accordingly and the set of nodes that are won by the current valuation is returned along with the corresponding strategy.

Lemma 3.3 *Let (SC), (WC), (WE), (BE), (VC) and (IC) hold for both players and let q be a player. Then, calling* EVALUATE(q) *terminates and – returning W and ρ – preserves (SC), (WC), (WE), (BE), (VC) and (IC) for both players. Additionally, ρ is a winning strategy of q on W and $C_q = \emptyset$.*

Also, $W_{q'} \subseteq W^\dagger_{q'}$ and $U_{q'} \subseteq U^\dagger_{q'} \cup W^\dagger_{q'}$ for both players q'.

Local SI is then realised by the procedure MAIN which runs for as long as the initial node v^* is not contained in one of the two winning sets. If both the improvement set and the expansion set of the current player are empty we know that the other player wins the whole subgraph. Then it adds the entire subgraph to the opponent's winning set. Otherwise, either one of the two sets is non-empty. If the improvement set is empty, it expands the subgraph according to the picked non-empty set. Otherwise it improves the subgraph according to the improvement policy. Additionally, switching between the two players is consulted. Now, it may be the case that the valuation for one of the two players needs to be updated. Since an update of the valuation of one player may affect the winning sets of both and hence may require another update of either player's valuation afterwards, it re-evaluates for as long as the two change sets are non-empty.

Algorithm 4 Local Evaluation

1: **procedure** EVALUATE(q)
2:   $\tau \leftarrow \perp$
3:   $W \leftarrow \emptyset$
4:   $D \leftarrow \emptyset$
5:   **while** C_q is not empty **do**
6:    $v \leftarrow$ pick from C_q
7:    $C_q \leftarrow C_q \setminus \{v\}$
8:    $\vartheta \leftarrow \{v\}$
9:    **if** $v \in V_q$ and $v \in \mathrm{dom}(\sigma_q)$ **then**
10:     $\vartheta \leftarrow \Xi_q(\sigma_q(v)) \oplus v$
11:    **else if** $v \in V_{1-q}$ **then**
12:     $\vartheta \leftarrow \min_{\prec_q}\{\Xi_q(u) \oplus v \mid u \in vE \cap U_q\}$
13:    **end if**
14:    **if** $\vartheta \neq \Xi_q(v)$ **then**
15:     $\Xi_q(v) \leftarrow \vartheta$
16:     $D \leftarrow D \cup \{v\}$
17:     $C_q \leftarrow C_q \cup (Ev \cap U_q)$
18:    **end if**
19:   **end while**
20:   $D \leftarrow D \cup \bigcup_{v \in D} Ev$
21:   **while** D is not empty **do**
22:    $v \leftarrow$ pick from D
23:    $D \leftarrow D \setminus \{v\}$
24:    **if** $\Xi_q(v) = \infty$ **then**
25:     $I_q \leftarrow I_q \setminus \{v\}$
26:     $W \leftarrow W \cup \{v\}$
27:     **if** $v \in V_q$ **then**
28:      $\tau \leftarrow \tau[v \mapsto \sigma_q(v)]$
29:     **end if**
30:    **else if** $v \in V_q$ **then**
31:     **if** there is some $u \in vE \cap U_q$ with $\Xi_q(\sigma_q(v)) \prec_q \Xi_q(u)$ **then**
32:      $I_q \leftarrow I_q \cup \{v\}$
33:     **else**
34:      $I_q \leftarrow I_q \setminus \{v\}$
35:     **end if**
36:    **end if**
37:   **end while**
38:   **return** W and ρ
39: **end procedure**

Lemma 3.4 *Calling* MAIN *terminates s.t. there is one player q with $v^* \in W_q$ and ρ_q being a winning strategy for q on W_q.*

Algorithm 5 Main

1: **while** $v^* \notin W_0 \cup W_1$ **do**
2: **if** $E_{q^*} = \emptyset$ **and** $I_{q^*} = \emptyset$ **then**
3:   $W_{1-q^*} \leftarrow W_{1-q^*} \cup U_{q^*}$
4:   $\rho_{1-q^*} \leftarrow \rho_{1-q^*} \cup \tau_{G|_{U_{q^*}},\sigma_{q^*}}$
5: **else**
6:   **if** $I_{q^*} = \emptyset$ **then**
7:    $N \leftarrow$ pick non-empty subset of E_{q^*}
8:    EXPAND(q^*, N)
9:   **else**
10:    $I \leftarrow$ IMPRPOL(data)
11:    **for all** $(v, u) \in I$ **do**
12:     $\sigma_{q^*}(v) \leftarrow u$
13:     $C_{q^*} \leftarrow C_{q^*} \cup \{v\}$
14:     $q^* \leftarrow 1 - q^*$
15:    **end for**
16:   **end if**
17:   **repeat**
18:    stable $\leftarrow$ **true**
19:    **for** $i = 0, 1$ **do**
20:     $(W, \rho) \leftarrow$ EVALUATE(i)
21:     **if** $W \neq \emptyset$ **then**
22:      stable $\leftarrow$ **false**
23:      WINNING(i, W, ρ)
24:     **end if**
25:    **end for**
26:   **until** stable
27: **end if**
28: **end while**

---

4 Experimental Results

We report on the performance of the local SI algorithm in practice. We compare it in particular to two other algorithms for parity games: global SI by Jurdziński and Vöge [13] and Stevens and Stirling's local μ-calculus model checker [11]. Local SI is supposed to remedy one of global SI's distinct disadvantages, hence this comparison. Note that both global SI algorithms [13, 10] perform quite well in practice although Zielonka's recursive algorithm [14] is usually even a bit better [4]. However, it is the case that, whenever local SI is faster than global SI, it is also faster than the recursive algorithm. We also remark that the recursive algorithm is inherently global in nature, and we know of no idea how to create a local variant of this one. Furthermore, the parity game solving algorithm derived from the local model checking algorithm is of course a local algorithm already. It is therefore also natural contender for local SI. Previous tests have shown that it can be much slower than the global algorithms [4]. This comparison therefore shows that this is not due to the locality but to the solving technique.

Local SI has been implemented in PGSOLVER,[1] a publicly available tool for solving parity games which is written in OCaml and which contains implementations of the two competing algorithms as well as many others. Furthermore, PGSOLVER uses a specialised routine for global solving called *generic solver*. It does not feed the entire game graph to the chosen solving algorithm but performs optimising pre-processing steps like decomposition into SCCs and removal of trivial winning regions etc. These greatly speed up global solving [4]. The runtime results of the global algorithm reported here are obtained with all optimising pre-processing steps. On the other hand, these steps do not make sense in the context of local solving because the removal of trivial winning regions for instance is in fact a global operation. Hence, we do in fact compare local SI as it is against global SI embedded in this optimisaing framework.

Local SI is run in these experiments with the standard all-max policy on the respective improvement sets.

All tests have been carried out on a 64-bit machine with four quad-core Opteron[TM] CPUs and 128GB RAM space. As benchmarks we use some hand-crafted verification problems and random games. We report on the times needed to solve the games (globally vs. locally) as well as the number of nodes in the games that have been visited. This is an important parameter indicating the memory consumption of the algorithms on these benchmarks. The times for global SI are composed of the time it takes to generate the parity game (which is not part of the solving, strictly speaking, but needs to be considered too in comparison to the local algorithms since they generate the game while they are solving it) and the time it takes to perform the strategy iteration. We only present instances of non-negligible running times. On the other hand, the solving of larger instances not presented in the figures anymore has experienced time-outs after one hour, marked †.

			global SI			local MC		local SI	
	n	|Game|	$t_{generate}$	$t_{iterate}$	t_{solve}	visited	t_{solve}	visited	t_{solve}
FIFO elevator	3	603	0.02s	0.00s	0.02s	603	0.12s	603	0.28s
	4	3,120	0.08s	0.02s	0.10s	3,120	2.32s	3,120	8.66s
	5	19,263	0.66s	0.25s	0.91s	19,263	206.32s	19,263	414.55s
	6	138,308	5.64s	2.68s	8.32s	−	†	−	†
	7	1,130,884	57.57s	35.94s	93.51s	−	†	−	†
LIFO elevator	5	20,126	0.80s	0.34s	1.14s	1,237	0.02s	404	0.03s
	6	142,720	7.30s	4.13s	11.43s	3,195	0.07s	581	0.06s
	7	1,154,799	83.45s	66.91s	150.36s	22,503	0.66s	806	0.09s
	8	10,505,651	2342.61s	3596.42s	5939.03s	60,247	2.11s	1,085	0.15s

Figure 1: μ-Calculus Model Checking

μ-Calculus Model Checking.   As first benchmark we use the problem of verifying an elevator system. It serves *n* floors and maintains a list of requests which may contain each floor at most once. A detailed description can be found in [4]. The task is to verify that all paths satisfy the following fairness property: if the top floor is infinitely often requested then it is infinitely often served. This property can be formalised in the modal μ-calculus with alternation depth 2, and the resulting benchmarking parity games are obtained as μ-calculus model checking games.

We consider two different implementations of the system, differing in the way that the request list is stored: in FIFO or in LIFO style. Note that the FIFO implementation satisfies this fairness property because any requested floor will eventually be served whereas the LIFO implementation does not satisfy it. This is because new requests will be served before older ones and therefore requests can starve.

---

[1]http://www.tcs.ifi.lmu.de/pgsolver

With the FIFO system, both local algorithms need to explore the whole game in order to detect that the universally path quantified formula is satisfied or, equivalently, to find a winning strategy for player 0 which comprises the entire game graph. It is not too surprising that global SI with all its optimisations outperforms both of them in this case. But note that on the LIFO system, both local algorithms explore only a tiny part of the whole game and hence outperform the globally operating algorithm. Moreover, local SI is even much faster than the local model checker. See Fig. 1 for all the runtime results.

LTL Model Checking. We take the well-known dining philosophers problem as a system to be veri-fied. It consists of n forks, each of which can be in three different states (on the table or held by the left or right philosopher). Transitions are being done by lifting up or placing down one fork. Philosopher i eats as soon as he holds two forks. We check the LTL formula $\bigvee_{i=1}^{n} \mathtt{FG}eats_i$ on the system which asks for a scheduling of the fork lifting that enables one of them to eventually eat all the time. Note that LTL model checking can also be viewed as parity game solving [8].

Here the local algorithms only need to find a particular scheduling out of all possible ones which enables one of the philosophers to eventually eat all the time. Both the local SI and the local model checker outperform the global algorithm, again just because they only explore a small part of the whole game graph. Furthermore, local SI is remarkably more efficient than the local model checker in these test cases, i.e. it explores only a fractional part of the entire game in comparison and therefore needs much less time. See Fig. 2 for all the runtime results.

	n	|Game|	global SI			local MC		local SI	
			$t_{generate}$	$t_{iterate}$	t_{solve}	visited	t_{solve}	visited	t_{solve}
	9	1,084,808	29.48s	28.19s	57.67s	33,599	0.95s	3,439	0.75s
	10	3,615,173	177.93s	329.53s	507.46s	90,101	3.05s	4,374	1.21s
Philo-	11	11,927,966	1427.76s	2794.55s	4222.31s	201,566	8.16s	5,450	2.04s
sophers	12	?	†	—	—	854,922	70.38s	6,673	4.32s
	13	?	†	—	—	2,763,751	525.45s	8,052	6.82s
	14	?	†	—	—	—	†	9,595	10.00s

Figure 2: LTL Model Checking

Random Games. Finally, we consider games generated by a random model distributed with PG-SOLVER. It essentially creates small random games from a uniform distribution and connects them ran-domly in order to obtain a non-trivial SCC structure and therefore to prevent the creation of untypically special games [4]. The initial node is the — arbitrarily chosen — first node in the games representation. The results in Fig. 3 show that local SI is again much better than global SI and the local model checker on these games. Note that the number of visited nodes given there — as well as the running times — is the average over 100 runs and therefore not necessarily an integer value. Also, these random games cannot be created locally. Thus, the times given for all algorithms are the pure solving times without generation.

5 Further Work

We have presented an on-the-fly strategy improvement algorithm for solving parity games and shown that it can outperform existing global and even other local methods by several orders of magnitude. To be precise, the local SI algorithm presented here beats global algorithms in cases where a winning strategy

	|Game|	global SI	local MC		local SI	
		t_{solve}	visited	t_{solve}	visited	t_{solve}
	1,000	0.03s	37.93	0.00s	93.02	0.01s
	2,000	0.07s	103.99	0.01s	211.05	0.02s
	5,000	0.24s	159.14	0.11s	255.78	0.03s
	10,000	0.33s	592.42	0.95s	430.36	0.08s
Random	20,000	0.77s	6794.45	8.73s	453.05	0.10s
games	50,000	2.98s	–	†	669.90	0.26s
	100,000	6.53s	–	†	1001.80	0.48s
	200,000	17.92s	–	†	1154.40	1.15s
	500,000	68.33s	–	†	3443.40	5.82s

Figure 3: Random Games

does not stretch over the entire game graph, even when the global algorithms are embedded into the optimising framework of the generic engine in PGSOLVER.

It remains to see which of such optimisations which have been shown to aid global solving significantly [4] can be used in order to tune local SI, and whether these will also act as optimisations in this context rather than a slow-down. For instance, it may be possible to do the SCC decomposition of a game graph in a top-down fashion.

Note that in the expansion step, the local SI algorithm – as it is presented here – picks some node with a yet unexplored subgraph behind it and continues the exploration with this node. It is of course possibly to parametrise the algorithm with respect to the choice of this node. To be precise, one could consider yet expansion policies which – like the improvement policies – guide the algorithms choice of such nodes. However, note that the only information that these policies would have to rely on in order to choose a good node is a current node's successor names, priorities and their evaluation. The question whether or not one of them is good for the choosing player, however, depends on the structure of the subgraph lying behind this node rather than on this information. Naturally, this subgraph is not visible to the policy for otherwise the algorithm would not be local. We therefore suggest to simply pick such a successor by random choice.

A natural question that also arises is how to localise other global parity game solvers, in particular Zielonka's recursive one [14] and Jurdziński's small progress measures algorithm [7]. We currently see no method of doing this in a way that could improve over the global variants like local SI improves over global SI.

References

[1] A. Arnold, A. Vincent & I. Walukiewicz (2003): *Games for synthesis of controllers with partial observation*. *Theor. Comput. Sci.* 303(1), pp. 7–34.

[2] G. Bhat & R. Cleaveland (1996): *Efficient Local Model-Checking for Fragments of the Modal µ-Calculus*. In: *Proc. 2nd Int. Workshop on Tools and Algorithms for Construction and Analysis of Systems, TACAS'96*, *LNCS* 1055, Springer, pp. 107–126.

[3] E. A. Emerson & C. S. Jutla (1991): *Tree Automata, µ-Calculus and Determinacy*. In: *Proc. 32nd Symp. on Foundations of Computer Science*, IEEE, San Juan, Puerto Rico, pp. 368–377.

[4] O. Friedmann & M. Lange (2009): *Solving Parity Games in Practice*. In: *Proc. 7th Int. Symp. on Automated Technology for Verification and Analysis, ATVA'09, LNCS* 5799, pp. 182–196.

[5] O. Friedmann & M. Lange (2009): *Tableaux with Automata*. In: *Proc. Workshop on Tableaux vs. Automata as Logical Decision Procedures, AutoTab'09*.

[6] M. Jurdziński (1998): *Deciding the winner in parity games is in UP∩co-UP*. Inf. Process. Lett. 68(3), pp. 119–124.

[7] M. Jurdziński (2000): *Small progress measures for solving parity games*. In: *Proc. 17th Ann. Symp. on Theoretical Aspects of Computer Science, STACS'00, LNCS* 1770, Springer, pp. 290–301.

[8] M. Lange & C. Stirling (2002): *Model checking games for branching time logics*. Journal of Logic and Computation 12(4), pp. 623–639.

[9] A. Puri (1995): *Theory of hybrid systems and discrete event systems*. Ph.D. thesis, University of California, Berkeley.

[10] S. Schewe (2008): *An Optimal Strategy Improvement Algorithm for Solving Parity and Payoff Games*. In: *Proc. 17th Ann. Conf. on Computer Science Logic, CSL'08, LNCS* 5213, Springer, pp. 369–384.

[11] P. Stevens & C. Stirling (1998): *Practical Model-Checking using Games*. In: *Proc. 4th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'98, LNCS* 1384, Springer, pp. 85–101.

[12] C. Stirling (1995): *Local Model Checking Games*. In: *Proc. 6th Conf. on Concurrency Theory, CONCUR'95, LNCS* 962, Springer, pp. 1–11.

[13] J. Vöge & M. Jurdziński (2000): *A Discrete Strategy Improvement Algorithm for Solving Parity Games*. In: *Proc. 12th Int. Conf. on Computer Aided Verification, CAV'00, LNCS* 1855, Springer, pp. 202–215.

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