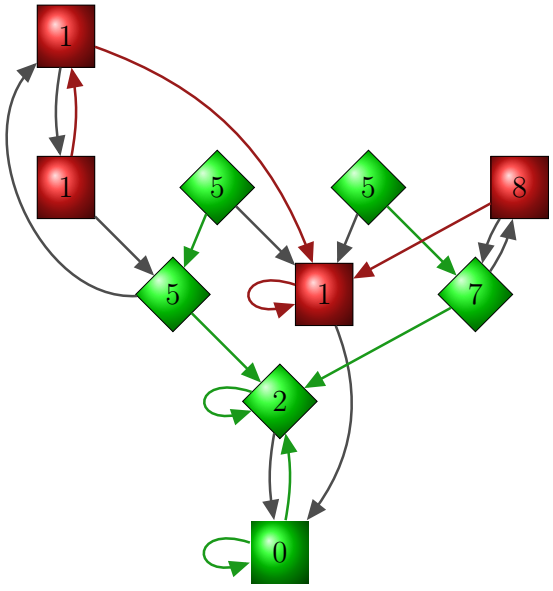
The PGSolver Collection of Parity Game Solvers

Version 4.0

Oliver Friedmann
Institut für Informatik
Ludwig-Maximilians-Universität München, Germany

Martin Lange
Theoretical Computer Science / Formal Methods
School of Electr. Eng. and Computer Science
University of Kassel, Germany

February 20, 2017



Contents

1 Introduction

1.1 The Importance of Parity Games

Parity games are simple two-player games of perfect information played on directed graphs whose nodes are labeled with priorities. The name *parity game* is due to the fact that the winner of a play is determined according to the parities (even or odd) of the priorities occurring in that play. In fact, it is determined by the maximal priority occurring infinitely often.

Parity games are an interesting object of study in computer science, and the theory of formal languages and automata in particular, for (at least) the following reasons.

- They are closely related to other games of infinite duration like mean payoff games, discounted payoff games, stochastic games, etc. [Jur98, Pur95, Sti95].
- They are at the core of other important problems in computer science, for instance, solving a parity game is known to be polynomial-time equivalent to model checking for the modal μ-calculus [EJS93, Sti95].
- They arise in complementation or determinisation problems for tree automata [GTW02, EJ91] and in emptiness and word problems for various kinds of (alternating) automata [EJ91].
- Controller synthesis problems can be reduced to satisfiability problems for branching-time logics [AAW03] which in turn require the solving of parity games because of determinisations of Büchi word automata into parity automata [Pit06, KW08].
- Solving a parity game is one of the rare problems that belongs to the complexity class NP∩co-NP and that is not (yet) known to belong to P [EJS93]. The variety of algorithms that have been invented for solving parity games is surely due to the fact that many people believe the problem to be in P.

1.2 Aim and Content of PGSolver

This variety of algorithms has provided a good understanding of the theory of parity games even though its computational complexity has possibly not yet been determined precisely. However, this theoretical knowledge is unmatched by the little amount of investigation into practical aspects of solving parity games. The aim of this project is to provide a platform for this: it should enable the comparison between different algorithms not just by the Landau-terms for their worst-case time complexities but by their actual performance on various classes of parity games.

The current version of this tool contains implementations of the following algorithms found in the literature:

- the recursive algorithm due to Zielonka [Zie98],
- the local model checking algorithm due to Stevens and Stirling [SS98],
- the strategy-improvement algorithm due to Jurdziński and Vöge [VJ00],
- the strategy-improvement algorithm due to Schewe [Sch08],
- the strategy-improvement algorithm reduction to discounted payoff games due to Puri [Pur95],
- the randomized strategy-improvement algorithm due to Björklund and Vorobyov [BV07],
- another randomized strategy-improvement algorithm due to Björklund, Sandberg and Vorobyov [BSV03],
- the small progress measures algorithm due to Jurdziński [Jur00],
- the small progress measures reduction to SAT due to Lange [Lan05],
- the dominion decomposition algorithm due to Jurdziński, Paterson and Zwick [JPZ06],
- the big-step variant of the latter due to Schewe [Sch07].

In addition, there is a new local strategy improvement algorithm by ourselves. Moreover there is a direct reduction to SAT based on strategy iteration due to Friedmann.

Finally, there is one heuristic solvers. Such solvers are sound: the answers they provide are correct. But they are not necessarily complete, for example because they may not terminate. Our heuristic algorithm just guesses strategies for both players until a (partial) winning strategy has been found.

The heuristic is included because it can solve certain classes of parity games very quickly and – most importantly – in a time that is independent of the number of priorities present in the game. On the other hand, it is easy to construct games on which it does not terminate, resp. infinite families of games on which the probability of termination decreases exponentially.

1.3 Structure of this Report

Chapter 2 formally introduces parity games and standard notions around the problem of solving them like winning regions and strategies, but also others that are needed in order to understand the constructions implemented in various solvers like attractor strategies, decompositions into subgames, etc. It then describes implemented meta-level optimisations for solving parity games, i.e. optimisations that apply to *any* solver. Next,

it shortly describes the implemented algorithms and heuristics. For those known before we refer to the corresponding literature for a detailed introduction into these algorithms. Here we only want to point out the rough functionality in order to be able to compare these algorithms and possibly attribute slow/fast solving to certain techniques.

Chapter 3 is the user's guide. It describes how to compile, install and run PGSolver, as well as how to specify parity games that it takes as input and how to read its output, etc.

PGSolver comes with programs that generate benchmarks. These are for example random games, games constructed in a way such that they are difficult for a certain algorithm to solve, or application-oriented games. These are described in Chapter 4.

Finally, Chapter 5 contains the developer's guide. It explains how to integrate another parity game solver – implemented in OCaml – into this tool.

2 Solving Parity Games

2.1 Technicalities

2.1.1 Parity Games

A *parity game* is a tuple $G = (V, V_0, V_1, E, \Omega)$ where (V, E) forms a directed graph whose node set 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 assume the underlying graph to be total, i.e. for every $v \in V$ there is a $w \in W$ s.t. $(v, w) \in E$.

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 an infinite path through the graph as follows. If the construction so far has yielded a finite sequence $v_0 \dots v_n$ and $v_n \in V_i$ then player i selects a $w \in v_nE$ and the play continues with the sequence $v_0 \dots v_n w$.

Every 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 $v_0 v_1 v_2 \dots$ 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.

In the following we will restrict ourselves to finite parity games. It is easy to see that in a finite parity game the winner of a play is determined uniquely since the range of Ω must necessarily be finite as well. Technically, we are considering so-called max-parity games. There is also the min-parity variant in which the winner is determined by the parity of the *least* priority occuring infinitely often. On finite graphs, though, these two games are equivalent in the sense that a max-parity game $G = (V, V_0, V_1, E, \Omega)$ can be converted into a min-parity game $G' = (V, V_0, V_1, E, \Omega')$ whilst preserving important notions like winning regions, strategies, etc. Simply let p an even upper bound on all the priorities $\Omega(v)$ for any $v \in V$. Then define $\Omega'(v) := p - \Omega(v)$. This construction also works the other way round, i.e. in order to transform a min-parity into a max-parity game.

A *strategy* for player i is a partial function $\sigma : V^* V_i \to V$, s.t. for all sequences $v_0 \dots v_n$ with $v_{i+1} \in v_i E$ for all $j = 0, \dots, n-1$, and all $v \in V_i$: $\sigma(v_0 \dots v_n) \in v_n E$. That is, a strategy for player i assigns to every finite path through G that ends in V_i a successor of the ending node. A play $v_0 v_1 \dots$ *conforms* to a strategy σ for player i if for all $j \in \mathbb{N}$ we have: if $v_j \in V_i$ then $v_{j+1} = \sigma(v_0 \dots v_j)$. 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* starting in some node $v \in V$ if player i wins every play that conforms to this strategy and begins in v. We say that player i *wins* the game G starting in v iff he/she has a winning strategy for G starting in v.

With G we associate two sets $W_0, W_1 \subseteq V$ with the following definition. W_i is the set of all nodes v s.t. player i wins the game G starting in v. We write W_i^G in order to name the parity game that the winning regions refer to, for example when it cannot uniquely be identified from the context.

Clearly, we must have $W_0 \cap W_1 = \emptyset$ for otherwise assume that there is a node v such that both players 0 and 1 have winning strategies σ_0 and σ_1 for G starting in v. Then there is a unique play $\pi = v_0 v_1 \ldots$ such that $v_0 = v$ and π conforms to both σ_0 and σ_1. It is obtained by simply playing the game while both players perform their choices according to their respective strategies. However, by definition π is won by both players, and therefore the maximal priority occurring infinitely often would have to be both even and odd.

On the other hand, it is not obvious that every node should belong to either of W_0 or W_1. However, this is indeed the case and known as *determinacy*: a player has a strategy for a game iff the opponent does not have a strategy for that game.

Theorem 1 ([Mar75, GH82, EJ91]) *Let $G = (V, V_0, V_1, E, \Omega)$ be a parity game. Then $W_0 \cap W_1 = \emptyset$ and $W_0 \cup W_1 = V$.*

A strategy σ for player i is called *positional* or *memory-less* or *history-free* if for all $v_0 \ldots v_n \in V^* V_i$ and all $w_0 \ldots w_m \in V^* V_i$ we have: if $v_n = w_m$ then $\sigma(v_0 \ldots v_n) = \sigma(w_0 \ldots w_m)$. That is, the value of the strategy on a finite path only depends on the last node on that path. An important feature of parity games is the fact that such strategies suffice.

Theorem 2 ([EJ91]) *Let $G = (V, V_0, V_1, E, \Omega)$ be a parity game, $v \in V$, and $i \in \{0, 1\}$. Player i has a winning strategy for G starting in v iff player i has a positional winning strategy for G starting in v.*

A positional strategy σ for player i induces a *subgame* $G|_\sigma := (V, V_0, V_1, E|_\sigma, \Omega)$ where $E|_\sigma := \{(u, v) \in E \mid u \in dom(\sigma) \Rightarrow \sigma(u) = v\}$. Such a subgame $G|_\sigma$ is, roughly speaking, basically the same game as G with the restriction that whenever σ provides a strategy decision for a node $u \in V_i$ all transitions from u but $\sigma(u)$ are no longer accessible.

A set $U \subseteq V$ is said to be i-closed iff player i can force any play to stay within U. This means that player $1 - i$ must not able to leave U but player i must always have the choice to remain inside U:

$$\forall v \in U : \left(v \in V_{1-i} \Rightarrow vE \subseteq U \right) \text{ and } \left(v \in V_i \Rightarrow vE \cap U \neq \emptyset \right)$$

Note that W_0 is 0-closed and W_1 is 1-closed.

A set $U \subseteq V$ induces a *subgame* $G|_U := (U, U \cap V_0, U \cap V_1, E \cap U \times U, \Omega|_U)$ iff the underlying transition relation $E \cap U \times U$ remains total i.e. for all $u \in U$ there is at least one $v \in U$ s.t. uEv. Clearly, each i-closed set U induces a subgame. We often identify a set $U \subseteq V$ that induces a subgame w.r.t. a fixed parity game with the induced subgame itself.

2.1.2 Dominions

A set $U \subseteq V$ is called an *i-dominion* iff U is i-closed and the induced subgame is won by player i. Clearly, W_0 is a 0-dominion and W_1 is a 1-dominion. That is, an i-dominion U covers the idea of a region in the game graph that is won by player i by forcing player $1-i$ to stay in U on the one hand; but on the other hand an i-dominion U is only won by player i when using a winning strategy on U.

To see more precisely what the concept of dominions is used for we need to introduce *attractors* and *SCC decompositions* of parity games.

2.1.3 Attractors and Attractor Strategies

Let $U \subseteq V$ and $i \in \{0,1\}$. Define for all $k \in \mathbb{N}$

$$\begin{aligned} Attr_i^0(U) &:= U \\ Attr_i^{k+1}(U) &:= Attr_i^k(U) \\ &\quad \cup (V_i \cap \{v \mid vE \cap Attr_i^k(U) \neq \emptyset\}) \\ &\quad \cup (V_{1-i} \cap \{v \mid vE \subseteq Attr_i^k(U)\}) \\ Attr_i(U) &:= \bigcup_{k \in \mathbb{N}} Attr_i^k(U) \end{aligned}$$

Intuitively, $Attr_i^k(U)$ consists of all nodes s.t. player i can force any play to reach U in at most k moves.

Lemma 3 ([Zie98, Sti95]) *Let $G = (V, V_0, V_1, E, \Omega)$ be a parity game and $U \subseteq V$. Let $V' := V \setminus Attr_i(U)$. Then $G' = (V', V_0 \cap V', V_1 \cap V', E \cap V' \times V', \Omega)$ is again a parity game with its underlying graph being total.*

In other words $V \setminus Attr_i(U)$ is $(1-i)$-closed; if additionally U is an i-dominion then $Attr_i(U)$ also is an i-dominion. This yields a general procedure for solving parity games: find a dominion in the game graph that is won by one of the two players, build its attractor of the dominion and investigate the complement subgame.

Each attractor for player i induces an *attractor strategy* for player i. It is defined for all $v \in Attr_i^k(U) \cap V_i$ for any $k \geq 1$ as $\sigma(v) = w$ iff $w \in Attr_i^{k-1}(U)$.

2.2 Local vs. Global Solvers

The problem of *solving* a parity game $G = (V, V_0, V_1, E, \Omega)$ is, roughly speaking, to determine which of the players has a winning strategy for that game. However, this does not take starting nodes into account. In order to obtain a well-defined problem, this is refined in two ways.

The problem of solving the parity game G *globally* is to determine, for *every* node $v \in V$ whether $v \in W_0$ or $v \in W_1$. The problem of solving G *locally* is to determine for a *given* $v \in V$ whether $v \in W_0$ or $v \in W_1$ holds. In addition, we require a *solver* to

compute (partial) winning strategies in the following sense: a local solver should return one strategy that is a winning strategy for player i if $v \in W_i$ for the given input node v. A global solver should return two strategies, one for each player s.t. player i wins exactly on the nodes $v \in W_i$ if he/she plays according to the strategy computed for her.

Clearly, the local and global problem are interreducible, the global one solves the local one for free, and the global one is solved by calling the local one $|V|$ many times. But neither of these indirect methods is particularly clever. Thus, there are algorithms for the global, and other algorithms for the local problem. We focus on the global problem here, but also use local solvers. In that case we expect them to determine for *at least* the given node v to which winning set it belongs but possibly and preferably also for other nodes. These methods can then be used to solve the global problem by calling the local algorithm *at most* $|V|$ many times.

Suppose $V = \{v_0, \ldots, v_n\}$. Then one can solve G globally using a local solver A as follows. Start A with starting node v_0. It will return two (possibly empty) winning sets W_0' and W_1'. In any case, we will have $W_0' \cap W_1' = \emptyset$, but not necessarily $W_0' \cup W_1' = V$. Let $W' := V \setminus (Attr_0(W_0') \cup Attr_1(W_1'))$. Then taking out all nodes in W' from G will result in a total parity game again by two applications of Lemma 3 and the fact that $Attr_0(W_0') \cap Attr_1(W_1') = \emptyset$. Let G' be this game. Then the winning positions for player i in G are $Attr_i(W_i')$ plus the winning positions for him/her in G'. Since G' will be smaller than G this can be used iteratively or recursively, to compute the entire W_0 and W_1.

The winning strategies can equally be assembled to a winning strategy σ for player i. Let σ' be player i's winning strategy on the subgame G', α be his/her attractor strategy for reaching W_i', and η be the strategy that guarantees him/her to win on W_i'. Then define

$$\sigma(v) \;:=\; \begin{cases} \eta(v)\ , & \text{if } v \in W_i' \\ \alpha(v)\ , & \text{if } v \in Attr_i(W_i') \setminus W_i' \\ \sigma'(v)\ , & \text{if } v \text{ is a node in } G' \\ \text{anything} & \text{otherwise} \end{cases}$$

The following theorem provides correctness of this construction.

Proposition 4 *Let $G = (V, V_0, V_1, E, \Omega)$ be a parity game, let W_i be the winning set of player i and $i \in \{0, 1\}$. Player i has a positional strategy σ for G s.t. σ is a positional winning strategy for G starting in any node $v \in W_i$.*

To see that this holds let σ_v be positional winning strategies for G starting in v for each node $v \in W_i$ (which have to exist by the former theorem). We assume all nodes $v \in W_i$ to be ordered w.r.t. a well-ordering $<$. A positional strategy σ winning from each node $v \in W_i$ can be defined as follows:

$$\sigma : v \in W_i \cap V_i \mapsto \sigma_{\min_<\{u \in W_i | v \in dom(\sigma_u)\}}(v)$$

To see that σ indeed is a positional winning strategy for G starting in any node $v \in W_i$ let $v_0 v_1 \ldots$ be a σ-conforming play with $v_j \in W_i$ for all $j \in \mathbb{N}$. Since $<$ is a well-ordering σ finally simulates the strategy of σ_u for some $u \in W_i$ and thus player i is wins this play.

2.3 Verifying Strategies

The problem of *verifying strategies* is to decide whether a given partition (W_0', W_1') of a parity game G along with two strategies $\sigma_0 : W_0' \cap V_0 \to V$ and $\sigma_1 : W_1' \cap V_1 \to V$ matches the partition into winning sets, i.e. whether $W_0' = W_0$ and $W_1' = W_1$ as well as σ_0 and σ_1 ensure the win of the respective player in the respective winning set.

The verification process basically traverses three phases. The algorithm verifies in the first phase that σ_0 and σ_1 are welldefined in the sense that a strategy actually uses valid transitions in the game graph. In the second phase the algorithm checks whether the strategies stay in their winning regions, i.e. $\sigma_i[W_i'] \subseteq W_i'$ for both player i, as well as whether the winning regions are closed w.r.t. the respective player.

The third phase finally checks whether the given strategies are winning strategies. In order to solve this problem, the algorithm computes the subgames $G_i := (G|_{W_i'})|_{\sigma_i}$ induced by the respective sets and strategies in question. Note that both G_i are special games, namely one-player games. Hence, they can be solved as described above.

Since σ_i is a winning strategy on W_i' iff σ_i is a winning strategy on G_i, it suffices to check whether the computed winning set $W_i^{G_i}$ correspond with W_i', and if they do not, the counter strategy for player $1-i$ can be used to extract a cycle in G following strategy σ_i that is won by $1 - i$.

2.4 Universal Optimisations

There are some optimisations that apply to *all* solvers. These universal optimisations efficiently try to reduce the overall complexity of a given parity game in order to reduce the effort spent by any solver. Clearly, such optimisations have to ensure that a solution of the modified game can be effectively and efficiently translated back into a valid solution of the original game.

In the following we describe optimisations that are implemented on top of every solving algrithm: SCC decomposition, detection of special cases, and compression. The next section then describes a generic algorithm that uses (some of – depending on the configuration) these optimisations in order to call a real solver on as few and little parts of a game as possible.

2.4.1 SCC Decomposition

Let $G = (V, V_0, V_1, E, \Omega)$ be a parity game. A *strongly connected component* (SCC) is a non-empty set $S \subseteq V$ with the property that every node in S can reach every other node in S, i.e. uE^*v for all $u, v \in S$ (where E^* denotes the transitive-reflexive closure of E). A strongly connected component S is *proper* iff uE^+v for all $u, v \in S$ (where E^+ denotes the transitive closure of E). In other words: An SCC S is proper iff $|S| > 1$ or $S = \{u\}$ and uEu.

Theorem 5 ([Tar72]) *Every parity game $G = (V, V_0, V_1, E, \Omega)$ can, in time $\mathcal{O}(|E|)$, be partitioned into SCCs $S_0, ..., S_n$ with $V = \bigcup_{i \leq n} S_i$ and $S_i \cap S_j = \emptyset$ for all $i \neq j$.*

Additionally there is a strict partial ordering $\rightarrow$ on these SCCs which is defined as follows:

$$S_i \rightarrow S_j \quad :\Longleftrightarrow \quad i \neq j \wedge \exists u \in S_i, v \in S_j : uEv$$

This strict partial ordering is generally known as the *topology* of the SCC decomposition. An SCC S is called *final* w.r.t. $\rightarrow$ if there is no SCC T s.t. $S \rightarrow T$. Note that every SCC topology of a finite graph must have at least one final SCC.

SCC decomposition of parity games as a universal optimisation works as follows. First, the game is decomposed into SCCs along with the computation of the strict partial ordering $\rightarrow$. Then, all final SCCs with respect to $\rightarrow$ are solved by a parity game solver. Since these SCCs are not connected to any other SCCs, all solutions obtained in this manner can be directly used as solutions in the global game.

Second, the attractors for both players with respect to the computed winning sets of all maximal solved SCCs are computed and removed from the game. The remainder is still a game, but because of the removal some of the original SCCs may not be SCCs anymore. All "damaged" ex-SCCs are again decomposed into SCCs and replaced by the new respective decomposition. In this way, the remaining decomposition can be used again to solve the rest of the game, again starting with those SCCs that are now final.

With this SCC decomposition it is not necessary to require solvers to solve an entire SCC let alone an entire game. Instead it suffices to have them solve at least a dominion for one of the players. Given an SCC S and two dominions $D_0, D_1 \subseteq S$ with $D_0 \subseteq W_0$ and $D_1 \subseteq W_1$, one simply computes the attractors $A_i := Attr_i(D_i)$ and considers the induced subgame $(S \setminus A_0) \setminus A_1$ which can be recursively solved by decomposition into SCCs etc.

2.4.2 Detection of Special Cases

There are certain kinds of special games that can be solved very efficiently by the following procedures. W.l.o.g. we can assume games to be proper strongly connected components. Remember that SCCs which are not proper consist of a single node v only that does not have an edge back to itself. The winner of v is the owner iff there is a successor that he/she wins. Since all successors belong to topologically greater SCCs we can assume them to be solved already, and thus, the winner of v is easily determined.

- *Self-cycle games*: Suppose there is a node v such that vEv. Then there are two cases depending on the node's owner p and the parity of the node's priority. If $\Omega(v) \not\equiv_2 p$ then taking the edge (v, v) is always a bad choice for player p and this edge can be removed from the game for as long as totality is preserved. If $\Omega(v) \equiv_2 p$ then taking this edge is always good in the sense that $\{v\}$ is a dominion for player p. Hence, its attractor can be removed as described above.

- *One-parity games*: If all nodes in a proper SCC have the same parity, the whole game is obviously won by the corresponding player no matter which transition the player uses. Hence, a winning strategy can be found by random choice.

- *One-player games*: A game G is a one-player game for player i iff for all $v \in V_{1-i}$ we have $|vE| = 1$. Such a one-player game that is an SCC can be solved using a simple fixed-point iteration. Player i wins the game iff there is a node u with $\Omega(u) \equiv_2 i$ and u is reachable from itself on a path that does not contain a priority greater than $\Omega(u)$. If there is such a cycle won by player i then the rest of the SCC lies in the attractor of the cycle (since player i is the only one to make choices); otherwise, if there is no cycle won by player i, the whole game is won by player $1 - i$.

2.4.3 Priority Compression

The complexity of a parity game rises with the number of different priorities in the game. This optimisation step attempts to reduce this number. Note that it is not the actual values of priorities that determine the winner. It is rather their *parity* on the one hand and their *ordering* on the other. For instance, if there are two priorities $p_1 < p_2$ in a game with $p_1 \equiv_2 p_2$ but there is no p' such that $p_1 < p' < p_2$ and $p' \not\equiv_2 p_1$ then every occurrence of p_2 can be replaced by p_1.

In general, let $P = (p_0, \ldots, p_k)$ be the list of all the priorities occurring in a game $G = (V, V_0, V_1, E, \Omega)$ s.t. $p_{i-1} < p_i$ for all $0 \leq i < k$. W.l.o.g. we assume p_0 to be even. If the least priority occurring in G is odd, then simply add $p_0 = 0$ to this list which does not affect the construction in any way. We will also use P to denote the *set* of all elements in P.

Take a decomposition of P into maximal sublists of elements with the same parity, i.e.

$$P \;=\; (p_{0,0}, \ldots, p_{0,m_0}, p_{1,0}, \ldots, p_{1,m_1}, \ldots, p_{n,0}, \ldots, p_{n,m_n})$$

with $p_{i,j} \equiv_2 p_{i,j'}$ for all $0 \leq i \leq n$, $0 \leq j < j' \leq m_i$ and $p_{i,m_i} \not\equiv_2 p_{i+1,0}$ for all $0 \leq i < n$. This defines a partial mapping $\omega : \mathbb{N} \to \mathbb{N}$ as

$$\omega(p) \;=\; \begin{cases} i & \text{, if } p = p_{i,j} \text{ for some } j \\ \text{undefined} & \text{, otherwise} \end{cases}$$

Note the following facts about ω:

- ω is defined on all priorities occurring in G;
- ω is *decreasing*: we have $\omega(p) \leq p$ for all $p \in P$;
- ω is *monotone*: for all p, p' with $p \leq p'$ we have $\omega(p) \leq \omega(p')$;
- ω *preserves parities*: we have $\omega(p) \equiv_2 p$ for all $p \in P$;
- ω is *dense*: for all $p, p' \in P$ with $\omega(p) + 1 < \omega(p')$ there is a p'' with $\omega(p) < \omega(p'') < \omega(p')$.

Now define another parity game $G' := (V, V_0, V_1, E, \Omega')$ with $\Omega'(v) := \omega(\Omega(v))$. I.e. G' results from G by reducing the priorities according to the function ω. Then G' is

equivalent to G in the sense that the players' winning regions coincide in the two games, and a winning strategy σ for some player i in G is also a winning strategy for him/her in G' and vice-versa. This is guaranteed by the properties of ω identified above: due to monotonicity and preservation of parities, the greatest priority ocurring in an infinite play in G is even iff it is even in the same play in G'. The property of being decreasing guarantees that this should in general be an optimisation, and density says that G' is optimal w.r.t. this optimisation.

2.4.4 Priority Propagation

Note that any play visiting a node v has to – due to totality – also visit one of the successors of v. Now suppose that the priorities of all successors of v are greater than the priority of v itself. Then v's priority is irrelevant in the sense that no play is won by either player because v's priority occurs in it. For those plays not visiting v at all this is trivial and for those plays that do visit v this is simply because v is certainly not the geatest priority occurring in this play, let alone occurring infinitely often. Hence, v's priority can be replaced by a greater one.

In general, let $G = (V, V_0, V_1, E, \Omega)$. Applying *backwards propagation* in node v results in the game $G' = (V, V_0, V_1, E, \Omega')$ where $\Omega' = \Omega[v \mapsto \max\{\Omega(v), \min\{\Omega(w) \mid w \in vE\}\}]$. Similarly, *forwards propagation* replaces v's priority with the minimum of all priorities of its predecessors if that is greater than its current priority. It is equally sound because any play that visits v infinitely often must visit one of its predecessors infinitely often too.

Both backwards and forwards propagation can be iterated and combined thus reducing the range of priorities in a game.

2.5 The Generic Solver

Solving a parity game is done by a central module called the *generic solver*. It combines the universal optimisations described above with any of the implemented algorithms of heuristics described above. This is realised by taking a solver, i.e. one of these algorithms or heuristics as a parameter. Then it roughly works as follows.

1. Self-cycles are eliminated from the game, and attractors of nodes for which the self-cycle is part of a winning strategy are computed and removed.
2. The entire game is decomposed into SCCs.
3. Terminal SCCs are solved as follows.
   a) Priorities are compressed.
   b) The SCC is checked for being a special case of a game.
      - If it is, winning regions and strategies are constructed accordingly.
      - Otherwise, the solver given as the parameter is used to solve this SCC.

c) Attractors of the computed winning regions are also computed, together with corresponding strategies which are added to the winning regions and strategies.

d) The computed winning regions are removed from the game.

e) All non-terminal SCCs which have lost some nodes in the removal of these attractors are again decomposed into more fine-grained SCCs.

4. Step 3 is repeated until the entire game is solved.

Note that the removal of terminal SCCs will make other, previously non-terminal SCCs terminal. Also, note that this scheme is sound – the regions and strategies it computes are in fact winning regions and winning strategies for the corresponding players on the given game – if the parameter solver is sound. Hence, it can also be used with sound heuristics. Furthermore, it is complete if it is guaranteed that the parameter solver solves at least one node of every SCC that it is given. Hence, the solvers used as backends need not be complete for the generic solver to be complete. This is why this scheme can solve whole games using heuristics that are incomplete themselves.

We also note that the features SCC decomposition, detection of special cases, and priority compression can be switched off via command-line options. This may be useful when the performance of an algorithm on its own is to be measured. In addition, it is possible to turn the feature priority propagation on in which case it is done before priority compression. However, in general this does not seem to be an optimisation since it slows down virtually any backend.

2.6 Implemented Algorithms

We shortly describe the algorithms that are implemented in PGSolver. The aim is not to give a complete description of these algorithms. For details and an account of the theory behind them please follow the literature. We just give an idea of how these algorithms work, in order to be able to name implementation details and the optimisations that are carried out.

For the terms describing the asymptotic time and space complexities of these algorithms we introduce the convention that n denotes the number of nodes in a game, e denotes the number of edges, and d denotes the number of priorities.

2.6.1 The Recursive Algorithm

This algorithm falls out of the constructive determinacy proof for parity games due to Zielonka. It decomposes the game at hand to smaller ones recursively by simultaneous induction on the number of priorities and the number of nodes in the game. In the base cases, if the game only has one node or one priority, the winner and corresponding strategy can easily be obtained, in the latter case as a random strategy for example. In the other cases a winning strategy can be assembled out of strategies for smaller

subgames and an attractor strategy for one of the players reaching the set of nodes with maximal priority in the game.

Algorithm 1 (Recursive Algorithm)

Author(s)	W. Zielonka
Literature	[Zie98]
Short description	Decomposition into subgames with recursion on number of nodes and priorities
Time complexity	$\mathcal{O}(e \cdot n^d)$
Space complexity	$\mathcal{O}(e \cdot n)$

2.6.2 The Local Model Checking Algorithm

The only algorithm solving parity games locally in our collection is the μ-calculus model checker due to Stevens and Stirling. Since a parity game can be regarded as the product of an unknown transition system and an unknown μ-calculus formula (even though there may not exists such factors), this algorithm can also be used to solve parity games. It basically explores a game depth-first and whenever it reaches a cycle it stops, storing the node starting the cycle along with a cycle progress measure as an *assumption* for the cycle-winning player. Then, the exploration is backtracked in the sense that if the losing player could have made other moves they are again explored depth-first. If this leads to a cycle-win for the other player, the whole process starts again, now with respect to the other player. Whenever the backtracking finally leads to the starting node of a cycle the node is registered as a *decision* for the player which basically can be seen as being a preliminary winning node for the respective player. Additionally, if there are assumptions of the other player for the respective node, these assumptions are dropped, and all depending assumptions and decisions are invalidated.

Algorithm 2 (Model Checking Algorithm)

Author(s)	P. Stevens and C. Stirling
Literature	[SS98]
Short description	Exploring the game depth-first, detecting cycles and backtracking subsequently for other possible moves

There are no sensible estimations on the worst-case time and space complexities of this algorithm in the literature.

2.6.3 The Strategy Improvement Algorithm

The strategy improvement algorithm due to Jurdziński and Vöge picks a strategy for one of the two players, say for player 0, and computes a valuation of the strategy-induced subgame. This valuation is used to select a new strategy for player 0 by choosing transitions from player 0 choice points maximizing the valuation for the respective target node.

The process of valuating the current strategy and subsequently picking a new one is iterated until all transitions of the new strategy are not assigned better valuations than the transitions of the former strategy. The final valuation is then used to infer winning sets and winning strategies for both players.

Algorithm 3 (Strategy Improvement Algorithm)

Author(s)	M. Jurdziński and J. Vöge
Literature	[VJ00, SV00]
Short description	Iteratively improves an initialization strategy until it satisfies a winning-strategy-predicate
Time complexity	$\mathcal{O}(2^e \cdot n \cdot e)$
Space complexity	$\mathcal{O}(n^2 + n \cdot \log d + e)$

2.6.4 The Optimal Strategy Improvement Method

This strategy improvement algorithm due to Schewe guarantees to select, in each improvement step, an optimal combination of local strategy modifications. The estimation produced in each iteration step are used to infer winning regions along with winning strategies for both players.

Algorithm 4 (Optimal Strategy Improvement Method)

Author(s)	S. Schewe
Literature	[Sch08]
Short description	Iteratively improves an estimation until a fixed point is reached
Time complexity	$\mathcal{O}(e \cdot (\frac{n+d}{d})^d \cdot \log(\frac{n+d}{d}))$
Space complexity	$\mathcal{O}(n^2)$

2.6.5 The Strategy Improvement by Reduction to Discounted Payoff Games

This algorithm translates the given parity game to a corresponding discounted payoff games and solves the latter by Puri's algorithm. The winning regions as well as winning strategies for the discounted payoff games directly correspond to those of the original parity game.

Algorithm 5 (Strategy Improvement for DPGs)

Author(s)	A. Puri
Literature	[Pur95]
Short description	Iteratively improves an estimation until a fixed point is reached

2.6.6 Probabilistic Strategy Improvement

A probabilistic strategy iteration method that has an subexponential upper bound on the number of expected iterations that are required to solve the game.

Algorithm 6 (Probabilistic Strategy Improvement)

Author(s)	H. Björklund and S. Vorobyov
Literature	[BV07]
Short description	Iteratively improves an estimation until a fixed point is reached

2.6.7 Probabilistic Strategy Improvement 2

Another probabilistic strategy iteration method that has an subexponential upper bound on the number of expected iterations that are required to solve the game.

Algorithm 7 (Probabilistic Strategy Improvement 2)

Author(s)	H. Björklund, S. Sandberg and S. Vorobyov
Literature	[BSV03]
Short description	Iteratively improves an estimation until a fixed point is reached

2.6.8 Local Strategy Improvement

A local version of the strategy iteration paradigm.

Algorithm 8 (Local Strategy Improvement)

Author(s)	O. Friedmann and M. Lange
Literature	[FL10, FL12]
Short description	Iteratively improves an estimation until a fixed point is reached

2.6.9 The Small Progress Measures Algorithm

The existence of a winning strategy for either of the players can be characterised be a condition that is local to the nodes of the parity game. Each node carries a tuple of values, and those values have to be larger than one or all of the values in successor nodes depending on the own of the node and its priority. It can be shown that the values in a node can be bounded by a number depending on the nodes reachable from the one at hand.

Jurdziński suggest to find these values iteratively starting with 0 everywhere and increasing them whereever necessary to respect their relation. Non-existence of a winning strategy on certain parts of the game is found when the iteration tries to increase values beyond their pre-computed maxima.

Algorithm 9 (Small Progress Measures Algorithm)

Author(s)	M. Jurdziński
Literature	[Jur00]
Short description	Iteratively increase lexicographically ordered tuples
Time complexity	$\mathcal{O}(d \cdot e \cdot (\frac{n}{d})^{d/2})$
Space complexity	$\mathcal{O}(d \cdot n \cdot \log n)$
Optimisations	tight computation of maximal values

The performance of the algorithm depends very much on a good approximation of the maximal values in these tuples since this directly affects the maximal running time. Suppose this algorithm is used to find a winning strategy for player 0. Then each tuple contains a field for every odd priority that occurs in the game. The maximal value in the field p of the tuple at node v for some odd priority p is 1 plus the number of nodes

of priority p in the same SCC as v that are reachable from v without passing through a node with a priority higher than p.

Our implementation performs two iterations – one for each player. Note that the original exposition of the small progress measures algorithm allows to compute the winning region and a strategy for one of the players. Clearly, the winning region for the other player can easily be inferred from this but his/her strategy cannot. One possibility would be to rerun the algorithm on the winning region for the other player measuring the progress for him/her. Another possibility is to iterate the progress measures for both players in all nodes right away from the beginning.

2.6.10 The Small Progress Measures Reduction to SAT

Since solving parity games is known to be in NP there must be a polynomial reduction to SAT, the satisfiability problem for propositional logic. Such a reduction is basically given by the small progress measures algorithm. Instead of iteratively computing values one lets a SAT solver find them. Since these values can be bounded, there is a finite number of bits representing these numbers, and these bits can be seen as propositional variables for the SAT solver. The necessary relations between the numbers are only less-than and less-than-or-equals, and it is not difficult to write down propositional formulas which describe such relations between natural numbers in terms of constraints on their bits.

Again, this is supposed to be a *global* parity game solver. Hence, it has to report winning regions and strategies for both players. As said above, Jurdziński's original characterisation of these in terms of small progress measures caters for one of the players only, and therefore needs space linear in $\lceil \frac{d}{2} \rceil$. In order to obtain strategies for both players one has to use the algorithm twice. The same holds for the symbolic execution by a reduction to SAT. The obvious choice here is to simply create constraints for two sets of progress measures. But then the reduction technically does not map into the satisfiability problem for propositional logic because the resulting formula is always satisfiable, and a satisfying variable assignment encodes both the partition into winning regions and the positional strategies.

Algorithm 10 (Small Progress Measures Reduction to SAT)

Author(s)	M. Lange
Literature	[Lan05]
Short description	Symbolic encoding in propositional logic of the small progress measure algorithm
Time complexity	$\mathcal{O}(e \cdot d)$ + running time of the SAT solver
Space complexity	$\mathcal{O}(e \cdot d)$ + space needed by the SAT solver

2.6.11 The Direct Reduction to SAT

This formalises in propositional logics the existing of strategies and requires them to be winning by checking that every cycle which is reached by following the strategy for one of the players sees has a greatest priority which is good for that player. As with the previous reduction, the resulting formula is always satisfiable.

Algorithm 11 (Direct Reduction to SAT)

Author(s)	O. Friedmann
Short description	Symbolic encoding in propositional logic of the a direct predicate
Time complexity	$\mathcal{O}(n^3)$ + running time of the SAT solver
Space complexity	$\mathcal{O}(n^3)$ + space needed by the SAT solver

2.6.12 The Dominion Decomposition Algorithm

This algorithm is the first deterministic subexponential algorithm for parity games. It basically refines the recursive algorithm due to Zielonka. First, one searches for small ($\leq \sqrt{2 \cdot n}$) dominions by simply building each small subset, checking whether it is closed w.r.t. player 0 or player 1 and if so checking whether the closed small subset is an i-dominion by using the original recursive algorithm. If it actually is a dominion the attractor is built and the complement subgame recursively solved. If there is no small dominion the algorithm switches to the original recursive algorithm (and searches for smaller dominions in subsequent recursive calls).

Algorithm 12 (Dominion Decomposition Algorithm)

Author(s)	M. Jurdziński, M. Paterson, and U. Zwick
Literature	[JPZ06]
Short description	Brute-force search for small dominions with subsequent decomposition into subgames with recursion on number of nodes and priorities
Time complexity	$n^{O(\sqrt{n})}$
Space complexity	$\mathcal{O}(e \cdot n)$

2.6.13 The Big-Step Algorithm

This algorithm refines the dominion decomposition algorithm by replacing the brute-force search for small dominions by a restricted run of the small progress measures

algorithm. Basically, the sum of the entries in a small progress measure is limited by the size of the largest dominion that is searched for ($\sqrt[3]{n^2 \cdot d}$). Hence, the small progress measures algorithm identifies all dominions less or equal to that limit. If there is no small dominion the algorithm switches to the original recursive algorithm (and searches for smaller dominions in subsequent recursive calls).

Algorithm 13 (Big-Step Algorithm)

Author(s)	S. Schewe
Literature	[Sch07]
Short description	Small progress measures-based search for small dominions with subsequent decomposition into subgames with recursion on number of nodes and priorities
Time complexity	$O(e \cdot n^{\frac{1}{3}d})$
Space complexity	$\mathcal{O}((e + d \cdot \log n) \cdot n)$

2.7 Implemented Heuristics

In addition to the algorithms described above, there is also one heuristics implemented in PGSolver. Heuristics are not necessarily supposed to compete with the proper algorithms but are interesting on their own because they can provide insight into the difficulty of solving certain parity games, as well as be very quick on certain classes of games.

As with heuristics in general, it can be difficult to give estimations on their worst-case time complexities. They may even be non-terminating. We classify them according to three properties: being *sound*, i.e. if they claim that a node belongs to the winning region of one of the players then it really does so; being *complete*, i.e. finding those regions; and being *terminating*.

We distinguish completeness and termination explicitly because a heuristic may be terminating but incomplete, for example when it realises that it cannot solve the game at hand. In this case it notionally provides a *don't know* answer and practically one of the algorithms above is used to solve the game.

2.7.1 The Strategy Guessing Iteration

This heuristic guesses random strategies for both players and computes the winning regions w.r.t. these strategies. If the winning regions are not empty, the algorithm returns these sets. Otherwise the whole process starts all over again.

Heuristic 1 (Strategy Guessing Iteration)

Author(s)	O. Friedmann
Short description	tries to find dominions by selecting a random strategy
Space complexity	$\mathcal{O}(n)$
Soundness	yes
Completeness	no
Termination	no

3 User's Guide

3.1 License

This software is distributed under the BSD license.

3.2 Change Log

Changes are classified according to their importance. Changes effecting only the performance of parts of the tool or are less important otherwise are marked with one asterisk ($*$). Changes that are worth noting like the addition of new features etc. are marked with two asterisks ($**$). Finally, major changes that would effect usability, for example breaks in backwards compatibility, changes in the user interface, etc., are marked with three asterisks ($***$).

Changes incorporated into version 4.0:

*** The interface and implementation of the module `Paritygame` have undergone some major changes, mostly concerning the data types for parity games. This type is now abstract, so access to and manipulations of parity games are now only possible through specialised functions in this module. A direct access of parity games as arrays is no longer possible. Consequently, access to parity games in all other modules has been changed accordingly. This encapsulation step enables us to change the internal data structures for parity games easily for optimisation purposes without changing any of the algorithms around them. It is also worth noticing that a node in a parity game now knows its predecessors as well as its successors. Hence, functions for computing transposed graphs etc. have been removed.

** PGSolver is now built using `ocamlbuild`. It requires the OCaml packages `ocamlfind` and `ounit`. Get them using `opam`. Moreover, PGSolver now requires OCaml version 4.03.0.

*** If a textual presentation of a parity game starts with the keyword `parity` then it is followed by the number of nodes in the game, rather than the largest identifier of a node in the game. So any "`parity` n" should now be "`parity` $n+1$".

* A framework for the easy creation of benchmarks derived from model-checking problems is now provided.

** Support for the `Z3` has been discontinued.

Changes incorporated into version 3:

** PGSolver is now linkable as library.

* A new local strategy improvement algorithm has been added.

** Local Solving of Games has been enabled.

* Generators can now be linked directly into PGSolver.

* Two new randomized strategy improvement algorithms have been incorporated.

* The lists of solvers and generators are now maintained in `./Solvers` and `./Generators` respectively.

* Printing and parsing of solution and strategies.

* Conversion between min-parity and max-parity games as a new transformer feature.

* Two new generators: Towers of Hanoi as a reachability game and a fairness verification of an elevator system.

Changes incorporated into version 2:

* In order to allow more efficient parsing, the specification format for parity games has been extended. It is now possible to include at the beginning of the specification the maximal index of a node in the game. If this is done, then parsing will be quicker.

* All the random game generators are now based on a more efficient generation of sets of random numbers.

* The computation of attractor regions has been improved.

*** Confusing terminology has been clarified: the algorithm due to Stevens and Stirling [SS98] is now referred to as the *model checker* rather than the former *game-based algorithm*. Command line parameters to `pgsolver` have been changed accordingly.

** A useful benchmarking tool has been included in the distribution.

* This change log has been included in this documentation – in case you hadn't noticed.

3.3 Installation Guide

3.3.1 Obtaining the Relevant Parts

You can obtain the source code for PGSolver from

`https://github.com/tcsprojects/pgsolver`

Download the latest sources.

```
~> git clone https://github.com/tcsprojects/pgsolver
```

This will create a directory `pgsolver` and various subdirectories in it. Get the required submodules.

```
~> cd pgsolver
~/pgsolver> git submodule update --init
```

In order to compile PGSolver from source code you will need the OCaml compiler and the compilation tool `ocamlbuild`. A convenient way to get both is to use the OCaml Package Manager `opam`. Install it via any package manager for your system or download it from

`https://opam.ocaml.org/`

Get the OCaml compiler and Ocamlfind.

```
~> opam switch 4.03.0
~> eval ‘opam config env‘
~> opam install ocamlbuild
```

If you intent to contribute to the development of PGSolver you may want to use unit tests as well. This requires two more packages.

```
~> opam install ocamlfind ounit
```

PGSolver consists of many executables; their creation is still guided by `make`, so you need to make sure that GNU Make is installed. We recommend version 3.81 or higher but earlier ones, as well as other implementations may suffice as well.

If you want to use any of the reductions to SAT in order to solve parity games then you need one of more SAT solvers. There are two SAT solver backends that are already supported by PGSolver:

- `MiniSat` developed in Göteborg. It is available from

  `https://github.com/niklasso/minisat`

  We recommend the C-version of 2.2.0. Other version will probably not work with our interface.

- `PicoSAT` developed in Linz. It is available from

  `http://fmv.jku.at/picosat`

  We recommend version 965. Earlier version will probably not suffice.

In addition to these two, the (somewhat outdated) SAT solver `zChaff`, developed in Princeton, is also still supported. It is available from `http://www.princeton.edu/~chaff/zchaff.html`. We recommend at least version 2007.3.12. Earlier version will probably not suffice.

3.3.2 Compiling PGSolver

Now change into the PGSolver directory.

```
~> cd pgsolver
```

To start the compilation, type

```
~/pgsolver> make
```

This is the same as

```
~/pgsolver> make all
```

which is an abbreviation for `make pgsolver generators tools`. If you only want the executable PGSolver then

```
~/pgsolver> make pgsolver
```

suffices. The same holds for the executables containing benchmark generators and some tools to manipulate parity games. After successful compilation, the executables can be found in the subdirectory `bin`.

You can delete all files that have been created during the compilation process by running

```
~/pgsolver> make clean
```

3.3.3 Integrating SAT solvers

PGSolver currently supports three backend SAT solvers, namely `PicoSAT`, `ZChaff` and `MiniSat`. It should be also possible to integrate other SAT solvers; see the Developer's Guide for more information on that subject. In order to integrate one of them into PGSolver, you need to follow these steps:

1. Download and compile the respective SAT solver.
2. Adjust the `./SatConfig` file s.t. the usage of the respective SAT solver is enabled and all required links point to the correct target.
3. Remove the current compilation of PGSolver: `make clean`.
4. Recompile PGSolver: `make all`.

Compiling `PicoSAT`

Obtain the source code of `PicoSAT` (we recommend at least version 632), unpack it and consult the included `README` file for instructions on how to compile it. Usually,

```
~/picosat> ./configure && make
```

will do the job.

The compilation of `PicoSAT` produces an executable `picosat`. However, PGSolver uses the object files which can be linked into the program directly. Make sure that the variable `PICOSAT` in PGSolver's `./SatConfig` file points to the directory in which the object files can be found, for example

```
PICOSAT=/usr/local/lib/picosat-965/libpicosat.a
```

Compiling `MiniSat`

Obtain the source code of `MiniSat` (we recommend the C-version 2.20), unpack it and consult the included `README` file for instructions on how to compile it. Usually,

```
~/minisat> make
```

will do the job.

The compilation of `MiniSat` produces an executable `minisat`. However, PGSolver uses the object files which can be linked into the program directly. Make sure that both `MINISAT` variables in PGSolver's `./SatConfig` file point to the respective directories, for example

```
MINISAT = /usr/local/lib/minisat/build/release/lib/libminisat.a
MINISAT_INC = /usr/local/lib/minisat
```

Compiling `zChaff`

Obtain the source code of `zChaff` (we recommend at least version 2007.3.12), unpack it and consult the included `README` file for instructions on how to compile it. If you are lucky then a simple

```
~/zchaff> make
```

will do the job.

The compilation of `zChaff` produces an executable `zchaff`. However, PGSolver uses the library version which can be linked into the program directly. This is usually called `libsat.a` and is also produced by `zChaff`'s compilation process. Make sure that the variable `ZCHAFF` in PGSolver's `./SatConfig` file points to the directory in which `libsat.a` can be found, for example

```
ZCHAFF=/usr/local/lib/zchaff-2007-3-12/libsat.a
```

3.4 Running PGSolver

3.4.1 Invocation

A successful compilation produces an executable binary called `pgsolver` which, unless specified otherwise in `Makefile`, resides in the subdirectory `bin`. Type

```
~/pgsolver> bin/pgsolver --help
```

or

```
~/pgsolver> bin/pgsolver -help
```

for a list of command-line options and a description of the command-line parameters that the program expects.

The specification of the parity game to be solved – cf. Sect. 3.5 – can either be given in a file or through `stdin`. The latter case is the default, and to switch to the former one only needs to give the name of the file including the specification as a command-line argument.

```
~/pgsolver> bin/pgsolver tests/test1.gm
```

3.4.2 Command-Line Parameters

PGSolver's behaviour can be changed through command-line parameters. In particular, you can determine the algorithm that is used for solving, specify whether or not you want to view the result in text mode or graphically, etc. Command-line parameters can be given in any order, although some are conflicting and in that case the latter overrides the former; for example when you specify more than one algorithm that should be used for solving. The currently understood command-line parameters are:

⟨*filename*⟩
: Tells `pgsolver` to look for the specification of the parity game to solve in the file ⟨*filename*⟩.

`-v` ⟨*level*⟩
: Sets the verbosity level, valid arguments are 0–3, the default is 1. With verbosity level 0, `pgsolver` will be very humble and not bother `stdout` with its pathetic goobledigook. With verbosity level 1, it will tell you what it does and what the winning regions and strategies are. With verbosity level 2, it will tell you a bit more, but this very much depends on which algorithm is chosen for solving etc. It may for example tell you something it has found out about the SCC structure or priority distribution in the game. Verbosity level 3 is for debugging purposes. Again, this very much depends on the solving algorithm.

`-d` ⟨*filename*⟩
: Tells `pgsolver` to open the file ⟨*filename*⟩ for writing and print into it the `dot`-code of the parity game as a graph, see also Sect. 3.6. If the game has been solved during this invocation then the graphical presentation will display the winning regions and strategies. You can use option `-n` in combination with this one to display a pure parity game, i.e. without the winning information.

`--disableglobalopt` or `-dgo`
: Tells `pgsolver` to completely disable any global optimisations. Global optimisation is enabled by default.

`--disablesccdecomposition` or `-dsd`
: Tells `pgsolver` to decompose the game into strongly connected components. This is enabled by default.

`--disablelocalopt` or `-dlo`
: Tells `pgsolver` to completely disable any local optimisations. Local optimisation is enabled by default.

`--disablespecialgames` or `-dsg`
: Tells `pgsolver` to completely disable any algorithms solving special instances. This is enabled by default.

`--verify` or `-ve`
: This causes `pgsolver` to perform, after solving, an additional check that the reported winning strategies are correct; useful for finding bugs in new algorithm implementations.

`--justheatCPU` or `-jh`
: On large parity games, printing the winning information can take a long time because of the bottleneck `stdout`. This option turns the printing of that information off. Useful if one is interested in running times only for instance.

`--changesat` ⟨*satsolver*⟩ or `-cs` ⟨*satsolver*⟩
: Selects the SAT solver backend that is to be used. Only affects parity game solving algorithms that depend on SAT solving. If there are less than two SAT solvers linked into PGSolver, this parameter is disabled.

`--printsolonly`
: Instead of printing a human readable form of the solution, PGSolver prints a format that can be directly parsed again by the framework.

`--printsolvedgame` or `-pg`
: Prints the solved game in a parseable form. This will be a subgame of the original parity game in which all nodes where the winner of the node is also the owner are restricted to the winning edge.

`--parsesolution` ⟨*file*⟩ or `-ps` ⟨*file*⟩
: Parses a previously printed solution into PGSolver. This allows you to, for instance, verify a particular solution.

`--solverinfo`
: Prints out a list of all available solvers.

`--globallysolve` ⟨*solver*⟩ or `-global` ⟨*solver*⟩
: Chooses a particular global solver to solve the game at hand.

`--locallysolve` ⟨*solver*⟩ ⟨*node*⟩ or `-local` ⟨*solver*⟩ ⟨*node*⟩
: Chooses a particular local solver to solve the game at hand starting from a specified node.

`--args` ⟨*arguments*⟩ or `-x` ⟨*arguments*⟩
: Provides additional arguments to a solvers. You can get the full list of arguments for a particular solver by calling PGSolver with `-x "help"`.

`--generator` ⟨*generator*⟩ ⟨*arguments*⟩ or `-gen` ⟨*generator*⟩ ⟨*arguments*⟩
: Generates a game using one of the included generators directly for PGSolver to solve.

3.4.3 Output

An invocation of

```
~/pgsolver> bin/randomgame 10 10 2 4 | bin/pgsolver -global recursive
```

will typically create output on `stdout` like this:

```
Parsing ..................... 0.00 sec
Chosen solver 'recursive' ..... 0.00 sec

Player 0 wins from nodes:
  {0,2,5,6,7,8}
with strategy
  [0->6,2->2,5->0,6->2,7->7,8->8]

Player 1 wins from nodes:
  {1,3,4,9}
with strategy
  [1->9,3->9,4->3,9->9]
```

It reports the times it took to parse the input and to solve the game specified in this input as well as which solver has been used. Then it reports for both players the sets of winning regions in the game as a list of natural numbers (each node's index in the game) in ascending order. The positional strategies are reported as a list of pairs of the form x`->`y meaning that in node x, the player at hand should move to node y. This list is also sorted in ascending order w.r.t. x. Furthermore, the set of nodes given as x's in this list is exactly the set of nodes belonging to that player.

3.5 Specifying Parity Games

PGSolver expects the parity game it should solve or display to be given in its own specification language. There, a parity game consists of an optional *header line* and a list of *node specifications*. The header line tells PGSolver the highest occurring identifier in the game helping to speed up the parsing process. Each node specification contains an identifier of a node (a natural number), its priority, the player who owns the node, the list of its successors and, optionally, a symbolic name of the node. The format can

easily be described in EBNF.

$$
\begin{aligned}
\langle \textit{parity\_game} \rangle &::= [\texttt{parity}\ \langle \textit{identifier} \rangle\ \texttt{;}]\ \langle \textit{node\_spec} \rangle^{+} \\
\langle \textit{node\_spec} \rangle &::= \langle \textit{identifier} \rangle\ \langle \textit{priority} \rangle\ \langle \textit{owner} \rangle\ \langle \textit{successors} \rangle\ [\langle \textit{name} \rangle]\ \texttt{;} \\
\langle \textit{identifier} \rangle &::= \mathbb{N} \\
\langle \textit{priority} \rangle &::= \mathbb{N} \\
\langle \textit{owner} \rangle &::= \texttt{0} \mid \texttt{1} \\
\langle \textit{successors} \rangle &::= \langle \textit{identifier} \rangle\ (\texttt{,}\ \langle \textit{identifier} \rangle)^{*} \\
\langle \textit{name} \rangle &::= \texttt{"}\ (\text{ any ASCII string not containing `"'})\ \texttt{"}
\end{aligned}
$$

There must be whitespace characters between the following pairs of tokens: ⟨*identifier*⟩ and ⟨*priority*⟩, ⟨*priority*⟩ and ⟨*owner*⟩, ⟨*owner*⟩ and ⟨*identifier*⟩.

The *identifier* – in effect a natural number – that is given at the beginning of the specification after the keyword `parity` allows more efficient parsing. It should be the maximal identifier of a node in the game. If it is smaller, parsing will fail. If it is bigger then parsing will be successful and solving should be possible as well, but internally the game will be blown up with undefined nodes. For large games we recommend adding the `parity` ⟨*identifier*⟩ at the beginning. Note that currently, the old format without the size specification is still supported but may be dropped in a future version of PGSolver.

In order to give a precise semantics to this syntax we introduce the following notation. Let $G = (V, V_0, V_1, E, \Omega)$ be a parity game with $V \subseteq \mathbb{N}$, and $\alpha := (v, p, i, s) \in \mathbb{N} \times \mathbb{N} \times \{0,1\} \times \mathbb{N}^{+}$. Then $G + \alpha$ is defined as the parity game $(V', V_0', V_1', E', \Omega')$ where

$$
\begin{aligned}
V' &:= V \cup \{v\} \\
V_0' &:= V_0 \cup \begin{cases} \{v\} & \text{, if } i = 0 \\ \emptyset & \text{, otherwise} \end{cases} \\
V_1' &:= V_1 \cup \begin{cases} \{v\} & \text{, if } i = 1 \\ \emptyset & \text{, otherwise} \end{cases} \\
E' &:= (E \cap (V \setminus \{v\})^2) \cup \{(v, w) \mid s = \ldots, w, \ldots\} \\
\Omega'(w) &:= \begin{cases} p & \text{, if } w = v \\ \Omega(w) & \text{, otherwise} \end{cases}
\end{aligned}
$$

Now let $G_0 := (\emptyset, \emptyset, \emptyset, \emptyset, \perp)$ with $\perp$ being the empty function. Then we can recursively define the semantics of a parity game specification as follows. Let ϵ denote the empty

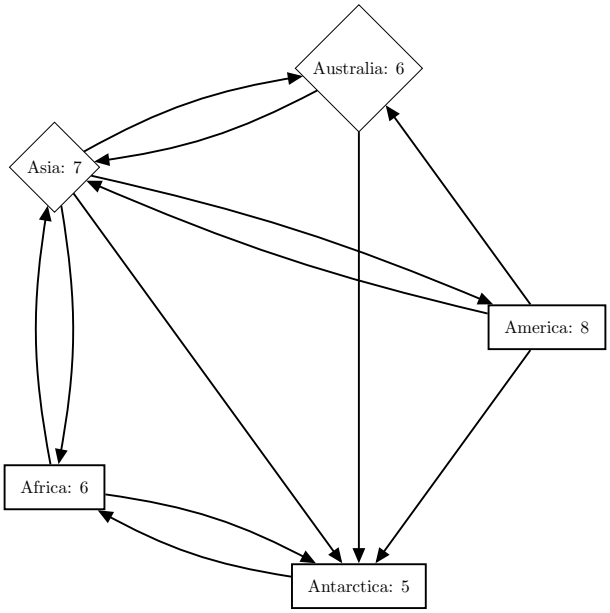


Figure 3.1: An example of a parity game.

list of node specifications.

$$\begin{aligned} [\![\epsilon]\!] &:= G_0 \\ [\![l\,(\texttt{vpisn;})]\!] &:= [\![l]\!] + (v, p, i, s) \\ [\![l\,(\texttt{vpis;})]\!] &:= [\![l]\!] + (v, p, i, s) \end{aligned}$$

Note that the EBNF definition of the syntax does not ensure well-formedness of a parity game. For instance, one can list identifiers in the successors list of a node that do not have node specifications for themselves. PGSolver assumes the input games to be well-formed in that sense. Otherwise it will terminate with an error message.

Example 6 The parity game depicted in Fig. 3.1 can, for example, be specified as follows.

```
parity 4;
0 6 1 4,2 "Africa";
4 5 1 0 "Antarctica";
1 8 1 2,4,3 "America";
3 6 0 4,2 "Australia";
2 7 0 3,1,0,4 "Asia";
```

Nodes belonging to player 0 are shown in diamond shape, the others in box shape. The label in each node is composed of the node's name and its priority. Note that symbolic names are optional.

Currently, parity games are stored as arrays of node specifications. The index in that array at which a node is stored, equals its identifier. This has two important and noteworthy consequences.

- As noted above, using identifiers multiply leads to overriding of nodes. If the parity game contains more than one node specification with the same identifier then the *last* node specification determines the properties of the node with that index.
- The size of the array that is allocated to store a parity game is always $n + 1$ where n is the maximal identifier occurring in the game's specification. To avoid unnecessary waste of space you should ensure that the nodes' identifiers in a parity game occupy a closed interval of the natural numbers of the form $\{0, 1, 2, \ldots, n\}$.

3.6 Viewing Parity Games

PGSolver can display parity games in two different ways: textually and graphically. The former can be invoked using the command-line option `-f`.

```
~/proj/pgsolver> bin/pgsolver -f
```

This results in a simple reprinting of the parity game to be solved at the end of PGSolver's usual output. The format is the same as the input format described in the previous section.

In order to display parity games graphically one needs the `graphviz` package, available for free from

```
http://www.graphviz.org/
```

PGSolver can create output in the `dot`-format which can be displayed using `dotty` from the `graphviz` package for example. The relevant command-line parameter is `-d` with a filename which tells PGSolver to write the `dot` code into that file after solving the game. Beware that you need to tell PGSolver explicitly (how) to solve the game. If you omit this, PGSolver will parse the input but not solve the game. However, this can be used to display the game as it is.

```
~/proj/pgsolver> bin/pgsolver -d graph.dot tests/test1.gm
```

Then, in order to view it, try

```
~/proj/pgsolver> dotty graph.dot
```

PGSolver can also display a game together with the winning information. This happens when you tell it to create `dot`-code for the game *and* tell it to solve it.

```
~/proj/pgsolver> bin/pgsolver -global recursive -d graph.dot tests/test1.gm
```

Again, the result can be viewed using

```
~/proj/pgsolver> dotty graph.dot
```

for example. However, now nodes and some edges are coloured according to the following specification.

- The winning region for player 0, i.e. all nodes from which he/she can win the game, is coloured green. The winning region for player 1 is coloured red.
- An edge coloured green belongs to the positional strategy for player 0 that is winning on his/her winning region. An edge coloured red belongs to player 1's respective winning strategy.

An example display of a solved parity game is given on the title page.

3.7 Additional Tools

3.7.1 Obfuscator

This tool randomly permutes the identifiers of nodes as well as the order of nodes' successors in a parity game. This is useful to confuse algorithms which perform very well on certain benchmarks because of the order in which nodes and successors are given.

Example 7 Take the ladder game of index 4, as introduced in Sect. 4.2.2 below. In PGSolver's specification format it looks like this.

```
~/pgsolver> bin/laddergame 4
parity 8;
0 0 0 1,2;
1 1 1 2,3;
2 0 0 3,4;
3 1 1 4,5;
4 0 0 5,6;
5 1 1 6,7;
6 0 0 7,0;
7 1 1 0,1;
```

Each player owns 4 out of the eight nodes. Since all nodes have outdegree 2, there are $2^4 = 16$ strategies for each of the player. In general, there are 2^n different strategies in the game of index n but only one of them is a winning strategy. However, the winning strategy for player 0 for example is an obvious one in this case. It consists of always choosing the first node in the list of successors. Equally, the winning strategy for player 1 consists of always choosing the last node in each list. This is hardly a good hiding of the winning strategies.

Obfuscator employs a random generator to destroy the order of the successors and the order in which the nodes are given. A typical obfuscation of the ladder game of index 4 would be the following.

```
~/pgsolver> bin/laddergame 4 | bin/obfuscator
parity 8;
0 1 1 6,1;
1 1 1 7,5;
2 1 1 3,0;
3 0 0 0,6;
4 0 0 2,3;
5 1 1 4,2;
6 0 0 1,7;
7 0 0 5,4;
```

Obfuscator is automatically built through `make obfuscator`, `make tools`, or `make all`. It takes a parity game on `stdin` in the specification format described above and returns the permuted game on `stdout`. The currently understood command-line parameters are:

`--disablenodeobfuscation` or `-dn`
: Disables the obfuscation of the ordering of nodes.

`--disableedgeobfuscation` or `-de`
: Disables the obfuscation of the ordering of the edges.

3.7.2 Compressor

This tool compresses a given parity game s.t. the winning sets and strategies of the resulting game equal those of the original game.

Example 8 Take the following game and suppose it is stored in a file called `test.gm`.

```
~/pgsolver> cat/test.gm
parity 12;
0 1 0 5;
2 1 0 7;
4 2 1 5,0;
6 2 1 5,7,2;
5 2 0 4,6,9;
7 2 0 7,6;
8 4 0 9;
10 4 0 9,11;
9 3 1 8,10,5;
11 3 1 10,11,7;
```

It is possible to push priorities along edges overriding smaller priorities. It is also possible to reassign priorities just keeping parities and the order inbetween them. This leads to the following game:

```
~/pgsolver> cat test.gm | bin/compressor -pr -pp
parity 11;
0 0 0 5;
2 0 0 7;
4 0 1 5,0;
5 0 0 4,6,9;
6 0 1 2,5,7;
7 0 0 6,7;
8 2 0 9;
9 1 1 5,8,10;
10 2 0 9,11;
11 1 1 7,10,11;
```

It is also possible to compress the original game's identifier space eliminating unused node indetifiers.

```
~/pgsolver> bin/jurdzinskigame 1 2 | bin/compressor --nodes
parity 10;
0 1 0 3;
1 1 0 5;
2 2 1 0,3;
3 2 0 2,4,7;
4 2 1 1,3,5;
5 2 0 4,5;
6 4 0 7;
7 3 1 3,6,8;
8 4 0 7,9;
9 3 1 5,8,9;
```

Note that the original game does not possess a node called 1 for instance.

Compressor is automatically built through `make compressor`, `make tools`, or `make all`. It takes a parity game in the specification format described above on `stdin` or from a given file, and returns the compressed game on `stdout`. The currently understood command-line parameters are:

⟨*filename*⟩
: Tells the tool to look for the specification of the parity game to transform in the file ⟨*filename*⟩.

`--priorities` or `-pr`
: It reduces the priority of each node preserving their parities as well as the relation $\leq$ among priorities of nodes within an SCC of the game graph.

`--priopropagation` or `-pp`
: It tries to reduce the overall number of priorities occurring in the game by pushing

priorities along edges of the game graph overriding smaller ones. If all nodes leading to a node v have a greater priority than the node itself, the priority of this node is altered to the least predecessing priority. Similarly the priority of a node is adapted if all successors of a node have a greater priority than the node itself. This process is iterated until stability is reached.

Obviously, this process may destroy the preservation of parities and the less-or-equals relation among priorites described above.

`--antipropagation` or `-ap`
: It tries to set the priority of as many nodes as possible to zero. If all nodes leading to a node v have a greater priority than the node itself, the priority of this node can be set to zero. Similarly the priority of a node is adapted if all successors of a node have a greater priority than the node itself. This process is iterated until stability is reached.

`--fakealt` or `-fa`
: Keep fake alternation w.r.t. priorities. If this option is enabled the compaction of priorities never assigns the same priority to nodes that had different priorities before, even if there is no other priority occurring inbetween and both priorities in question are of the same parity.

`--wholegame` or `-pp`
: The compression of priorities is usually done independently for each SCC of the game graph. This switch causes the graph's SCC decomposition to be ignored. This will in general result in a worse compression rate.

`--nodes` or `-no`
: Performs a compression of the space of identifiers of a parity game, resulting in a possible down-shift of identifiers.

`--minmaxswap` or `-mm`
: Transforms a min-parity game into a max-parity one and vice versa.

3.7.3 Combinator

This tool combines two or more parity games into one single parity game by shifting the node numbers. Neither additional edges between the games are added nor other modifications are done.

Example 9 We just create two small random games and combine them to see what happens.

```
~/pgsolver> bin/randomgame 5 5 2 3 | tee game1.gm
parity 5;
```

```
0 1 1 3,4 "0";
1 4 0 1,4 "1";
2 5 0 0,2,4 "2";
3 1 0 0,2,4 "3";
4 0 1 0,1,3 "4";

~/pgsolver> bin/randomgame 5 5 2 3 | tee game2.gm
parity 5;
0 4 0 2,3 "0";
1 5 0 0,4 "1";
2 4 1 0,2,3 "2";
3 4 0 2,4 "3";
4 4 1 0,3 "4";
```

A combination of both games can be achieved as follows:

```
~/pgsolver> bin/combine game1.gm game2.gm
parity 10;
0 1 1 3,4 "0";
1 4 0 1,4 "1";
2 5 0 0,2,4 "2";
3 1 0 0,2,4 "3";
4 0 1 0,1,3 "4";
5 4 0 7,8 "0";
6 5 0 5,9 "1";
7 4 1 5,7,8 "2";
8 4 0 7,9 "3";
9 4 1 5,8 "4";
```

Combine is automatically built through `make combine`, `make tools`, or `make all`. It takes arbitrarly many parity games as command-line parameters and returns the combined game on `stdout`. There are no other command-line parameters.

3.7.4 Transformer

This tool transforms a given parity game into an (somehow) equivalent parity game fulfilling certain properties – depending on the user's command line specification. The transformed parity game can be associated with the original game s.t. winning sets and strategies can be easily back-transformed.

Example 10 Again, take the ladder game of index 3, as introduced in Sect. 4.2.2 below. In PGSolver's specification format it looks like this.

```
~/pgsolver> bin/laddergame 3
parity 6;
0 0 0 1,2;
```

```
1 1 1 2,3;
2 0 0 3,4;
3 1 1 4,5;
4 0 0 5,0;
5 1 1 0,1;
```

The game is obviously total but not choice-alternating as node 0 is directly connected to node 2 with both of them belonging to player 0. A choice-alternation version of this game can obtained as follows:

```
~/pgsolver> bin/laddergame 3 | bin/transformer --alternating
parity 12;
0 0 0 1,6;
1 1 1 2,7;
2 0 0 3,8;
3 1 1 4,9;
4 0 0 5,10;
5 1 1 0,11;
6 0 1 2;
7 0 0 3;
8 0 1 4;
9 0 0 5;
10 0 1 0;
11 0 0 1;
```

Transformer is automatically built through `make transformer`, `make tools`, or `make all`. It takes a parity game in the specification format described above on `stdin` or from a file, and returns the transformed game on `stdout`. The currently understood command-line parameters are:

⟨*filename*⟩
: Tells the tool to look for the specification of the parity game to transform in the file ⟨*filename*⟩.

`--total` or `-to`
: Perform a totality transformation on the given parity game.

`--alternating` or `-al`
: Perform a choice-alternation transformation on the given parity game.

`--singlescc` or `-scc`
: Perform a simple transformation that results in a single-scc-parity game. The winning sets and strategies of the original game and the transformed game match.

`--prioalignment` or `-pa`
: Performs a transformation s.t. the parity of each node of the resulting game matches its player unless its priority is zero.

`--dummynodes` or `-dn`
: Performs a transformation that divides each edge by an additional node with priority 0.

`--uniquizeprios` or `-up`
: Performs a transformation of the occurring priorities s.t. every occurring priority occurs only once.

`--cheapescapecycles` or `-ce`
: Adds two low-priority cycles c_0 and c_1 to the game that are profitable for either one of the player (c_0 for player 0 and c_1 for player 1). All nodes in the original belonging to player 0 get additional edges leading to c_1 and all nodes belonging to player 1 get additional edges leading to c_0.

`--bouncingnode` or `-bn`
: Replaces every self-cycle by a bouncing node.

`--increasepriorityoccurrence` or `-ip`
: Increases the occurrence of all priorities by one by simply inserting a long cycle that is not connected to the rest of the game.

`--antiprioritycompactation` or `-ap`
: Adds additional nodes that prohibits any priority compactation.

Note that the command-line parameters that are specified are carried out in the same ordering as they appear in the call of the transformer tool. It is also possible to specify a parameter more than once.

3.7.5 Benchmark Tool

This tool helps the user to carry out benchmarks and to compare the different solvers with each other. You can specify a list of games that are to be benchmarked with a list of solver that also has to be specified.

You can either choose to receive a verbose output showing the exact timing of each case or a simple line containing data that can be easily parsed by `gnuplot`. The latter output option is usually to be used in combination with a shell script creating a whole benchmark series.

Example 11 The following example shows how to benchmark the recursive and the strategy improvement algorithm on a random game with 1000 nodes. To smooth multi-tasking interference factors, each test case is carried out 3 times.

```
~/pgsolver> bin/randomgame 1000 1000 2 4 | bin/benchmark -re '' -si '' -t 3
Benchmarking stratimprove...
Game # 0, Iteration #1...0.02 sec
Game # 0, Iteration #2...0.02 sec
Game # 0, Iteration #3...0.02 sec
Finished. Best: 0.02 sec  Avg: 0.02 sec  Worst: 0.02 sec

Benchmarking recursive...
Game # 0, Iteration #1...0.01 sec
Game # 0, Iteration #2...0.00 sec
Game # 0, Iteration #3...0.01 sec
Finished. Best: 0.00 sec  Avg: 0.01 sec  Worst: 0.01 sec

+-------------------------------------------------------------+
| Benchmark Statistics                                        |
+-------------------------------------------------------------+
| Solver       |         Best |      Average |        Worst |
+-------------------------------------------------------------+
| recursive    |     0.00 sec |     0.01 sec |     0.01 sec |
| stratimprove |     0.02 sec |     0.02 sec |     0.02 sec |
+-------------------------------------------------------------+
```

Benchmark is automatically built through `make benchmark`, `make tools`, or `make all`. It takes one ore more parity games in the specification format described above on `stdin` or from files, and carries out the benchmark. The currently understood command-line parameters are:

⟨*filename*⟩
: Tells the tool to look for the specification of the parity game in the file ⟨*filename*⟩.

`--`⟨*solver*⟩ `'solver options'`
: Benchmark the specified solver. All solvers that are compiled into PGSolver are available here. Note: even if you don't want to pass any options to the solver, you still need to add ".

`--silent` or `-s`
: Only print the final statistics on `stdout`.

`--gnuplotformat` or `-gp`
: Only print a gnuplot-compatible line.

`--timeout` ⟨*timeout*⟩ or `-to` ⟨*timeout*⟩
: If one of the solvers requires more than t seconds on one of the test cases, the solver is ignored for the rest of the series. There is no timeout by default.

`--name` ⟨*title*⟩ or `-n` ⟨*title*⟩
: Specifies the title of the benchmark that is to be printed in the final statistics.

`--times` ⟨n⟩ or `-t` ⟨n⟩
: Specifies the number of iterations a single test case is carried out per algorithm. By default, n is 10.

`--`⟨*optimisation − option*⟩
: All optimisation options that are available to `pgsolver` are also available here.

4 Benchmarks

The PGSolver library contains a few programs that create benchmarks for the parity game solvers. They are built using the command

```
~/pgsolver> make generators
```

and reside afterwards in the subdirectory `bin`.

Additionally, it is possible to compile all generators directly into PGSolver by setting the variable `LINKGENERATORS` to `YES` in the `./Config`-file. This has the advantage that the process of generating and solving a game is not slowed down by formatting the game first and parsing it afterwards again into the same data structure.

4.1 Random Games

4.1.1 A Naïve Model

Randomly generated parity games are the simplest form of benchmarking utility to assess the performance of game solvers. The ones used here are parametrized by four parameters: the number n of nodes in the game, the highest priority p, a lower and an upper bound l and u on the out-degree of each node. The call

```
~/pgsolver> bin/randomgame 1000 200 2 5
```

creates a random game with 1000 nodes as follows: for each node v,

- the priority $\Omega(v)$ is uniformly chosen among $\{0, \ldots, 200\}$;
- node v belongs to player 0 with probability 0.5 and therefore to player 1 with probability 0.5 as well;
- a number d is uniformly chosen in the range $\{2, \ldots, 5\}$, and d pairwise different nodes are uniformly chosen as v's successors.

Beware that `bin/randomgame` may crash or not terminate if the lower bound on the outdegree is greater than the upper bound or that one is greater than the number of nodes.

4.1.2 Clustered Random Games

Almost every game created by the random generator above consists of a single big SCC and many adhesive chains.

Therefore this generator uses an other random model that accomplishes to bring a bit more structure into its instances. It depends on nine parameters: The number of nodes n, the highest possibly occurring priority p, an out-degree range $(l; h)$, a recursion depth r, a recursion breadth range $(a; b)$ and an interconnection range $(x; y)$. An instance of $\mathcal{G}^c_{(n,r)}$ with $c = (p, l, h, a, b, x, y)$ can be generated as follows. If $r = 0$ or $a > n$ simply return a randomly generated game with n nodes, highest priority p and out-degree bounds l and h as above. If $r > 0$ and $a \leq n$,

- a number d is uniformly chosen in the range $\{a, \ldots, \min(b, n)\}$;
- d numbers $k_0, \ldots, k_{d-1}$ are uniformly chosen in the range $\{0, \ldots, n-1\}$ s.t. $\sum_{i=0}^{d-1} k_i = n$;
- d instances $G_0, \ldots, G_{d-1}$ are chosen, where G_i is chosen from $\mathcal{G}^c_{(k_i,r-1)}$ (we assume the nodes of all G_i to be pairwise different);
- a game G is created by the union of all G_i;
- a number e is uniformly chosen in the range $\{x, \ldots, y\}$;
- e additional uniformly chosen edges in G are added to G;
- the game G is returned.

Basically, this recursive approach creates a topological tree structure that is partially intercepted by adding the additional interconnection edges.

A clustered random game with 1000 nodes, highest priority 2, out-degree bounds 2 and 5, recursion depth 3, recursion breadth between 4 and 6 and interconnection degree between 11 and 22 is created using the call

```
~/pgsolver> bin/clusteredrandomgame 1000 200 2 5 3 4 6 11 22
```

4.1.3 Steady Random Games

The Steady Random Generator tries to circumvent many universal optimisations in order to particularly benchmark the backend solvers. The Steady Random Generator is parametrized by five parameters: the number n of nodes (and different priorities) in the game, a lower and an upper bound and on the out-degree of each node and a lower and an upper bound and on the in-degree of each node. The call

```
~/pgsolver> bin/steadygame 1000 2 4 3 5
```

creates a random game with 1000 nodes as follows: for each node v,

- the priority $\Omega(v)$ is simply v;

- node v belongs to player 0 with probability 0.5 and therefore to player 1 with probability 0.5 as well;
- a number d is uniformly chosen in the range $\{2, \ldots, 4\}$, and d pairwise different nodes are uniformly chosen in v's subgame as v's successors;
- a number d is uniformly chosen in the range $\{3, \ldots, 5\}$, and an attempt is made to ensure that v has d pairwise different predecessors.

All edges are iteratively determined as long as there are at least two different nodes with one of them violating a lower bound and the other one being below the opposite upper bound uniformly between two such nodes.

4.2 Special Graph Structures

4.2.1 Clique Games

The clique game of order n is $G_n = (\{0, \ldots, n-1\}, \{0, 2, 4, \ldots\}, \{1, 3, 5, \ldots\}, E, \Omega)$ with $E = \{(v, w) \mid v \neq w\}$ and $\Omega(v) = v$. It forms a clique in a directed graph without self-cycles. Clique games exhibit a large amount of cycles in the game which may pose difficulties for certain solvers. Note that adding self-cycles to the nodes will result in easy solving because self-cycles are partial winning strategies in this case, and then all other nodes would lie in the attractor of these winning regions.

The clique game of order 50 for instance is generated as follows.

```
~/pgsolver> bin/cliquegame 50
```

The clique game of order 50 with self-cycles can also be generated.

```
~/pgsolver> bin/cliquegame 50 self
```

4.2.2 Ladder Games

The name of these games is derived from their structure which is reminiscent of a ladder. The *ladder game* of index n is $G_n = (V, V_0, V_1, E, \Omega)$, defined as follows.

- $V = \{0, \ldots, 2n-1\}$,
- $V_0 = \{0, 2, 4, \ldots, 2n-2\}$, $V_1 = \{1, 3, 5, \ldots, 2n-1\}$,
- $E = \{(v, w) \mid w \equiv_{2n} v + i \text{ for some } i \in \{1, 2\}\}$,
- $\Omega(v) = v \mod 2$.

where $\equiv_{2n}$ means equality modulo $2n$.

It is not hard to see that player 0 wins on V_0 and player 1 wins on V_1 since both can stay within these regions and only see the priorities 0, resp. 1.

Ladder games are chosen as benchmarks because in G_n there are 2^n many positional strategies for each player but only one of them is a winning strategy. Not surprisingly, ladder games are very difficult to solve for the strategy guessing heuristic for example.

The ladder game of index 19 is created using the call

```
~/pgsolver> bin/laddergame 19
```

4.3 Hard Games for Particular Algorithms

4.3.1 Jurdziński Games

Jurdziński has defined a family of games on which the small progress measures algorithm exhibits an exponential running time [Jur00]. The Jurdziński game $J_{d,w}$ of depth d and width w forms a rectangle of dimension $(2d+1) \times (2w)$. For instance, $J_{50,100}$ is generated by calling

```
~/pgsolver> bin/jurdzinskigame 50 100
```

4.3.2 Recursive Ladder Games

The recursive ladder games form a family on which the recursive algorithm exhibits an exponential running time. The recursive ladder game of index n has size $5 \cdot n$. The recursive ladder game of index 4 results in 20 nodes and is generated by calling

```
~/pgsolver> bin/recursiveladder 4
```

4.3.3 Exponential Strategy Improvement Games

The exponential strategy improvement games form a family on which both strategy improvement algorithms exhibit an exponential running time (in the index of the game) [Fri09]. The exponential strategy improvement game of index n has size $14 \cdot n + 11$. A game of index 4 results in 67 nodes and is generated by calling

```
~/pgsolver> bin/expstratimpr 4
```

In order to verify that the strategy iteration indeed requires exponential runtime on these games, you need to either disable all generic optimisations or transform the game using the `transformer` tool with the parameters `-ss -ap -bn`.

4.3.4 Model Checker Ladder Games

The model checker ladder games form a family on which the model checker algorithm exhibits an exponential running time. The model checker ladder game of index n has size $4 \cdot n$. A model checker ladder game of index 3 results in 12 nodes and is generated by calling

```
~/pgsolver> bin/modelcheckerladder 3
```

4.4 Verification Problems

4.4.1 Fairness of an Elevator System

We encode a simple fairness verification problem as a parity game. States of a transition system modelling an *elevator* for n floors are of type $\{1,\ldots,n\} \times \{\mathtt{o},\mathtt{c}\} \times (\bigcup\{Perm(S) \mid S \subseteq \{1,\ldots,n\})$. The first component describes the current position of the elevator as one of the floors. The second component indicates whether the door is *open* or *closed*. The third component – a permutation of a subset of all available floors – holds the *requests*, i.e. those floors that should be served next. The transitions on these are as follows.

- At any moment, any request or none can be issued. For simplicity reasons, we assume that at most one floor is added to the requests per transition. Note that nondeterministically, no request can be issued, and a request for a certain floor that is already contained in the current requests does not change them.
- If the door is open then it is closed in the next step, the current floor does not change.
- If it is closed, the elevator moves one floor (up or down) into the direction of the first request. If the floor reached that way is among the requested ones, the door is opened and that floor is removed from the current requests. Otherwise, the door remains closed.

We consider two different implementations of this elevator model: the first one stores requests in FIFO style, the second in LIFO style. The games G_n (with FIFO), resp. G'_n (with LIFO) result from encoding the model checking problem for this transition system and the CTL* formula $\mathtt{A}(\mathtt{GF}\mathit{isPressed} \rightarrow \mathtt{GF}\mathit{isAt})$ as a parity game [Sti95]. Proposition *isPressed* holds in any state s.t. the request list contains the number n, and *isAt* holds in a state where the current floor is n. Hence, this formula requires all runs of the elevator to satisfy the following fairness property: if the top floor is requested infinitely often then it is being served infinitely often. It can easily be formulated in the modal μ-calculus using a formula of size 11 and alternation depth 2 (of type ν–μ–ν). Hence the resulting parity games have constant index 3. Note that G_n encodes a positive instance of the model checking problem whereas G'_n encodes a negative one. The game G_4 is generated by calling

```
~/pgsolver> bin/elevators 4
```

and the game G'_5 by calling

```
~/pgsolver> bin/elevators -u 5
```

4.5 Particular Problems Encoded as Parity Games

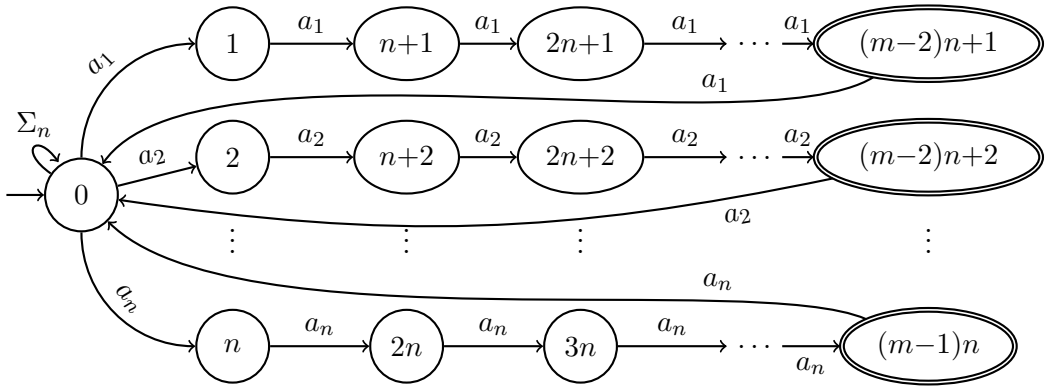
4.5.1 Towers of Hanoi

The Towers of Hanoi problem can be seen as a reachability game in the graph of game configurations. The problem of size n has n disks that are to be moved from the first to the third rod. A problem of size 3 is generated by calling

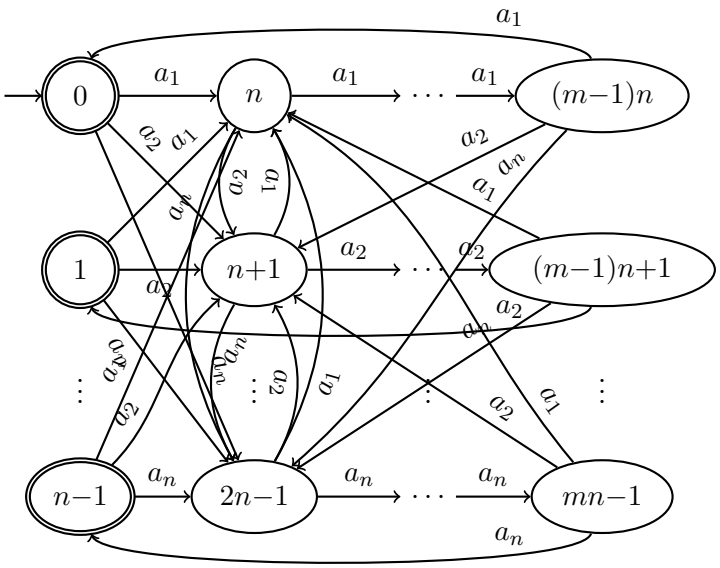
```
~/pgsolver> bin/towersofhanoi 3
```

4.5.2 Language Inclusion Between an NBA and a DBA

Let $n, m \geq 2$, $\Sigma_n = \{a_1, \ldots, a_n\}$ and consider the language $L_{n,m}$ of all words over the alphabet Σ_n that contain infinitely many occurrences of $(a_i)^m$ for some $i\{1, \ldots, n\}$. This language is accepted by the following NBA $\mathcal{A}_{n,m}$.



It is also accepted by the following deterministic Büchi automata $\mathcal{B}_{n,m}$.



The synchronous product of $\mathcal{A}_{n,m}$ and $\mathcal{B}_{n,m}$ is a parity game $\mathcal{G}_{n,m}$ in which every node is owned by player 1, and the priority of a node (q, p) is given as

$$\Omega(q,p) \;=\; \begin{cases} 2 & \text{, if } p \text{ is accepting,} \\ 1 & \text{, if } p \text{ is not accepting and } q \text{ is accepting,} \\ 0 & \text{, otherwise.} \end{cases}$$

This is the standard encoding of fair simulation as a parity condition [HKR02, HLL13]. This game encodes the language inclusion problem between an NBA and a DBA. In this case, the game is entirely won by player 0.

The game $\mathcal{G}_{6,9}$ is generated by

```
~/pgsolver> bin/langincl 6 9
```

The second parameter is optional. If it is not given then $m = 2$ is chosen by default.

5 Developer's Guide

The purpose of this chapter is to provide enough insight into the structure of the PGSolver tool such that it becomes possible to extend it with a new algorithm. We assume reasonable OCaml programming skills and some familiarity with parity games, though.

Implementing a new algorithm requires two steps only.

1. Create an OCaml source code file that contains the solver. This is just a function which takes a parity game and returns a partition of (a subset of) its node set into winning regions and winning strategies for the two players. It must reside in a new module.

2. Integrate this function into the main program such that the new solver can be used by giving PGSolver a certain command-line argument. Then adjust the `SolverList` such that the new module is compiled with the rest of the tool.

5.1 Structure of the Source Code

In the following, all references to files are given relatively to the directory into which PGSolver was unpacked during the installation process. There, the subdirectory `src` is the important one for this step. It contains the source files of all the modules that make up the PGSolver library. They are organised into the following subdirectories respectively. Note that, when we speak of *modules* we often refer to files only that implicitly count as a module in OCaml.

`generators` contains modules that make up the benchmark generators;

`tools` contains tools for the creation or manipulation of parity games;

`paritygame` contains modules that define parity games as a data structure and provide functionality for that, e.g. in the form of procedures that compute SCC decompositions etc.;

`pgsolver` contains modules for the actual program `pgsolver`;

`solvers` contains the parity game solvers, each in a separate module.

5.2 Data Types and Functions for Manipulating Parity Games

The signature file

`src/paritygame/paritygame.mli`

contains the definition of parity games as an abstract data structure. A parity game is internally stored as an array of node definitions but this may or may not change in the future in favour of a more efficient implementation. There are also (abstract) data types for representing nodes, players, priorities, sets of nodes, strategies, solutions and solutions, which will be introduced in detail in the following.

5.2.1 Nodes

The type of a node in a paritygame is one of the few types that are not abstract. For sake of efficiency, a node is simply an integer value:

```
type node = int
```

Beware that the type node may become abstract in the future, so try to avoid to arithmetic etc. on nodes.

There is currently only one function that is specific to nodes; it turns a node into a string for output purposes.

```
val nd_show : node -> string
```

5.2.2 Sets of Nodes

Many algorithms manipulate sets of nodes, for instance they take the set of all nodes with highest priority etc. For such purposes, `paritygame.mli` provides an abstract data type and functions for storing and manipulating sets of nodes.

```
type nodeset
```

Currently, this type is implemented as a simple ordered list without duplicates. The functions for creation and manipulation follow the standard `Set` interface. Use

```
val ns_empty   : nodeset
val ns_make    : node list -> nodeset
```

to create a set of nodes, either initially empty or from a given list of nodes. It is possible to add nodes to existing sets or deletes nodes from them using

```
val ns_add     : node -> nodeset -> nodeset
val ns_del     : node -> nodeset -> nodeset
```

In the latter case `ns_del v vs` returns vs if `v` is not a member of `vs`.

Desctruction of node sets is implemented as

```
val ns_nodes   : nodeset -> node list
```

which returns all the elements of a set of nodes as a list.

For tests on emptiness and on membership of a particular node in a set use

```
val ns_isEmpty : nodeset -> bool
val ns_elem    : node -> nodeset -> bool
```

There are more general tests, for instance if a node set includes some node with a particular property, or if all of them do so:

```
val ns_exists  : (node -> bool) -> nodeset -> bool
val ns_forall  : (node -> bool) -> nodeset -> bool
```

It is possible to extract nodes from a node set, either some with a particular property (`ns_find`) or a random one (`ns_some`) or the first, resp. last in some canonical order (`ns_first`, `ns_last`), or the one that is maximal w.r.t. a total order (`ns_max`).

```
val ns_find    : (node -> bool) -> nodeset -> node
val ns_some    : nodeset -> node
val ns_first   : nodeset -> node
val ns_last    : nodeset -> node
val ns_max     : nodeset -> (node -> node -> bool) -> node
```

The cardinality of a set of nodes is computed by

```
val ns_size    : nodeset -> int
```

Finally, there are the typical iterator functions

```
val ns_fold    : ('a -> node -> 'a) -> 'a -> nodeset -> 'a
val ns_iter    : (node -> unit) -> nodeset -> unit
val ns_map     : (node -> node) -> nodeset -> nodeset
val ns_filter  : (node -> bool) -> nodeset -> nodeset
```

that behave as one would expect. Note that there is no control over the order in which nodes in a set are processed in `ns_iter f vs` or `ns_fold f x vs`; hence for a uniquely determined result the function `f` should be independent of that order.

5.2.3 Players and Priorities

There are two types representing players and priorities.

```
type player
type priority = int
```

Players are abstract but priorities are just integers. Only non-negative integers should be used as priorities.

There are constants defining the two players of a parity game. Some algorithms may have to defer the decisio about who wins which node in a game; for such cases there is also a constant representing an undefined player. At last, there is a function which can be used to obtain a random player with even distribution between `plr_Even` and `plr_Odd`.

```
val plr_Even   : player
val plr_Odd    : player
val plr_undef  : player
val plr_random : unit -> player
```

There are a few functions realising some minor arithmetic on the data type `player`, for instance computing a player's opponent, for obtaining a string representation, or for applying a function to both players.

```
val plr_opponent : player -> player
val plr_show     : player -> string
val plr_iterate  : (player -> unit) -> unit
```

There are some functions that link players and priorities, for instance by checking whether a priority is good for some player in the sense that even priorities are good for `plr_Even` and odd ones for `plr_Odd`. Likewise it is possible to obtain the benefitting player directly.

```
val prio_good_for_player : priority -> player -> bool
val plr_benefits         : priority -> player
```

These functions essentially test the parity of an integer value; there are also functions to do this directly.

```
val odd: priority -> bool
val even: priority -> bool
```

It is unlikely, yet not impossible, that the type `priority` will become abstract in the future. Hence, it is advisable to only manipulate priorities through these functions.

5.2.4 Parity Games

The most important type in this module is the data type for parity games.

```
type paritygame
```

Note that as of version 4.0, the implementation of this type is no longer visible to the other modules, for reasons of code reliability. There are, however, several functions that allow parity games to be created, manipulated and used.

There are three ways for creating a fresh parity game.

```
val pg_create : int -> paritygame
val pg_init   : int ->
                (node -> (priority * player * node list * string option)) ->
                paritygame
val pg_copy   : paritygame -> paritygame
```

The former two expect the number of nodes of the game as a parameter. In case of `pg_create`, the resulting game has no edges, the owner of each node is `plr_undef` and the priorities cannot be assumed to be non-negative integer values. The second function can be used to create a game and fill it with some structure. Calling `pg_init n f` will return a parity game with `n` nodes such that the result of calling `f i` contains the priority, owner, list of successor nodes and possibly a name of the `i`-th node in the game. Function `pg_copy` creates a fresh copy of an existing game which may be handy for algorithms that manipulate games and have to revert certain changes.

Given a parity game, such information about a node can be extracted using the following functions.

```
val pg_get_priority     : paritygame -> node -> priority
val pg_get_owner        : paritygame -> node -> player
val pg_get_successors   : paritygame -> node -> nodeset
val pg_get_predecessors : paritygame -> node -> nodeset
```

It is possible to search for the node name (as an integer, resp. value of type `node`) given its description.

```
val pg_find_desc  : paritygame -> string option -> node
```

There are two function for finding nodes according to their priorities.

```
val pg_prio_nodes              : paritygame -> priority -> node list
val pg_get_selected_priorities : paritygame ->
                                 (priority -> bool) ->
                                 priority list
```

A parity game can be modified inplace by setting, resp. changing the priority or string description or owner of a node, or by adding an edge between two nodes.

```
val pg_set_priority : paritygame -> node -> priority -> unit
val pg_set_owner    : paritygame -> node -> player -> unit
val pg_get_desc     : paritygame -> node -> string option
val pg_set_desc     : paritygame -> node -> string option -> unit
val pg_get_desc'    : paritygame -> node -> string
val pg_set_desc'    : paritygame -> node -> string -> unit
val pg_add_edge     : paritygame -> node -> node -> unit
```

It is also possible to delete a set of nodes given as a list, or to remove a single edge or a set of edges, given as a list of node pairs. These deletions also work inplace.

```
val pg_remove_nodes : paritygame -> node list -> unit
val pg_del_edge     : paritygame -> node -> node -> unit
val pg_remove_edges : paritygame -> (node * node) list -> unit
```

Note that using any of these can result in a game in which a node has no successor anymore. Deleting undefined nodes or non-existing edges results in no change.

Some structural information about a parity game can be obtained, namely the size (i.e. number of nodes), the number of edges, or the index (i.e. number of different priorities).

```
val pg_size          : paritygame -> int
val pg_node_count     : paritygame -> int
val pg_edge_count     : paritygame -> int
val pg_get_index      : paritygame -> int
val pg_get_priorities : paritygame -> priority list
```

The latter returns a list of all the priorities that are being used in the game.

There is a subtle difference between `pg_size` and `pg_node_count`. The latter only counts nodes that are defined, so for instance `pg_size (pg_create 2)` will return `2`, while `pg_node_count (pg_create 2)` will return `0`. Checking whether a node is defined can be done using

```
val pg_isDefined : paritygame -> node -> bool
```

Finally, there are iterator functions for parity games that traverse through all defined nodes, resp. all existing edges.

```
val pg_iterate      : (node ->
                       (priority * player * nodeset * nodeset * string option) ->
                       unit) -> paritygame -> unit
val pg_edge_iterate : (node -> node -> unit) -> paritygame -> unit
```

5.2.5 Solutions and Strategies

A solution of a parity game is a mapping of nodes to players, determining who wins the game starting in that particular node. A (memory-less) strategy (for both players) is a set of edges, including exactly one edge for each node that is won by the player who owns the node. The interface `paritygame.mli` provides two data structures for solutions and strategies which are based on the fact that nodes are non-negative integer values.

```
type solution = player array
type strategy = node array
```

A solution with value `plr_Even` at position `v` for instance represents the fact that player Even wins node `v`. Such arrays can contain the value `plr_undef`.

An array of type `strategy` can contain the value `nd_undef`, representing the fact that the strategy decision at that position is either unknown or unnecessary because the node is not won by its owner.

Note that an array of type `strategy` contains, in effect, two positional strategies – one for each player.

There is no mechanism that links a solution or strategy array to a particular parity game. It is the programmer's task to ensure that the solution for one game is not used for another game. This could result in runtime errors like array access outside their bounds.

5.3 Implementing a New Solver

We will exemplarily describe how to implement and integrate a new algorithm that solves parity games. There is more than one way to do so but there is a unique easiest way which we will follow here.

Suppose you have finally come up with a deterministic polynomial time algorithm. Surely this cannot miss in the PGSolver library. It is also quite probable that the algorithm is complex – for otherwise someone else would have found it before. Therefore you cannot wait for us to implement it. By the time we have understood all the algorithm's subtleties our programming skills will have bitten the dust. In that case you better do it yourself.

Find an expressive name for your algorithm that will be used to create a module and also later a command-line parameter for the `pgsolver` program. Say *Deterministic Polynomial Time Solver* was a good name. Then create two files in the `src/solvers` subdirectory, for example called

```
detpoly.ml
detpoly.mli
```

The signature file `detpoly.mli` must look like this.

```
val solve : Paritygame.paritygame ->
              Paritygame.solution * Paritygame.strategy
```

The name of the function is actually irrelevant, but calling it `solve` is not a bad idea. Its type is mandatory though. It must take a parity game and return a pair of a solution and a strategy using the data types described above.

Now sit down and hack the code for the implementation of your solver into the file `detpoly.ml`. In the simplest form this will look like

```
open Paritygame ;;
let solve game = ...
```

However, note that this function will be called in the main program, and as the argument `game` it will receive the parity game from the input. Hence, this does not automatically make use of the universal solver described in Sect. 2.4. This is because one cannot guarantee that the universal optimisations compression, SCC decomposition and solving of special cases speed up *any* solver.

If your algorithm turns out to be that quick you might not want the universal optimisations in which case you can skip the following description of how to employ the universal solver with your new algorithm as a backend and continue with the next section.

So you are still with us – thanks for the trust you have in our universal solver! Now, you will still have to implement your algorithm in a function of type `Paritygame.paritygame -> Paritygame.solution * Paritygame.strategy` but it may be better to choose a different name. For instance, create a function

```
let my_solver game = ...
```

in `detpoly.ml` that contains the implementation of your solver. This may now assume that the argument `game` consists of a single SCC only such that the priorities of at least two nodes have different parities and it is not just one player who has real choices in all the nodes in this game.

Now all you have to do is to make sure that the `solve`-function declared in `detpoly.mli` calls the universal solver using your solver as a backend. This is easily done, `detpoly.ml` should look like this.

```
let solve = Univsolve.universal_solve
              (Univsolve.universal_solve_init_options_verbose
               !Univsolve.universal_solve_global_options)
               my_solver
```

Note that, if your solver implemented in `my_solver` is recursive, it can also call `solve` instead of `my_solver` recursively in order to have the universal optimisations done in every step.

5.4 Integrating the New Solver

Continuing with the example of the previous section we assume that you have implemented a solver in the file `src/solvers/detpoly.ml`. In particular, this file contains a function `solve` of type `Paritygame.paritygame -> Paritygame.solution * Paritygame.strategy` which is declared in `src/solvers/detpoly.mli`.

You will have to ensure that your file does automatically get compiled when PG-Solver is being built using the `make` command. In `SolverList` you will find a variable declaration listing all the modules that make up the entire program. It should look like this.

```
PGSOLVERSLIST=
obj/recursive.cmx \
obj/stratimprovement.cmx \
...
obj/stratimprlocal.cmx \
obj/stratimprdisc.cmx
```

Append to this list `obj/detpoly.cmx`. It need not be at the end of this list but you have to make sure that it occurs behind any other module that contains code which is used by your solver. This could be the `paritygame` and `univsolve` part for example and will most certainly be the `solvers` part. Now running `make` should compile the `detpoly`-module as well.

At last, you have to integrate your algorithm into the program so that it can be used through the `pgsolver` program when given a certain command-line option. This could not be an easier. Add to your file `detpoly.ml` the following code after your `solve` function.

```
let _ = Solvers.register_solver
          solve
          "detpoly"
          "dp"
          "use the brand-new deterministic polytime algorithm"
```

The function `register_solver` from the module `solvers.ml` does everything for you. You simply have to give it four arguments.

- The first one, here `solve` is the function of the type described above that implements your algorithm.
- The second one is a string which should contain a concise but reasonably expressive synonym for your algorithm. It must not contain white spaces because it will be prefixed by two dashes "`--`" and used as a command-line parameter that tells PGSolver to use your algorithm.
- The third one is a string which contains an even shorter synonym for your algorithm. It will be prefixed by a single dash symbol "`-`" and used as a command-line parameter for `pgsolver` with the same effect.
- The fourth argument is again a string which contains a description of what PGSolver does when given any of the the command-line parameters derived from the second or third argument.

After compilation, called `bin/pgsolver --help` should result in output containing the following lines.

```
Options are
  ...
  --globallysolve, -global <solver>
     solves globally, valid solvers are ... detpoly ...
  ...
```

Finally, calling `bin/pgsolver -global detpoly` will solve a parity game using your algorithm.

5.5 Useful Functions

When designing a new solver you may need some functionality that other solvers rely on as well. In that case there is a good chance that one of the modules already contains the function you are looking for, and you do not have to re-implement it. In the following we describe some of the implemented functions that are most likely to be useful for solving parity games in general. All path names are relative to the subdirectory `src`, i.e. when we speak about module `Basics` in the `paritygame` directory for example then this can be found in the file `src/paritygame/basics.ml`.

5.5.1 The `Basics` Module

The main function in this module in the subdirectory `pgsolver` is used for producing output messages to `STDOUT` during a run of PGSolver depending on the configured verbosity level.

`type verbosity_level = int`
Used to define different levels of verbosity during the program's execution.

`val message : verbosity_level -> (unit -> string) -> unit`
Calling `message v (fun _ -> s)` outputs the string `s` on `STDOUT` if the currently set verbosity level is greater than or equal to `v`.

`val verbosity_level_verbose : verbosity_level`
Predefined verbosity level constant for verbose output (2).

`val verbosity_level_default : verbosity_level`
Predefined verbosity level constant for debug output (3).

5.5.2 The `Paritygame` Module

In addition to the basic functions for creating, accessing and manipulating parity games that have been described in Section 5.2.4 already, this module in the `paritygame` subdirectory contains more useful routines for handling parity games.

`val print_game : paritygame -> unit`
Calling `print_game game` prints game on `STDOUT` s.t. it could be parsed again.

`val pg_max_prio : paritygame -> priority`
Calling `pg_max_prio game` returns the greatest priority occurring in the `game`.

`val pg_min_prio : paritygame -> priority`
Calling `pg_min_prio game` returns the least priority occurring in the `game`.

`val pg_max_prio_for : paritygame -> player -> priority`
Calling `pg_max_prio_for game player` returns the greatest reward for `player` occurring in the `game`.

`val collect_max_prio_nodes : paritygame -> node list`
Calling `collect_max_prio_nodes game` returns all nodes with greatest priority.

`val subgame_by_list : paritygame -> node list -> paritygame`
Calling `subgame_by_list game nodes` returns a compressed subgame induced and ordered by the `nodes`-list

```
val merge_strategies_inplace : strategy -> strategy -> unit
```
Calling `merge_strategies_inplace strat1 strat2` adds all strategy decisions from `strat2` to `strat1`. Throws an `Unmergable`-Exception if the intersection of the domains of both strategies is not empty, i.e. if there is an index i such that the values of `strat1` and `strat2` at index i are both not `nd_undef`.

```
val merge_solutions_inplace : solution -> solution -> unit
```
Calling `merge_solutions_inplace sol1 sol2` adds all solution information from `sol2` to `sol1`. Throws an `Unmergable`-Exception if the intersection of the domains of both solutions is not empty, see above.

```
val strongly_connected_components: paritygame ->
                                        (node list array *
                                         scc array *
                                         scc list array *
                                         scc list)
```
Calling `strongly_connected_components game` decomposes the `game` into its SCCs. It returns a tuple `(sccs, sccindex, topology, roots)` where `sccs` is an array mapping each SCC to its list of nodes, `sccindex` is an array mapping each node to its SCC (represented by an integer value), `topology` is an array mapping each SCC to the list of its immediate successing SCCs and `roots` is the list of SCC having no predecessing SCCs. The type `scc` is (currently) just a synonym for `int`.

```
val attr_closure_inplace: paritygame -> strategy ->
                              player -> node list -> node list
```
Calling `attr_closure_inplace game strategy player region returns` the attractor for the given `player` and `region`. Additionally all necessary strategy decisions for `player` leading into the `region` are added to `strategy`.

5.5.3 The `Univsolve` Module

This module in the `paritygame` subdirectory contains the universal solver and some possibly helpful functions for the universal solving process:

```
val universal_solve :
      universal_solve_options -> (paritygame -> solution * strategy) ->
          paritygame -> solution * strategy
```
Calling `universal_solve verbosity solver game` starts the universal solving process using `solver` as a backend. It returns the solved game as a pair of `solution` and `strategy`.

```
val universal_solve_by_player_solver :
      universal_solve_options ->
          (paritygame -> player -> solution * strategy) ->
```

```
paritygame -> solution * strategy
```

Calling `universal_solve_by_player_solver verbosity player_solver game` starts the universal solving process using `player_solver` as a backend. Instead of a solver backend that solves arbitrary SCCs `player_solver` is supposed to solve an SCC w.r.t. a given player, for instance if `player_solver scc plr_Even` is called, the backend is supposed to determine whether a node is won by player 0 (return `plr_Even` for this node) or not (return `plr_undef` for this node). The strategy that is returned by `player_solver` is only considered w.r.t. the given player. The function returns the solved game as a pair of `solution` and `strategy`.

```
val universal_solve_trivial :
      verbosity_level -> paritygame -> solution * strategy
```

Calling `universal_solve_trivial verbosity game` starts the universal solving process with a trivial i.e. not-solving backend. This function is legal to be called when `game` can be completely solved by the universal solving process without requiring the backend. Calling `universal_solve_trivial` returns the solved game as a pair of `solution` and `strategy`.

```
val compute_winning_nodes :
      verbosity_level -> paritygame -> strategy -> player -> node list
```

Calling `compute_winning_nodes verbosity game strategy player` considers the subgame of `game` w.r.t. the `strategy` decisions for `player`; the strategy is assumed to be total w.r.t. `player`. It returns the list of nodes `player` wins on the game following `strategy`.

In all these cases the argument `verbosity` is used to determine whether or not statistics should be printed at the end.

5.6 Creating New Benchmarks

5.6.1 The Functor `Build`

The creation of new families of parity games, for example as new benchmarks, is eased by the functor `Build` in `paritygame.mli`. It automates the creation of the necessary data structures for a parity game and allows the user to specify such a game freely, using their own data types for nodes. In order to do so, one has to implement the signature

```
module type PGDescription =
  sig
    type gamenode

    val compare    : gamenode -> gamenode -> int

    val owner      : gamenode -> player
```

```
    val priority   : gamenode -> priority
    val successors : gamenode -> gamenode list
    val show_node  : gamenode -> string option

    val initnodes  : unit -> gamenode list
  end
```

i.e. create a module that implements some data type `gamenode` and functions which determine the owner and the priority of a node as well as its successors. Additionally, one has to provide a function that potentially yields a textual representation of a node, used for instance in displaying a game graphically. Finally, the function `initnodes` should return a list of initial nodes in such a parity game.

The `compare` function needs to provide a total order on nodes. It typically suffices to implement it generically via `let compare = compare`.

Applying the functor `Build` to a module of type `PGDescription` results in a module of type

```
module type PGBuilder =
  sig
    type gamenode

    val build            : unit -> paritygame
    val build_from_node  : gamenode -> paritygame
    val build_from_nodes : gamenode list -> paritygame
  end
```

i.e. a module which provides functions that build a parity game, either starting generically from the initial nodes as defined by the function `initnodes`, or specifically from a particular node or node list. The game is created by traversing its description starting at the chosen nodes and exploring successors successively. The user has to guarantee that the description yields a finite game; otherwise the `build`-functions may not terminate.

The total order given by `compare` is needed to store the nodes of the game description in a hash table or balanced tree where they are implicitly converted into values of type `node`.

Examples of how to use the functor `Build` for conveniently building parity games on-the-fly can be found in the `generators`-subdirectory, for example in the benchmark modules `towersofhanoi`, `elevatorverification`, `jurdzinskigame`, `langincl`, etc.

5.6.2 Registering New Generators

The modules for benchmarks typically only provide a function that creates a parity game as a value of type `paritygame`. They do not call the functions themselves, nor are they concerned with where the resulting parity games are being used.

These functions are then used in two places: the most visible one is through the module `rungenerator` in the `pgsolver` subdirectory. It calls the corresponding function

and then prints the resulting game on standard output. This code is used to create the stand-alone binaries for benchmark creation in the `bin`-subdirectory.

Printing games and parsing games is very costly, so it is not advisable to use this method to run serious benchmarks. It is also obviously wasteful to create a paritygame in the correct data type, print it and then parse it again into this data type. This is why PGSolver provides a way of integrating the benchmark creation into the main program. This means that games only need to be created once before they can be solved, without the need to print and parse them. The key to this is the function

```
val register_generator: (string array -> paritygame)
                        -> string -> string -> unit
```

in the `generators` module in the subdirectory `paritygame`. It takes as first argument a function which produces a parity game according to an array of strings which are the command-line parameters given to the generator. The two other arguments of type `string` are a short (i.e. not containing white spaces) description of the benchmark, used to identify this when telling PGSolver which benchmark to use, and a long description used for instance in the help text.

To integrate your benchmarks fully, create a function `my_generator` of type

```
string array -> paritygame
```

that returns a parity game in accordance to a parameter array (for instance conveniently using the functor `Build`), and include code of the form

```
let _ = Generators.register_generator my_generator
                                      "mygen"
                                      "My own benchmark generator"
```

somewhere, preferably in the same module as the one containing the benchmark generation code. Say its name is `mygenerator`. To ensure that this code is included in the main program during compilation, you need to add `mygenerator.gen` in the list of generators in `Makefile`.

Bibliography

[AAW03] A.Vincent, A. Arnold, and I. Walukiewicz. Games for synthesis of controllers with partial observations. *Theoretical Computer Science*, 303:7–34, 2003.

[BSV03] H. Björklund, S. Sandberg, and S. G. Vorobyov. A discrete subexponential algorithm for parity games. In *Proc. 20th Ann. Symp. on Theoretical Aspects of Computer Science, STACS'03*, volume 2607 of *LNCS*, pages 663–674. Springer, 2003.

[BV07] Henrik Björklund and Sergei Vorobyov. A combinatorial strongly subexponential strategy improvement algorithm for mean payoff games. *Discrete Appl. Math.*, 155(2):210–229, 2007.

[EJ91] E. A. Emerson and C. S. Jutla. Tree automata, μ-calculus and determinacy. In *Proc. 32nd Symp. on Foundations of Computer Science*, pages 368–377, San Juan, Puerto Rico, 1991. IEEE.

[EJS93] E. A. Emerson, C. S. Jutla, and A. P. Sistla. On model-checking for fragments of μ-calculus. In *Proc. 5th Conf. on Computer Aided Verification, CAV'93*, volume 697 of *LNCS*, pages 385–396. Springer, 1993.

[FL10] O. Friedmann and M. Lange. Local strategy improvement for parity game solving. In *Proc. 1st Symp. on Games, Automata, Logic, and Formal Verification, GandALF'10*, volume 25 of *EPTCS*, pages 118–131, 2010.

[FL12] O. Friedmann and M. Lange. Two local strategy improvement schemes for parity game solving. *Journal of Foundations of Computer Science*, 23(3):669–685, 2012.

[Fri09] O. Friedmann. An exponential lower bound for the parity game strategy improvement algorithm as we know it. In *Proc. 24th Ann. IEEE Symp. on Logic in Computer Science, LICS'09*, pages 145–156. IEEE, 2009.

[GH82] Y. Gurevich and L. Harrington. Trees, automata, and games. In *Proc. 14th Annual ACM Symp. on Theory of Computing, STOC'82*, pages 60–65. ACM, ACM Press, 1982.

[GTW02] E. Grädel, W. Thomas, and Th. Wilke, editors. *Automata, Logics, and Infinite Games*, LNCS. Springer, 2002.

[HKR02] T. A. Henzinger, O. Kupferman, and S. K. Rajamani. Fair simulation. *Inf. Comput.*, 173(1):64–81, 2002.

[HLL13] M. Hutagalung, M. Lange, and E. Lozes. Revealing vs. concealing: More simulation games for Büchi inclusion. In *Proc. 7th Int. Conf. on Language and Automata Theory and Applications, LATA'2013*, volume 7810 of *LNCS*, pages 347–358. Springer, 2013.

[JPZ06] M. Jurdziński, M. Paterson, and U. Zwick. A deterministic subexponential algorithm for solving parity games. In *Proc. 17th Ann. ACM-SIAM Symp. on Discrete Algorithm, SODA'06*, pages 117–123. ACM, 2006.

[Jur98] M. Jurdziński. Deciding the winner in parity games is in $UP \cap$co-UP. *Inf. Process. Lett.*, 68(3):119–124, 1998.

[Jur00] M. Jurdziński. Small progress measures for solving parity games. In H. Reichel and S. Tison, editors, *Proc. 17th Ann. Symp. on Theoretical Aspects of Computer Science, STACS'00*, volume 1770 of *LNCS*, pages 290–301. Springer, 2000.

[KW08] D. Kähler and Th. Wilke. Complementation, disambiguation, and determinization of Büchi automata unified. In *Proc. 35th Int. Coll. on Automata, Languages and Programming, ICALP'08*, volume 5125 of *LNCS*, pages 724–735. Springer, 2008.

[Lan05] M. Lange. Solving parity games by a reduction to SAT. In R. Majumdar and M. Jurdziński, editors, *Proc. Int. Workshop on Games in Design and Verification, GDV'05*, 2005.

[Mar75] D. A. Martin. Borel determinacy. *Ann. Math.*, 102:363–371, 1975.

[Pit06] N. Piterman. From nondeterministic Büchi and Streett automata to deterministic parity automata. In *Proc. 21st Symp. on Logic in Computer Science, LICS'06*, pages 255–264. IEEE Computer Society, 2006.

[Pur95] A. Puri. *Theory of hybrid systems and discrete event systems.* PhD thesis, University of California, Berkeley, 1995.

[Sch07] S. Schewe. Solving parity games in big steps. In *Proc. of the 27th Int. Conf. on Foundations of Software Technology and Theoretical Computer Science, FSTTCS'07*, volume 4855, pages 449–460. Springer, 2007.

[Sch08] S. Schewe. An optimal strategy improvement algorithm for solving parity and payoff games. In *Proc. 17th Annual Conf. on Computer Science Logic (CSL 2008)*, volume 5213 of *LNCS*, pages 369–384, 2008.

[SS98] P. Stevens and C. Stirling. Practical model-checking using games. In B. Steffen, editor, *Proc. 4th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'98*, volume 1384 of *LNCS*, pages 85–101. Springer, 1998.

[Sti95] C. Stirling. Local model checking games. In *Proc. 6th Conf. on Concurrency Theory, CONCUR'95*, volume 962 of *LNCS*, pages 1–11. Springer, 1995.

[SV00] D. Schmitz and J. Vöge. Implementation of a strategy improvement algorithm for finite-state parity games. In S. Yu and A. Pun, editors, *Proc. Int. Conf. on Implementation and Application of Automata, CIAA'00*, volume 2088 of *LNCS*, pages 263–271. Springer, 2000.

[Tar72] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[VJ00] J. Vöge and M. Jurdzinski. A discrete strategy improvement algorithm for solving parity games. In *Proc. 12th Int. Conf. on Computer Aided Verification, CAV'00*, volume 1855 of *LNCS*, pages 202–215. Springer, 2000.

[Zie98] W. Zielonka. Infinite games on finitely coloured graphs with applications to automata on infinite trees. *TCS*, 200(1–2):135–183, 1998.