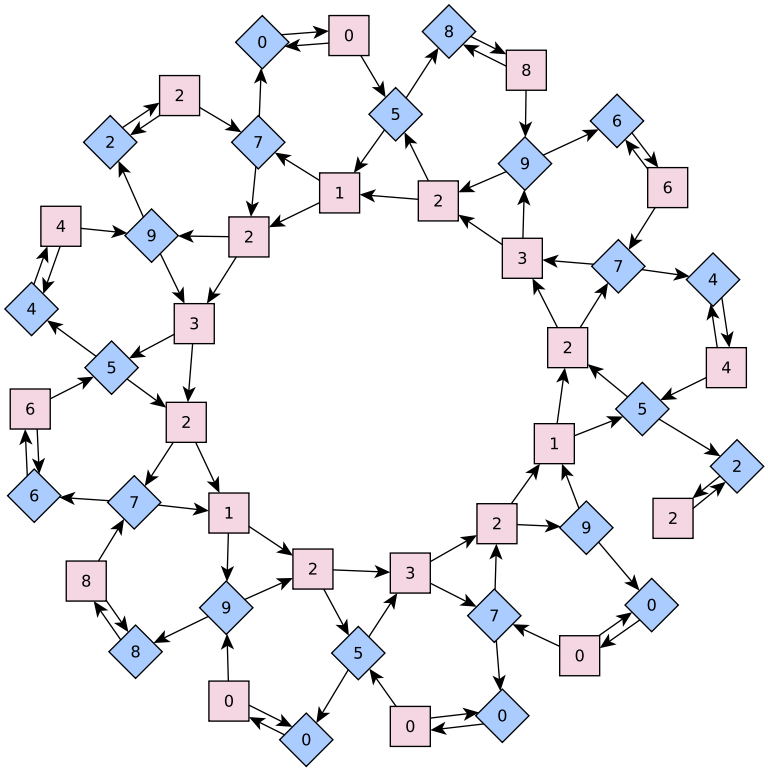
◇

Practical Improvements to Parity Game Solving

by Maks Verver



Master's thesis
Computer Science programme
University of Twente

12th December 2013

Graduation committee:
prof.dr. J.C. van de Pol
dr.ir. R. Langerak
prof.dr. M. Lange*
G. Kant, MSc

*University of Kassel

Abstract

The aim of this thesis is to investigate how parity game problems may be solved efficiently in practice. Parity games are a worthwhile research topic because their simultaneous simplicity and expressiveness makes them a useful formalism to represent the problems that occur when formal methods are applied to software and hardware engineering.

In this thesis, first an overview of the state of the art, both theoretical and practical, will be presented. Second, algorithms and data structures will be described which were used to develop a new parity game solving tool. These include a mix of well-known, relative obscure, previously undocumented, and completely novel techniques. The most valuable contributions are the introduction of novel preprocessing operations and improved heuristics for the Small Progress Measures solution algorithm. Third, the results of an empirical evaluation will be presented to demonstrate which of these techniques work best in practice.

The empirical results will support the conclusion that considerable improvements over the state of the art are possible using a combination of careful tool design and implementation, application of powerful preprocessing operations, and the use of advanced heuristics in the implementation of the Small Progress Measures algorithm.

Acknowledgements

I would like to thank my parents for their trust and support throughout my studies.

Contents

Chapter 1

Introduction

My final project deals with *parity games*. Let me start by disappointing the readers to which the word *game* implies a fun activity: parity games are actually a mathematical formalism involving games that literally go on forever and are not much fun to play by hand. Fortunately, however, they *are* an interesting subject to study.

One reason to study parity games is the observation that, despite considerable research interest, the computational complexity of parity games is not yet known. Perhaps as a result, previous research tended to focus on theoretical aspects related to parity games, somewhat neglecting practical considerations.

This is a pity, since parity games have practical applications too, most notably as a vehicle for formal verification (by reduction from model checking, bisimulation, satisfiability checking and software synthesis). Formal verification tools are important since engineers use them to create reliable hardware and software components. Parity game algorithms have the potential to improve these tools considerably — that is, if they can be made to work well in practice.

Perhaps surprisingly, there is comparatively little research that focuses on parity games as a practical tool rather than a mathematical formalism. Few implementations of parity game algorithms exist, and the strengths and weaknesses of these tools are often poorly understood.

The goal of my final project was to develop a framework for experimentation with parity games, in order to better understand the performance of a variety of parity game algorithms, and to push the boundaries of the range of parity game problems that can be solved in practice. Along the way I discovered several techniques to improve the standard algorithms; these improvements are documented in this report.

Although I met several hurdles while executing the project, I believe the final result is quite good. The tool I developed is orders of magnitude more efficient than the competitors (though few exist) which demonstrates a practical contribution to the state of the art. Indeed, the observation that such large performance gains are possible at all supports the notion that further research on practical performance is useful.

Finally, it is my hope that my contributions not only prove beneficial to practitioners today, but will also increase the confidence in parity games as a useful formalism for other, higher-level problems, as well as inspire new developments in high-performance parity game solvers.

1.1 Parity games

A *parity game* is a game played by two players, called Even and Odd, on a directed graph. Each vertex in the graph is associated with (*owned* or *controlled* by) one of the two players. Furthermore, to each vertex a *priority* is assigned, which is a non-negative integer.

There are different conventions for the player's names. They may be called Even and Odd, or denoted by symbols $\diamond$ and $\Box$, which is particularly useful when visualizing games, as can be seen in Figure 1.1: the shape of the vertices corresponds to the players that control them. For

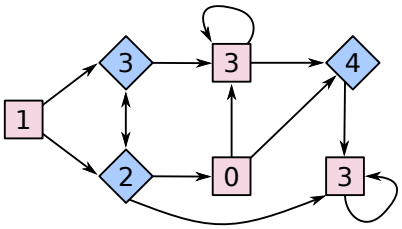


Figure 1.1: A small example of a parity game

the description of algorithms and data structures, especially when computations are involved, it is more convenient to use integers 0 and 1. For example, if we consider a player x, his opponent is $1 - x$.

Formally, a parity game can be described as a quadruple $\Gamma = (V_0, V_1, E, \phi)$, where $V = V_0 \cup V_1$ and $V_0 \cap V_1 = \emptyset$ (in other words: V_0 and V_1 partition the set of graph vertices V into those controlled by Even and Odd respectively). $E \subseteq V \times V$ is the set of directed edges in the game graph. This set may contain loops (edges which lead from a vertex back to itself). $\phi : V \to \mathbb{N}_0$ is the priority function that assigns a priority to every vertex in the graph. The number of distinct priority values assigned to vertices in the game, is called the *index of the game*, which is equal to the cardinality of the range of ϕ.

Parity games can be played on finite as well as infinite graphs, although I will consider finite graphs exclusively. The index of the game is always assumed to be finite.

1.1.1 Game play and winning conditions

A parity game is played by placing a token on some initial vertex. The player controlling that vertex moves the token along an edge to an adjacent vertex, which may belong to either player, who then makes the next move. When the token lands on a vertex without any outgoing edges, the game ends. However, it is more common for a game to continue indefinitely, causing an infinite sequence of moves. The sequence of moves on the graph is called a *play* and it can be described as the sequence of vertices visited by the token. Formally, a sequence $\pi = v_1 v_2 \dots v_n$ is a (finite) play if and only if $\forall i < n,\ v_i v_{i+1} \in E$ and v_n has no outgoing edges. Similarly, an infinite sequence $\pi = v_1 v_2 \dots$ is an (infinite) play if and only if $\forall i \in \mathbb{N},\ v_i v_{i+1} \in E$. A prefix of a play (ending on some vertex with outgoing edges) is called a *partial play*.

For finite plays, the player who is first unable to move is called the loser, and his opponent the winner. In a finite play, the loser is therefore simply the controller of the final vertex in the play.

For infinite plays, a more complicated notion of winning is used. Let the *dominant priority* $P(\pi)$ for a play $\pi = v_1 v_2 \dots$ be the smallest value that occurs infinitely often in the sequence $\phi(v_1)\phi(v_2)\dots$ or formally:

$$P(\pi) = \min \{p \in \mathbb{N}_0 :\ \forall i\ \exists j > i :\ \phi(v_j) = p\}$$

A play is won by player Even if the dominant priority for the play is even, and won by Odd otherwise (hence their names). Since the set of priorities is finite, the dominant priority is well-defined for any play, and thus every play has a winner.

It should be noted that there is no consensus in literature on how priorities should be ordered. Throughout this report I will use the convention of lower priority values taking precedence over higher values, thus 0 being the "highest" priority, which is consistent with the definition given above.

1.1.2 Strategies and solutions

A strategy for player x assigns a move to each position in which x is to move. Formally, the strategy is a function $\dot{\sigma}_x : V^* \times V_x \to V$ such that if $v_1 \dots v_n$ is a partial play, then $\dot{\sigma}_x(v_1 \dots v_n) = v_{n+1}$

and $v_1 \dots v_{n+1}$ is a (partial) play too. A play $\pi = v_1 v_2 \dots$ is called consistent with a strategy $\dot{\sigma}_x$ for player x if $\dot{\sigma}_x(v_1 \dots v_i) = v_{i+1}$ for all $v_i \in V_x$.

A strategy $\dot{\sigma}_x$ is called *winning* for player x at starting vertex v_1 if all plays $v_1 v_2 \dots$ consistent with $\dot{\sigma}_x$ are won by player x. Parity games of finite index have the important property that they are fully *determined*, i.e. for every starting vertex either player Even or player Odd has a winning strategy. Thus, for these games, we can partition the vertex set V of the game graph into two sets of vertices W_0 and W_1 which can be won by player Even and Odd respectively. When the index is infinite, we can still identify disjoint sets W_0 and W_1, but they may not be a true partition.

In many practical applications, determination of winning sets is enough to constitute a solution. For example, when using parity games as a vehicle for model checking, the question whether a formal property holds corresponds to the question whether a particular vertex in a game graph is won by player Even. In this case it suffices to determine the winner for this particular vertex only, without computing associated strategies, and even without fully determining winning sets.

A limitation of calculating winning sets without associated strategies is that even if we assume the output to be correct, the winning sets alone do not provide any insight into *why* a particular vertex is won by a particular player. Strategies are useful to understand the outcome of the games. In the application of model checking, strategies can be used to show why a certain property holds, or generate counter-examples if it doesn't. Additionally, if we have not just a winning set, but also associated strategies, we can check the correctness of a proposed solution without having to solve the game from scratch.

Therefore, *solving a game* in the most general sense means identifying optimal strategies for both players in addition to their winning sets.

1.1.3 Optimal strategies and finite memory

A strategy $\dot{\sigma}_x$ is called an *optimal strategy* when it is winning for player x starting from any vertex $v \in W_x$.

Strategies as described above are called *unbounded memory strategies*, because they can take the entire move history into account to determine the next move. By contrast, *bounded* or *finite memory strategies* limit the relevant move history to a fixed depth, while *memoryless strategies* are bounded memory strategies which depend only on the current position of the token, i.e. $\dot{\sigma}_x(v_1 \dots v_n) = \dot{\sigma}_x(w_1 \dots w_m)$ whenever $v_n = w_m$.

We will define memoryless strategies as functions $\sigma_x : V_x \to V$ such that if $\sigma_x(v) = w$ then $vw \in E$. A memoryless strategy σ_x is then consistent with a play $v_1 v_2 \dots$ if $\sigma_x(v_i) = v_{i+1}$ for all $v_i \in V_x$. We can restrict the domain of σ_x to $V_x \cap W_x$ since for vertices in V_x but not in W_x, player x has no winning move, and therefore any adjacent vertex can be selected without affecting the optimality of the strategy. Even if we leave out these vertices for which the controlling player has no winning move, optimal strategies are not (necessarily) uniquely defined, unlike winning sets.

Sometimes we want to refer to the combined strategies of both players, σ, defined as:

$$\sigma(v) = \begin{cases} \sigma_0(v) & \text{if } v \in V_0 \\ \sigma_1(v) & \text{if } v \in V_1 \end{cases}$$

It turns out that for all games of finite index optimal memoryless strategies exist. Research on finite-order games typically focuses on finding memoryless strategies for both players, which can be described succinctly by simply listing an optimal move for every vertex.

This report is about finding optimal strategies for finite parity games, and therefore the term *strategy* without further qualification will be used to mean *optimal memoryless strategy*, and the *solution* to a parity game is a triple W_0, W_1, σ describing the winning sets and optimal strategy for both players.

In Figure 1.2 the solution for the example game presented earlier is shown. Vertices are partitioned into winning sets for both players. Edges that cannot be part of winning strategies are dashed. In this particular case, any of the solid edges can be chosen to yield an optimal strategy.

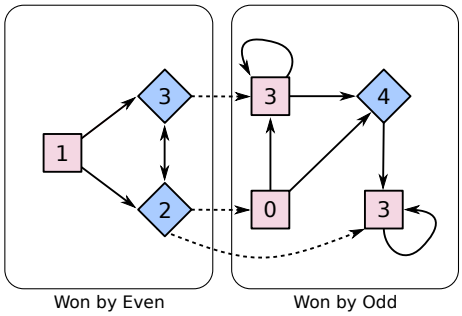


Figure 1.2: The example game solved

1.2 Computational complexity

Given a polynomial-time algorithm to verify the optimality of a pair of winning strategies (one such algorithm will be presented later on), the general problem of determining winning sets as well as strategies can be solved by nondeterministically guessing the winner and an optimal move for each vertex, which puts the problem in NP (the set of problems decidable in polynomial time by a non-deterministic Turing machine and verifiable by a deterministic Turing machine) and co-NP (by symmetry of the problem: absence of a winning strategy for one player can be proven by showing a winning strategy for the other player). The strategies are certificates to the solution.

NP and co-NP are generally believed to be distinct, and since the existence of NP-hard problems in co-NP would imply NP = co-NP, it seems unlikely that the problem is NP-hard. This makes it plausible that there should be a polynomial-time algorithm to solve parity games, but despite considerable research interest, none have been found.

If there exists a polynomial-time algorithm that determines winning sets only (without associated strategies) then full solutions can be constructed in polynomial time too, again by guessing winning strategies. This restricted problem has been shown to be in UP ∩ co-UP [22] where UP is the subset of NP containing those problems that are decidable by an *unambiguous* nondeterministic Turing machine. Since P ⊆ UP ⊆ NP this is a stronger result, although it is not known whether P ⊂ UP or UP ⊂ NP (or both, or neither).

The best currently known algorithms are either exponential in game index d (for example, Jurdziński's Small Progress Measures algorithm [23] has an upper bound of $O\left(d\,|E|\left(\frac{|V|}{\lfloor d/2\rfloor}\right)^{\lfloor d/2\rfloor}\right)$) while others are instead sub-exponential in the size of the game graph (for example, Jurdziński, Paterson and Zwick give an $|V|^{O(\sqrt{|V|})}$ algorithm in [24]).

Often d is small in practice; in those cases the first category of algorithms promises better performance. However, since many instances can be solved much faster than the best available bound would suggest, upper time bounds alone cannot be used to predict which algorithms work best in practice.

1.3 Application to model checking

The most notable practical application of parity games is their suitability as a component in model checking systems, which attempt to verify whether a model of a system (typically expressed in some high-level formal language) satisfies a certain specification (typically captured in a temporal logic formula).

Model checking tools must balance two concerns: on the one hand, the specification language available to the user should be as expressive as possible, while on the other hand, the internal representation of the resulting model checking problem should be kept simple enough so that it can be solved quickly using provably-correct algorithms.

Parity games fit these constraints well because they can be used to express whether a property specified in the modal μ-calculus holds on a labelled transition system (LTS) [10]. Since many high-level specification languages can be converted into labelled transition systems, and formulas expressed in commonly used temporal logics (including LTL, CTL and CTL*) can be translated to the modal μ-calculus [9] (though a linear translation is possible only for CTL), efficient algorithms for solving parity games could provide the basis of a comprehensive approach to model checking.

Not only are parity games sufficiently powerful to express these problems, but conversion from the model checking problem to parity games yields reasonably compact games. The number of vertices equals the product of the number of states in the LTS and the number of subexpressions in the μ-calculus formula. The number of priorities corresponds to the alternation depth of fixed-point operators in the formula (for example, see [34] for one explicit construction method). Finally, and importantly, these games can be constructed in linear time. Thus, model checking via parity games would be a very practical approach, assuming the final parity games can be solved quickly.

1.4 Related work

The work related to mine falls into three broad categories: theoretical work on parity game algorithms, practical work on formulating problems as parity games, and implementations of parity game algorithms. I will summarize the developments most relevant to my project in this section.

1.4.1 Automata theory and parity game algorithms

A considerable amount of literature is available on parity games specifically and more generally on perfect-information games played on (possibly infinite) graphs and trees, as well as their relationship to computing automata and their application to model checking. The majority of this work is theoretical in nature.

An important property of parity games is the fact that the winner can be determined for every vertex using memoryless strategies. A proof of determinacy of Borel games, of which parity games are a specialization, was first formulated by Donald A. Martin [32]. For Borel games with a Rabin winning condition, of which the parity condition is a special case, Klarlund showed that whenever the condition can be satisfied, there is a memoryless strategy to do so [30]. Both of these results are more general than needed for parity games; Zielonka gives two direct proofs for the memoryless determinacy of parity games specifically [41].

Some of the terminology introduced to discuss more general classes of games (like Muller games, which cannot generally be solved with memoryless strategies) is also useful for discussing parity games, and these will be repeated in section 2.2.

For parity games specifically, many solving algorithms have been proposed, including:

- A recursive algorithm based on a constructive proof of determinacy, first described by McNaughton [33] and reformulated for parity games by Zielonka [41]
- Fixed-point iteration using Small Progress Measures by Jurdziński [23]
- Strategy improvement by Vöge and Jurdziński [40]
- A randomized sub-exponential algorithm by Björklund, Sandberg and Vorobyov [3]
- A deterministic sub-exponential algorithm by Jurdziński, Paterson and Zwick [24]
- A combination of McNaughton's and Jurdziński's algorithms that proceeds in "big steps" by Schewe [35]
- Optimal strategy improvement by Schewe [36]

Additionally, various indirect approaches to solving parity games by reduction to other formulations such as SAT ([31], [21]) and Boolean equation systems have been proposed.

This gives plenty of algorithms to choose from, though there is no agreement on which algorithms work best in practice. Schewe favours approaches based on strategy improvement, stating these are: *"[..] fast simplex style algorithms that perform well in practice. While their complexity is wide open, they are often considered the best choice for solving large scale games."*[36]

However, Friedmann and Lange have performed an empirical evaluation of various algorithms, and conclude that *"the small progress measures algorithm as well as the strategy improvement turn out to be generally slower than the recursive algorithm*" and conclude that *"Zielonka's recursive algorithm is the best parity game solver in practice*" [12].

Of these algorithms, Jurdziński's Small Progress Measures, Schewe's "big steps" variant, and the sub-exponential algorithms are intended mainly to establish ever tighter upper bounds on the general problem. The approaches based on strategy improvement are intended to yield best results in practice. The relatively simple approach due to McNaughton/Zielonka often works well too. Its worst-case performance has been analysed by Gazda and Willemse ([17]) who conclude that Zielonka's algorithm can solve weak games (games in which priorities are nondecreasing along all edges) in polynomial time, but may require exponential time even on other, very simple, classes of games (dull games and solitaire games). The slowdown on these cases can be mitigated by interleaving Zielonka's algorithm with decomposition of the game graph into strongly-connected components although this does not lower the worst-case complexity in general.

1.4.2 Model checking tools

At least two formal verification systems exist that can transform model checking problems into parity games:

1. The mCRL2 toolkit [19] composes models and formulae into parametrized Boolean equation systems (PBES) [20], can linearise these into Boolean equation systems (BES), and either solve those directly (with a tool called `pbes2bool`) or translate them into parity games instead. Various transformations on the level of PBES or BES are also possible.

2. The LTSmin toolkit [4] supports a different approach: it transforms a PBES into a parametrized parity game (PPG) and then generates the parity game directly [27]. This approach was motivated by the observation that linearisation of the PBES was often the slowest step in the solution process, and that optimization of this conversion can drastically reduce both time and memory required.

In both of these toolkits, the parametrized forms tend to be fairly small, while the linearised forms (and therefore the final parity game) may be quite large. Model checking using a parity game solver as a backend is viable only if the resulting parity games can reasonably fit into memory. This is a limitation shared with many other approaches to model checking (in particular those relying on explicit state space exploration).

1.4.3 Simulation reductions

Recent work has focused on polynomial-time reductions on the game graph, based on the principle that any reduction of the graph that preserves winners is at least permissible, and will often simplify solving the game. In principle every parity game can be reduced to just two vertices, one won by each of the players, but computing such a reduction would be equivalent to solving the parity game and therefore this is not feasible in polynomial time. However, finer equivalence relations which preserve winners are efficiently computable, and may help to reduce the size and complexity of the problem before passing it to an (exponential time) generic solver.

All of these reductions depend on a high degree of redundancy in the input game to be effective; such redundancy typically arises in games that are automatically generated from higher level problems, but not necessarily in e.g. randomly generated games.

1.4.3.1 Delayed simulation

Fritz and Wilke ([15, 16]) investigated both *direct* and *delayed simulation* in parity automata (of which parity games are a special case). Since direct simulation equivalence is too restrictive to provide significant benefits, they introduce the weaker notion of delayed simulation instead: traces do not need to show equivalent priorities in all positions, as long as a priority visited in one automaton is eventually (after a finite number of steps) matched by the same (or better) priority in the other. A low priority encountered in one automaton thus creates an obligation that must eventually (but not necessarily immediately) be fulfilled in the other one.

This relation does not preserve the language recognized by the automaton and thus a somewhat more limited relation, called *biased* delayed simulation, is introduced, which can be used to partition vertices into equivalence classes and construct a valid quotient that preserves the winners of all vertices.

In theory, biased delayed simulation works well; for example, all games in the class introduced by Jurdziński in [23] can be reduced to their two-vertex minimal representation; the best possible result. However, in general, the costs seem to outweigh the benefits, as Fritz reports that: "experiments with an implementation indicate that solving a given parity game using Jurdziński's lifting algorithm directly is faster than first simplifying the parity game using our approach and then solving it using Jurdziński's algorithm". Although he does not specify which test cases were used to reach this conclusion (there may be practical cases which do benefit from the reduction, aside from Jurdziński's special cases) the fact that computing the delayed simulation equivalence relation is relatively expensive for a polynomial-time preprocessing step (requiring $O(|V|^3 \cdot |E| \cdot d^2)$ time in the worst case) probably limits the utility of their approach in practice.

1.4.3.2 Stuttering equivalence

Cranen, Keiren and Willemse instead introduce the notion of *stuttering equivalence* ([7]) which is neither strictly stronger nor weaker than biased delayed simulation reduction. It is somewhat limited in that it only groups together vertices with equal owners and priorities. However, the stuttering equivalence relation can be calculated quickly (in only $O(|V| \cdot |E|)$ time) and allows quotients of games to be constructed without further complications. The authors report that, in contrast to Fritz and Wilke's results, applying stuttering equivalence reduction before solving often provides a net benefit in practice, and usually performs better than strong bisimulation reduction.

Stuttering equivalence is further refined into *governed stuttering bisimulation* ([8]) which allows vertices with different controlling players to be grouped together too. In theory this should yield further reduction of the game graph, though at a higher computational cost (since the relation requires $O(|V|^2 \cdot |E|)$ time to calculate). An empirical evaluation suggests that the two reductions offer comparable performance in practice.

1.4.3.3 Bisimulation on Boolean Equation Systems

Another approach is taken by Keiren and Willemse ([29]) on Boolean equation systems, which are a formalism very similar to parity games (i.e. parity games can be considered a restricted form of Boolean equation systems) and are used as an intermediate representation in the mCRL2 toolkit.

Keiren and Willemse introduce the notion of *idempotence preserving bisimulation*, which has no direct analogue on parity games, but falls between strong bisimulation and branching bisimulation in terms of refinement. The cost of calculating the bisimulation relation is low, requiring $O(|E| \cdot \log |V|)$ time. An empirical evaluation shows that solving with idempotence preserving bisimulation reduction is slower than with strong bisimulation reduction, even though in some cases idempotence preserving bisimulation creates smaller quotients. However, both reductions provide great benefits over solving the game directly. The choice of algorithms used to solve the final parity games (Jurdziński's Small Progress Measures and Schewe's "big step" variant) could have exaggerated the results somewhat.

It is worth noting that these optimizations are also worthwhile because converting parametrized Boolean equation systems to parity games can take a significant amount of time in the total solution process, especially if the resulting game turns out to be easy to solve.

1.4.4 Parity game solvers

Although many algorithms have been proposed in literature (as discussed in section 1.4.1) few implementations are mentioned, and there is little discussion of practical considerations (speed in practice, memory use, parallelizability, distributability, et cetera). This is suprising: considering the utility of parity game solvers as part of a model checking toolchain, and the notorious gap between theoretical time complexity and solving time required in practical instances, one would imagine there to be more research interest in determining which algorithms work well in practice.

The main exception is PGSolver (*"a collection of tools for generating, manipulating and - most of all - solving parity games"* [14]) which implements a large number of algorithms described in literature, as well as a number of preprocessing operations to simplify instances before passing them on to a solver backend. This tool is actively maintained, and forms the basis of Friedmann and Lange's empirical evaluation [12].

Jeroen Keiren also used PGSolver to evaluate the suitability of parity game algorithms to solve the Boolean equation systems generated by the mCRL2 toolkit [28]. Although PGSolver is a powerful tool, it also has some limitations: it's a strictly sequential solver that doesn't support parallel or distributed solving, and since it is implemented in OCaml, it relies on automatic memory management, which may not be very efficient (especially when solving large problems).

Van de Pol and Weber developed a multi-core implementation of Small Progress Measures [38] (the first parallel implementation of a parity game algorithm). Their project provided a starting point for my research (in particular, it inspired the focus on heuristically improvements to Small Progress Measures). The tool itself, however, is unmaintained and somewhat limited in terms of supported algorithms; consequently its practical utility beyond research purposes is also limited.

In summary, it appears that PGSolver is the only comprehensive tool easily available to users that have a parity game problem to solve.

1.5 My contributions

From the above it should be clear that there is a gap between theory and practice with respect to the use of parity games as a vehicle for formal verification. On the practical side there is a desire to efficiently solve model checking problems that arise in practice. Tool support to express these problems as parity games already exists. On the theoretical side there is no dearth of proposed algorithms for solving parity games, but there is only one tool that actually implements these algorithms: PGSolver.

It should come as no surprise that my contributions lie in this area. They consist of a variety of ideas to improve upon the design and implementation of parity game solvers, empirical evaluation of these ideas, and, as a by-product, an efficient parity game solving tool which improves upon the state of the art in several ways:

- For mCRL2, my tool provides an alternative to `pbes2bool` (the default solver for PBES). My solver has already been integrated into mCRL2 as an experimental tool called `pbespgsolve` which can be used today.
- Compared to PGSolver, my tool features fewer different solver algorithms, but better preprocessing routines and heuristics. My tool typically uses less time and space to solve test cases, sometimes dramatically so.

I want to stress that improving upon PGSolver was not a goal of the project in itself. However, having independent implementations of some of the critical algorithms (in different programming languages and using different data structures) is very useful to put empirical data such as reported

by Friedmann and Lange in perspective, because it helps quantify to which extent results depend on the performance of the algorithms themselves and how much they are influenced by implementation techniques.

My solver also includes a few innovations that are documented in this report:

- A new preprocessing technique which removes cycles controlled by a single player from the game (described in subsection 4.3.3.2)
- An efficient strategy verification algorithm, independent of solving algorithms (described in section 2.4)
- Improved heuristics for the Small Progress Measures algorithm (described in subsection 4.2.5)
- Two improvements on the design of the Small Progress Measures algorithm (described in subsections 4.3.1 and 4.3.2)

Finally, my report includes an empirical evaluation of these algorithms and techniques (presented in chapter 6) in order to give more insight in which techniques are most useful to solve parity games in practice.

Chapter 2

Building blocks for parity game solvers

Before discussing algorithms used to solve parity games, it is useful to establish some common terminology related to parity game theory. Most of this terminology comes from literature, but since the terms used and their meaning varies considerably, it is necessary to define the concepts and terms that will be used in the rest of this report.

First, a number of restrictions on the game structure will be made to simplify the definition and discussion of the algorithms and data structures that follow. Second, a number of useful properties of parity games will be presented; none of these are new, but they will be defined in a manner consistent with the preceding definitions. Third, a number of notable special cases of parity games will be discussed; these deserve to be mentioned, although solving special games is not the main focus of this report.

The concepts described in this chapter help to reason about the efficiency and correctness of preprocessing operations and general solvers, which will be presented in later chapters. For example, Zielonka's recursive algorithm relies on the property that a game induced by the complement of an attractor set is a proper subgame, but so does the loop removal preprocessor, which is otherwise quite different. In that sense, these concepts form the common building blocks from which many useful algorithms can be constructed.

Finally, a polynomial-time algorithm for the verification of parity game algorithms will be presented, which is used to guarantee the correctness of the reported results. This is of practical utility.

2.1 Proper game graphs

For convenience, I will assume that every vertex has at least one outgoing edge. This property makes finite plays impossible, which simplifies the analysis of many algorithms. We will call a game a *proper game* if its graph satisfies this property. However, improper games can be turned into proper games by considering each vertex without outgoing edges: if it is controlled by player x, we can change its priority to $1-x$ and add an edge from the vertex back to itself. In the modified graph every vertex has an outgoing edge, yet it has the same solution and winning strategies as the original graph.

Additionally, I will assume the game graph is finite. This has practical as well as theoretical benefits. From a practical point of view, since all of the game data is now finite, it allows us to represent graphs explicitly using only finite memory (otherwise, a symbolic representation would be required). From a theoretical point of view, a finite vertex set allows for algorithms and proofs that do not generalize to infinite graphs.

2.2 Common terminology

There are a number of concepts which can be applied to parity games which have been described in literature before. In particular, Zielonka introduces some useful terminology in a treatise on two-player games played on coloured graphs (of which parity games are a subset) which will be repeated here. He describes attractor sets and traps. Additionally, I will describe subgames analogous to (though slightly different from) subarenas.

2.2.1 Subgames

A *subgame* of a game $\Gamma = (V_0, V_1, E, \phi)$ induced by a vertex set $U \subseteq V$ is the game $\Gamma|U = (V_0 \cap U, V_1 \cap U, E \cap (U \times U), \phi|U)$ where $\phi|U$ denotes ϕ with its domain limited to U. In other words, the game obtained when only considering vertices from U and ignoring the rest. $\Gamma|U$ is called a *proper subgame* if it is a proper game as described above.

2.2.2 Traps

Let vE be the set of vertices which are successors of v in E, or formally $vE = \{w \in V : vw \in E\}$. Analogously, $Ew = \{v \in V : vw \in E\}$. A non-empty vertex set $U \subseteq V$ is a trap for player x (or an x-trap, for short) when, informally, player x cannot force the token out of U. Formally, U is an x-trap if for all $v \in U$:

$$v \in V_x \rightarrow vE \setminus U = \emptyset$$

$$v \in V_{1-x} \rightarrow vE \cap U \neq \emptyset$$

In literature, x-traps are sometimes called *dominions* for player $1 - x$ instead.

2.2.3 Attractor sets

An attractor set for a player x on a vertex set $U \subseteq V$, denoted $Attr^x(\Gamma, U)$, is the set of vertices from which the player x can force the token into one of the vertices in U (including, by definition, vertices in U itself). Zielonka gives an iterative definition of an attractor set:

$$U_0 = U$$

$$U_{i+1} = U_i \cup \{v \in V_x : vE \cap U_i \neq \emptyset\} \cup \{v \in V_{1-x} : vE \setminus U_i = \emptyset\}$$

$$Attr^x(\Gamma, U) = U_0 \cup U_1 \cup \cdots$$

Since the graph is finite eventually $Attr^x(\Gamma, U)$ converges to some $U_i \subseteq V$ when $U_i = U_{i+1}$ and we can find that point by iteratively computing the sets up to this point. In practice a slightly different approach is used, as will be described in subsection 3.1.6.

In literature, attractor sets are sometimes called *force sets* instead.

2.2.3.1 Attractor strategies

An important property of attractor sets is that if $U \subseteq W_x$, then $Attr^x(\Gamma, U) \subseteq W_x$ too. Of course, if we know the optimal strategy for all vertices $v \in U$, then we also want to extend this strategy to $Attr^x(\Gamma, U)$. Fortunately, this can easily be done: every vertex that first appears in U_{i+1} (i.e. it is a member of $U_{i+1} \setminus U_i$) has a successor in U_i, and when we repeatedly choose such a successor, then we arrive at U_0 in i steps, at which point the rest of the strategy is known. Therefore, attractor set computation of a winning region with known strategy yields a strategy for the entire attractor set too.

2.2.3.2 Duality between attractor sets and traps

The second important property of attractor sets is that the complement V' of an attractor set for player x (formally: $V' = V \setminus Attr^x(\Gamma, U)$) is a trap for x. Moreover, if Γ is a proper game, $\Gamma|V'$ is a proper subgame of Γ, since if a vertex $v \in V'$ has no successor $w \in V'$ then all its successors must be in $Attr^x(\Gamma, U)$ and v would have, by definition, been in the attractor set, instead of its complement.

This property is important because it means that if we start with a proper game then we can safely remove attractor sets of arbitrary vertex sets to obtain proper subgames, which is not the case if we would remove arbitrary vertex sets. This technique can be used to break down a game in parts which are solved separately.

2.3 Degenerate cases

In addition to games which do not comply with the restrictions mentioned earlier, there are also a few classes of degenerate games that are special cases of the general game described above. They are mentioned separately because specific algorithms exist to solve them more quickly.

These special cases are occasionally provided as input to a solver (for example, as the representation of a particularly simple model checking problem) but more commonly they arise as subgames to be solved after preprocessing the game or after partial solving.

2.3.1 Single-parity games

If the priorities of vertices all have the same parity (even or odd) then the corresponding player will trivially win from every starting vertex, with an arbitrary strategy. A special case is the single-priority game, where only a single priority is used. (Priority compression, which will be described in Subsection 4.3.3.3, could also be used to convert the former case to the latter.)

2.3.2 Single-player games

A parity game is a *single-player game for player* x when all vertices controlled by player $1 - x$ have outdegree equal to 1. In such a game, only player x can make choices, and player $1 - x$ is forced to always move the token to the single available successor whenever it lands on one of his vertices.

In a single-player game, player x wins precisely from the vertices which lie on a cycle of which the minimum priority has parity equal to x, as well as from all vertices from which such a cycle can be reached. After all, his opponent has no choice, so he can never force the token out of a cycle or prevent player x from reaching a cycle when there exists a path to it. The remaining vertices (if there are any) are won by player $1-x$. We can find these cycles and solve single-player games in $O(d\,|E|)$ time, for example using the algorithm described in Subsection 2.4.1.

In extremely rare cases the game is played on a graph consisting of cycles only and then neither player has a choice. In these games, called cycle games, strategies are trivial and the winner of each cycle corresponds to the parity of the least priority occurring on the cycle.

2.3.3 Graphs of multiple components

Some game graphs are not strongly connected, in the sense that there may be pairs of vertices where there exists no path from one vertex to the other. These games can be solved more efficiently by identifying strongly connected components, solving each component separately, and then combining the results.

An easy way to implement this is to solve components in reverse topological order. When a component has edges pointing to vertices outside the current component, these can be replaced by edges to dummy vertices (one won by Even and one won by Odd) and then the subgame induced by the component can be solved. (Solving the components bottom-up guarantees the winner for

vertices outside of but reachable from the current component is well-defined.) This approach is simple and has the advantage that the decomposition algorithm is run only once, so its overhead is strictly limited to $O(E)$ time. However, it may miss some opportunities for decomposition.

A somewhat more sophisticated approach is to extend the winning regions identified after solving a component into their attractor sets in the main game. Then, when considering a component which contains some previously-solved vertices, we can construct a subgame without those vertices and recursively invoke the decomposition algorithm, which may be able to break down the graph into even smaller components.

The second method relies on the observation that removing attractor sets of winning regions from a game results in a proper subgame, so no dummy vertices are required, and has the advantage of being able to cheaply solve additional vertices through attractor set computation (thus never invoking the general solver for those vertices) and possibly creating smaller subgames to solve (when removal of attractor sets splits up a large component).

The latter method is implemented in PGSolver and my solver as well. It seems to work well in practice, though there is a risk: since the decomposition algorithm is invoked recursively, it is possible to waste a lot of time decomposing graphs rather than solving games. To prevent this, my solver imposes a limit on the recursion depth (by default: maximum 10 recursive invocations); when the maximum recursion depth is reached, subgames are passed directly to the general solver, without attempting to decompose the game graph further.

This trade-off occurs because finding even small components in a large graph may take time proportional to the size of the entire graph; an algorithm that could find a (bottom-most) component while requiring only time proportional to the size of that component would provide a more elegant resolution to the problem, but no standard algorithm with this property exists.

2.4 Verification

To ascertain the correctness of the implemented algorithms, it is useful to have a means of verifying the solutions produced by these algorithms. Of course, algorithms are typically published with a correctness proof before they are implemented, but mistakes could be introduced during implementation, which makes it worthwhile to implement a verification routine to validate the results independent of any solution algorithms.

Note that solutions produced by different algorithms cannot generally be compared: although winning sets are unique, strategies are generally not, so when two algorithms produce different strategies, that does not imply either is wrong.

The verification algorithm described below depends on the efficient solution of single-player games, for which a $O(|E|\,d)$ time algorithm will be presented first. Algorithms with the same goal and time complexity are also described in [14]. An even better complexity bound of $O(|E| \log d)$ can be achieved with an approach as described in [18] (which is more easily adapted to a direct verification algorithm than a solver for single-player games, although both are possible). Compared to these alternatives, the algorithms presented here are considerably simpler.

2.4.1 Solving single-player games

Without loss of generality, suppose we want to solve a single-player game for player Even. Player Even can win from at least some vertices if the graph contains a cycle with even dominant priority (for brevity, let's call this an *even cycle*).

To solve the game, we iteratively identify an even cycle $c_1 c_2 \ldots c_n \in V^+$ in the game ($c_i c_j \in E$ if $i + 1 = j \mod n$ and $\min \phi(c_j) = 0 \mod 2$) and then solve the smaller subgame $\Gamma|V \setminus Attr^0(\Gamma, \{c_1, c_2 \ldots, c_n\})$ in the same manner. The strategy for player Even is formed by combining $\sigma_0(c_i) = c_j$ if $i + 1 = j \mod n$ with the strategy obtained by computing the attractor set and solving the subgame. When eventually no even cycle remains, then all possible plays in the remaining subgame necessarily have odd dominant priority and player Odd wins from the

remaining vertices with a trivial strategy, since by definition of a single-player game Odd has no choice in the game.

The question now becomes how to find these even cycles. If we call a cycle with dominant priority i an i-cycle, then a game contains an i-cycle if and only if it contains some vertices with priority i lying on a cycle after removal of all edges incident with vertices of priority less than i, because an i-cycle can only include edges between vertices of priority i or greater. To find an i-cycle in a graph with edges between vertices of priorities i or greater, we can use the connection between strongly connected components of the graph and cycles in the graph: every cycle must lie in a single strongly connected component and if the edge set of a strongly-connected component is non-empty, then all vertices in the strongly-connected component must lie on a cycle (by the definition of strongly connected components).

To find an even cycle, then, it suffices to search for a i-cycles for all even values of i and for each value construct a graph with edges incident only to vertices of priority i or greater, which is then decomposed into strongly connected components. If a vertex with priority i exists in a component which contains at least one edge, then a cycle can be found with a backtracking search within the component, which will visit every edge in the component at most once.

Because identifying strongly connected components takes $O(E)$ time (for example, using Tarjan's algorithm, described in [37]) and subgame construction typically takes $O(E)$ time as well, it would not be very efficient to remove attractor sets one cycle at a time. Instead, after decomposing the graph for priority i, we can search for one cycle per component and compute the attractor set of all these cycles combined to remove all i-cycles from the game at once. This way, the algorithm requires at most $\lceil \frac{d}{2} \rceil$ iterations and in the worst case $O(|E|\, d)$ total time.

2.4.2 Verification algorithm

When verifying strategies, we are only interested in vertices that are in the winning set of the player that controls them. To verify the set W_x with optimal strategy σ_x for player x, we first define a graph with vertices limited to W_x and the set of edges $E|\sigma_x$ as follows:

$$E|\sigma_x = \{vw \in E : v \in (W_x \cap V_x) \wedge \sigma_x(v) = w\} \cup \{vw \in E : v \in (W_x \cap V_{1-x})\}$$

Informally, the edge set includes the edges that are consistent with x's strategy, as well as all edges originating at vertices controlled by his opponent. We must first check that $E|\sigma_x \subseteq W_x \times W_x$ (otherwise, either player x or his opponent would move the token outside W_x in which case it cannot be a correct winning set).

Assuming this property holds, then $\Gamma|\sigma_x = (W_x \cap V_x, W_x \cap V_{1-x}, E|\sigma_x, \phi|W_x)$ is a proper subgame of Γ, and precisely those plays in the original game consistent with strategy σ_x are possible in the game $\Gamma|\sigma_{\mathrm{x}}$ as well, except that all choice for player x has been removed, which makes $\Gamma|\sigma_x$ a single-player game controlled by player $1 - x$.

To verify that the original strategy was sound, we can solve this single player game using the method described earlier, and verify that the winning set for player W_{1-x} in the subgame is empty. This proves that the strategy σ_x is winning in W_x though it does not yet prove that W_x is maximal. To show that, we must verify that $W_{x-1} = V \setminus W_x$ is also a winning set for the opponent $x - 1$.

In conclusion, complete verification of winning sets and strategies requires solving two single-player games, each of which takes $O(|E|\, d)$ time (as described above). Since $d \leq |V|$ the verification algorithm runs in polynomial time and is fast in practice when the game graph is sparse and the number of distinct priorities is low, as is often the case.

Chapter 3

Common algorithms and data structures

The results that will be presented later on are based on empirical evaluation of various parity game solving algorithms on a variety of test cases. The results obtained therefore depend not only on the test cases used and the choice of algorithms implemented, but also on various implementation details, such as the data structures and programming techniques used to implement those algorithms.

Different tools typically exhibit different performance characteristics in practice, even when they are based on the same algorithms. This occurs due to differences in implementation that are often left unspecified when new algorithms are published. In these publications analysis of the theoretical properties of algorithms takes precedence, and as a result, various design decisions that do affect performance in practice are left up to the implementer.

To ensure that the results presented in this report are reproducible, and to make the differences in results obtained with different tools easier to understand and explain, I will document the choices that I made in the implementation of my solver in more detail than would be expected in a typical research paper. In particular, the core data structures and the algorithms will be documented precisely.

Finally, the descriptions provided here may aid in the understanding of the source code of my solver tool.

3.1 Parity games

3.1.1 Requirements

Recall that a parity game consists of a directed graph, a partition of vertices into sets owned by the two players, and the assignment of a priority to every vertex. This data must be represented in some way in a solver.

When executing a solving algorithm, the parity game data is read, but usually not modified. Therefore, an implementation that allows efficient read-only access is more important than a data structure with high flexibility in regards to updates. However, many of the simplification and preprocessing algorithms must either modify the parity game under consideration or be able to quickly construct a modified copy of it. This use case must be accommodated as well.

Finally, since practical instances of parity games tend to be fairly large, it is desirable that the parity game representation is as compact as possible, to the extent this is possible without compromising access speed. This not only reduces the amount of memory needed to solve particularly large instances, but may also help solving algorithms take advantage from available cache memory. These benefits may come from spatial locality of reference (when parts of the game that

are stored in contiguous memory are accessed in a linear fashion) as well as from temporal locality of reference (when algorithms tend to revisit recently-accessed parts of the game).

3.1.2 The game graph structure

A parity game is played on a directed graph, which consists of a set of vertices (V) and a set of edges ($E \subseteq V \times V$). Vertices will be identified with integers from 0 through $|V|$ (exclusive). At a minimum, we will store $|V|$ and $|E|$, the number of vertices and edges in the graph respectively.

To represent the graph in its entirety, we then only need to store the edges. We could store those as an array of pairs of integers (the source and destination vertices of a directed edge). This is reasonably compact (requiring $2\,|E|$ integers to be stored). However, this representation is impractical if we want to quickly access a set of successors (vE) or predecessors (Ev) of a vertex, which are common operations in many algorithms.

Therefore, a different representation is used. Suppose we start with the array of edges described above and sort them by source vertex first, and destination vertex second. Then, all the edges from a vertex v to its successors will occur as a consecutive sequence in the edge array, and we can store for each vertex the interval [`succBegin`[v], `succEnd`[v]).

This representation would require $2\,|E| + 2\,|V|$ integers to be stored, and allows the following operations to be performed efficiently:

1. Enumerate the successors of a vertex (vE), in order.
2. Calculate the number of successors of a vertex ($|vE|$), by calculating `succEnd`[v]−`succBegin`[v].
3. Determine if $vw \in E$ (using binary search, this takes $O\,(1 + log\,(|vE|))$ time).

Of course, the first operation is the one that is most common. We can apply two further simplifications. First, since the predecessor vertex of all edges with indices between `succBegin`[v] and `succEnd`[v] are known to be equal to v, we don't need to store predecessor vertices at all. Additionally, it is easy to see that `succEnd`[v] = `succBegin`[v + 1] for all v except the last vertex, so we can store all indices in a single array of length $|V| + 1$, instead of using two arrays.

This is the final representation that is used, and requires $|E| + |V| + 1$ integers to store the edge data. However, this edge representation only allows us to quickly find successors of edges. For some algorithms, it is useful to be able to find predecessors quickly as well. For this reason, the graph data structure by default stores the edge set in reverse order too, doubling the amount of memory required.

It should be noted that this dense edge representation does not allow efficient insertion or removal of individual edges in the game graph, because each such operation requires a large part of the edge array to be moved. Fortunately, the algorithms implemented in the solver can be applied to the graph as a whole, and the cost of individual changes can therefore be amortized over the entire graph-wide operation.

3.1.3 The parity game structure

In addition to the game graph, a parity game must store two attributes for each vertex:

1. The controlling player (Even or Odd), and
2. the associated priority value, $\phi(v)$.

These two attributes are packed into a structure, and stored in an array of length $|V|$. (To keep the representation compact, the maximum priority is limited to 65535, which can be stored in a 16-bit integer. High priority values occur very rarely in practical cases.)

Additionally, we store in the parity game structure two properties of the game:

1. The priority limit (d) which is calculated as the maximum priority value used + 1. (This is equal to the index of the game assuming all priority values are used.)

2. An array of integers of length d that stores how many vertices occur with each individual priority value.

This information can be recomputed from the vertex attributes in time $O(|V|)$, but it is useful in a number of situations, for example, to quickly calculate the worst-case execution time of the Small Progress Measures algorithm or to quickly determine whether priority compression is possible.

3.1.4 The solution structure

Every solving algorithm needs to return a solution to the given parity game, which consists of a partitioning of the vertex set into winning sets for both players, and a strategy for each player which is defined at least for vertices in the winning set of that player.

This characterization shows that there is a strong relation between winning sets and strategies of players: when a player controls a vertex which lies outside his winning set, he has no meaningful strategy there (as every possible move is by definition losing). Therefore, we will simply define solutions as arrays which assign to every vertex the successor vertex for the controlling player, or the special value -1 if it is in his opponent's winning set instead:

$$\texttt{solution}[\texttt{v}] = \begin{cases} \sigma_0(v) & \text{if } v \in V_0 \cap W_0 \\ \sigma_1(v) & \text{if } v \in V_1 \cap W_1 \\ -1 & \text{if } v \in (V_0 \cap W_1) \cup (V_1 \cap W_0) \end{cases}$$

From a solution array, winning sets and strategies can be trivially obtained as follows:

$$W_x = \{v \in V_x : \texttt{solution}[\texttt{v}] \neq -1\}$$

$$\sigma_x(v) = \begin{cases} \texttt{solution}[\texttt{v}] & \text{if } v \in W_x \\ \min(vE) & \text{if } v \notin W_x \end{cases}$$

Note that the choice of the minimum successor for vertices which are lost to the current player is arbitrary; in those cases any successor could be chosen.

3.1.5 Subgame construction

Many algorithms require subgames to be constructed. Since the data structure described above requires a dense representation of vertices, this requires that all data is reconstructed. The subgame $\Gamma|U$ is constructed from an array containing the vertex identifiers in U while the ordering of the elements determines the new identifiers of the vertices in the subgraph.

The computational cost of constructing a subgame is dominated by construction of the successor and/or predecessor arrays of the subgraph, which is done by iterating over all successors (or predecessors, as the case may be) of vertices in U, filtering out vertices which are outside of U. If the ordering of vertices in U differs from $\overline{V}$ then vertex lists need to be resorted too, but usually this is not necessary, as in most cases U is constructed as a subsequence of V.

To filter vertices efficiently, either a hash table or a Boolean array is used to represent U, depending on the size of U relative to V: for small subsets the mapping is sparse and a hashtable is more efficient. The exact time complexity depends on the outdegree for vertices in U; as long as the outdegree of the game and its subgame are equal, time required is proportional to the number of edges in the subgame. To analyse the complexity of algorithms that rely on subgame construction (which includes Zielonka's algorithm, and many preprocessing operations) we will generally assume this property holds.

Finally, it should be noted that when constructing subgames, the original array U may be kept around to be able to map the vertex identifiers in the subgame back to the original game, which is necessary to propagate solution and strategy information from a subgame back to the main game (as well as to gather global statistics and debugging information).

3.1.6 Attractor set computation

Attractor sets can be calculated straightforwardly using a queue and a set, where both data structures are assumed to have $O(1)$ performance for the relevant operations (for example, a deque and a hash table satisfy these requirements). The queue is initialized to the attractor set. Every iteration, a vertex is extracted from the queue and its predecessors are examined. Any predecessor which is controlled by the player for whom we are computing the attractor set, or which is controlled by his opponent but has no successors outside the attractor set, is added to both the queue and the attractor; typically, a strategy is assigned to this vertex as well. Attractor set computation works similar to a breadth-first search, and as a consequence the strategy reflects the shortest path from vertices in the extended attractor set to vertices in the initial set. The algorithm is presented in detail as Algorithm 3.1.

This algorithm is reasonably efficient for sufficiently sparse graphs, but it has two drawbacks. First, it requires both predecessor and successor information to be stored in the graph. Second, the worst-case complexity is as high as $O(|E|\,|V|)$, even for sparse graphs. This complexity arises from the fact that when a predecessor is controlled by the opponent of the player for whom the attractor set is computed, then all successors of this vertex must be evaluated to see if an edge to a vertex outside the attractor set exists; in a worst-case scenario, all vertices currently in the attractor set could be reachable before a successor outside it is found.

Fortunately, there is a way to mitigate this problem. Let's call the successors of a vertex which lie outside the attractor set its *liberties*. We can store the number of liberties for each vertex in a simple array of integers, which is initialized to the outdegree of each vertex, and updated whenever a vertex is found to be in the attractor set (by decrementing the count for each of its predecessors). This eliminates the need to examine successors at all, as Algorithm 3.2 shows, and reduces the time complexity to $O(|E|)$.

Note that for the second algorithm, the $O(|E|)$ bound is tight, while time required for the first algorithm correlates with the size of the attractor set calculated, and consequently the first implementation can outperform the second in practice. However, the second algorithm has the additional benefit of not requiring successor information to be stored in the graph. For algorithms which rely heavily on subgame construction, this can result in an additional performance improvement.

(Note: the worst-case time complexity of the first algorithm can also be reduced to $O(|E|)$ by tracking the first successor outside the attractor set for each opponent-controlled vertex. However, this does not remove the dependence on successor edges.)

Algorithm 3.1 Attractor set computation

```
1  Set<Vertex> make_attractor_set(
2      ParityGame game, Player player,
3      Set<Vertex> initial, Strategy s )
4  {
5      Queue<Vertex> todo
6      Set<Vertex> attr
7      for v : initial {
8          attr.insert(v)
9          todo.push_back(v)
10     }
11     while not todo.empty() {
12         Vertex v = todo.pop_front()
13         for u : game.graph.predecessors(v) {
14             if u in attr { continue }
15             if game.player(u) == player {
16                 strategy[u] = v
17             } else if game.graph.successors(u) is subset of attr {
18                 strategy[u] = NO_VERTEX
19             } else {
20                 continue
21             }
22             attr.insert(u)
23             todo.push_back(u)
24         }
25     }
26     return attr
27 }
```

Algorithm 3.2 Attractor set computation

```
1  Set<Vertex> make_attractor_set(
2      ParityGame game, Player player,
3      Set<Vertex> initial, Strategy s )
4  {
5      Vector<Integer> liberties = { 0, 0, 0 .. }
6      for v : game.graph.V {
7          for u in game.graph.predecessors(v) {
8              liberties[u]++
9          }
10     }
11     Queue<Vertex> todo
12     Set<Vertex> attr
13     for v : initial {
14         attr.insert(v)
15         todo.push_back(v)
16         liberties[v] = 0
17     }
18     while not todo.empty() {
19         Vertex v = todo.pop_front()
20         for u : game.graph.predecessors(v) {
21             if liberties[u] == 0 { continue }
22             if game.player(u) == player {
23                 strategy[u] = v
24                 liberties[u] = 0
25             } else {
26                 liberties[u]--
27                 if liberties[u] == 0 {
28                     strategy[u] = NO_VERTEX
29                 } else { continue }
30             }
31             attr.insert(u)
32             todo.push_back(u)
33         }
34     }
35     return attr
36 }
```

3.1.7 Graph decomposition

To compute strongly connected components, Tarjan's algorithm from [37] is used, which enumerates all strongly connected components in reverse topological order, which means that when a component is found all edges starting at a vertex in the component end at vertices either inside the same component, or in one of the components found before. The algorithm requires $O(|E|)$ time and $O(|V|)$ space.

The algorithm as described by Tarjan is based on a depth-first search of the graph, labelling vertices as they are visited. Depth-first search is typically implemented as a recursive procedure, but this is impractical for very large graphs, where high recursion depth may exhaust the available stack space. Therefore, my implementation uses an iterative approach, with an explicit stack data structure which is stored in the heap. In addition to removing the limitations on recursion depth, this reduces memory use substantially, because the on-heap state representation (consisting of a vertex and edge index) is much more compact than a full stack frame would be.

Components are passed to a callback function as an array of indices of vertices in the original graph. Such an array can be used to construct a subgame for the component, as described above, though it should be noted that the subgame induced by a strongly connected component of the game graph is not necessarily a proper subgame itself.

The decomposition algorithm is used to solve individual components separately (see Subsection 2.3.3), removal of winner-controlled cycles (see Subsection 4.3.3.2) and verification of games (see Section 2.4).

Chapter 4

Small Progress Measures

Small Progress Measures (or SPM for short) is a relatively simple, iterative algorithm for partially solving parity games proposed by Marcin Jurdziński. A game is solved only partially in the sense that the winning set and optimal strategy for one player is determined. To solve a game completely, the algorithm must therefore be run twice, but, fortunately, in the second pass the winning set of the first player can be omitted from the game graph, which typically reduces the time required to solve the remaining part of the game significantly.

The core algorithm is simple and allows for a straightforward implementation, but also provides ample opportunity for performance improvement (for example, by parallelizing the core algorithm). Additionally, the algorithm has one of the lowest worst-case complexity bounds known for solving parity games (excluding Schewe's big step approach, which invokes SPM recursively) requiring at most $O\left(|E|\left(|V|/\left\lfloor\frac{d}{2}\right\rfloor\right)^{\left\lfloor\frac{d}{2}\right\rfloor}\right)$ time and $O(|V|\,d)$ space (in addition to the space required to store the parity game itself). Unfortunately these are also lower bounds on the worst case.

Oliver Friedmann implemented a variation of the algorithm in PGSolver that effectively combines the two passes in one, solving both the game and its dual at the same time. This does not improve on the worst-case time bounds (and, in fact, may require around twice as much time and memory compared to the standard algorithm), but can avoid some of the pitfalls that cause excessive running times with the standard algorithm, which makes it a useful alternative in practice. Since this variation has not been published before, it will be described in subsection 4.3.4.

A lock-free concurrent version was implemented by Van de Pol and Weber which works on shared-memory systems that do not reorder memory operations. A concurrent implementation for the PlayStation 3 (taking advantage of the capabilities of the multi-core Cell processor) was written by Jorne Kandziora [25] and later improved upon by Freark van der Berg. [39] A parallel implementation for NVIDIA-based GPUs using CUDA is evaluated in [5].

My contributions relevant to Small Progress Measures include:

- An improvement of the basic algorithm that eliminates failed lifting attempts, described in subsection 4.3.1.
- An optimization that reduces the codomain for progress measures whenever a vertex is lifted to $\top$, described in subsection 4.3.2.
- The development of a novel lifting heuristic that reduces the number of lifting attempts required in many cases, described in subsection 4.2.5.
- As a practical result: implementing these improvements combined into a single tool.

4.1 Description

In this section the Small Progress Measures algorithm will be described as outlined by Jurdziński [23], using the same function names and similar definitions for consistency, but without the accompanying correctness proof, for which the reader is referred to the original paper.

Let's assume, without loss of generality, that we want to solve a parity game for player Even. First, a progress measure is defined as a function $\rho : V \to (\mathbb{N}_0^* \cup \top)$; a function which assigns to each vertex either a vector of nonnegative integers, or the special value $\top$ (top). (Lower-case Greek letters will be used to denote elements of $\mathbb{N}_0^* \cup \top$.)

On these values a comparison operator $<_i$ is defined that compares two vectors lexicographically up to (and including) the element with index i, with the special value $\top$ being considered greater than any other value. Formally:

$$\begin{array}{rl} \alpha <_i \beta \Leftrightarrow \alpha \neq \top \wedge \beta = \top & \text{if } \alpha = \top \text{ or } \beta = \top \\ \exists j \leq i : \alpha_j < \beta_j \wedge (\forall k < j : \alpha_j = \beta_j) & \text{otherwise} \end{array}$$

The operator $<_i$ establishes a strict weak ordering on vectors of length i or greater. The other operators can then be defined accordingly:

$$\alpha >_i \beta \Leftrightarrow \beta <_i \alpha$$

$$\alpha \leq_i \beta \Leftrightarrow \neg(\alpha >_i \beta)$$

$$\alpha \geq_i \beta \Leftrightarrow \neg(\alpha <_i \beta)$$

$$\alpha =_i \beta \Leftrightarrow (\alpha \leq_i \beta) \wedge (\alpha \geq_i \beta)$$

$$\alpha \neq_i \beta \Leftrightarrow (\alpha <_i \beta) \vee (\alpha >_i \beta)$$

For a finite parity game, the range of the priority function has a finite maximum too; without loss of generality, we can assume this range is strictly bounded by the game index d (where d is the number of priorities in the game). A progress measure is called *small* if its range is limited to the finite set $M^\top = M \cup \top$, where $M = M_0 \times M_1 \times \cdots \times M_{d-1}$ and M_i is defined as:

$$M_i = \begin{cases} \{0\} & \text{if } i \equiv 0 \mod 2 \\ \{0..\left|\phi^{-1}(i)\right|\} & \text{if } i \equiv 1 \mod 2 \end{cases}$$

The eventual goal of the algorithm is to calculate a particular progress measure which maps vertices won by Odd to $\top$ and vertices won by Even to non-$\top$ values. (Jurdziński proves that such a progress measure exists.) A few helper functions are needed to specify how such a progress measure is calculated.

For a small progress measure ρ and an edge $vw \in E$, the function $\mathit{Prog}(\rho, v, w) : (V \to M^\top) \times E \to M^\top$ determines a minimum value than can be assigned to v if the edge vw is included in the strategy for whichever player controls v, as follows:

$$\mathit{Prog}(\rho, v, w) = \begin{cases} \top & \text{if } \rho(w) = \top, \text{ otherwise:} \\ \min_{m \in M^\top} m \geq_{\phi(v)} \rho(w) & \text{if } \phi(v) \equiv 0 \mod 2, \\ \min_{m \in M^\top} m >_{\phi(v)} \rho(w) & \text{if } \phi(v) \equiv 1 \mod 2. \end{cases}$$

This function is used to define the function $\mathit{Lift}(\rho, v) : (V \to M^\top) \times V \to (V \to M^\top)$, which updates a progress measure ρ to reflect the optimal strategy for the controlling player of v:

$$\mathit{Lift}(\rho, v)(u) = \begin{cases} \rho(u) & \text{if } u \neq v \\ \max\{\rho(v), \min_{vw \in E} \mathit{Prog}(\rho, v, w)\} & \text{if } u = v \ \wedge \ v \in V_0 \\ \max\{\rho(v), \max_{vw \in E} \mathit{Prog}(\rho, v, w)\} & \text{if } u = v \ \wedge \ v \in V_1 \end{cases}$$

Note that this definition reflects that Even will try to minimize the progress measure while Odd tries to maximize it. Similarly, the function $\mathit{Lift}(\rho, v) : (V \to M^\top) \to (V \to M^\top)$ updates a progress measure at any one vertex, if possible:

$$\mathit{Lift}(\rho) = \begin{cases} \mathit{Lift}(\rho, v) & \text{if } \exists v \in V : \ \mathit{Lift}(\rho, v) \neq \rho, \\ \rho & \text{otherwise} \end{cases}$$

There may be different vertices v that satisfy the precondition on the first clause; in that case, any of the possible vertices may be chosen. Since $\mathit{Lift}(\rho)$ is a monotone function on a complete lattice, it has a unique minimum fixed point that can be found by iterative application starting from the minimum element, the zero progress measure, $\rho(v) = \vec{0}$ for all $v \in V$. Jurdziński proves that this minimum fixed point encodes both the winning sets and an optimal strategy for player Even, as follows:

$$W_0 = \{v \in V : \ \rho(v) < \top\}$$

$$W_1 = \{v \in V : \ \rho(v) = \top\}$$

$$\sigma_0(v) = w \text{ if } v \in V_0 \cap W_0 \ \wedge \ vw \in E \ \wedge \ \rho(v) =_{\phi(v)} \rho(w)$$

In the last definition, the choice of w may not be unique (though there necessarily must be one appropriate successor), and in that case any of the possible successors may be arbitrarily chosen to obtain an optimal strategy.

4.2 Lifting strategies

Jurdziński's algorithm is straightforward, but it does not explicitly address two implementation details:

1. How can we determine which vertices are liftable ($\mathit{Lift}(\rho, v) \neq \rho$)?
2. When multiple vertices are liftable, which should we select to improve the progress measure?

A simple answer to the first question is to try computing $\mathit{Lift}(\rho, v)$ for different values of v until either one is found that changes the current progress measure, or, when all options are eliminated, conclude that a fixed point has been found. All previous implementations (see introduction of this chapter) follow this approach, which has the benefit that it's simple and sound. However, it's not clear that this leads to best performance in practice. In Subsection 4.3.1 a better approach will be presented.

Jurdziński leaves the second question unanswered because any lifting order will lead to the same, unique fixed point. For the correctness of the algorithm, any selection policy will do. However, the order in which vertices are lifted does affect the runtime of the algorithm in practice. Since it is difficult (if not entirely impossible) to predict an optimal lifting sequence in advance, a heuristic strategy must be used instead. Jurdziński already hints at the importance of using a suitable lifting strategy (which he calls "evaluation policy"): *"our algorithm can be seen as a generic algorithm allowing many different evaluation policies; good heuristics can potentially improve performance of the algorithm".*

This raises the question how large the gap between good and bad policies actually is in practice. Jurdziński gives an upper bound for the total number of lifts in a game by simply calculating the size of the small progress measure set:

$$|V| \cdot |M| = |V| \cdot \prod_{i=1}^{\lfloor d/2 \rfloor} \left(\left| \phi^{-1}(2i-1) \right| + 1 \right) \leq \left(\frac{n}{\lfloor d/2 \rfloor} \right)^{\lfloor d/2 \rfloor}$$

This is useful to establish the worst-case complexity but it is also an overapproximation: only the vertices with greatest priority values can be lifted $|M|$ times. A more accurate characterization takes into account the priorities of the vertices and how often they occur:

$$\prod_{p=0}^{d-1} \sum_{q=0}^{p} |M_q| \left| \phi^{-1}(p) \right|$$

This is a tighter bound, but it can only be approached in specially crafted graphs, even with the worst possible lifting strategies. How much can be gained compared to the *worst* lifting strategies is not even very interesting; after all, we would never intentionally use a bad lifting strategy. The more interesting question to investigate is how much we can improve upon *trivial* lifting strategies (which may already work reasonably well in practice).

As will be discussed in Subsection 4.2.2.1, it turns out that we can bound the overhead of a naive lifting strategy to a factor $|V|$ in lifting attempts. Although this limits the gains we can expect from more sophisticated approach, the vertex set V is large enough in practice to justify trying to minimize the lifting sequence.

4.2.1 Core algorithm

A *lifting strategy* is a heuristically algorithm that, depending on information available about a partial solution (such as the structure of the graph, the values of the progress measure vectors, or the history of vertices being lifted) determines which vertex will be selected for the next lifting attempt. Lifting strategies are typically stateful, and although sophisticated lifting strategies may cause fewer work to be done in the SPM algorithm itself (by reducing the number of lifting attempts performed) it is important to keep in mind that this may come at the cost of spending more time and memory on internal bookkeeping.

To talk about lifting strategies and their efficiency, it is useful to establish some terminology. A vertex for which $\mathit{Lift}(\rho, v) = \rho$ is called stable (with respect to progress measure ρ), and unstable if $\mathit{Lift}(\rho, v) \neq \rho$. A lifting attempt at vertex v consists of calculating $\mathit{Lift}(\rho, v)$ and is called successful if $\mathit{Lift}(\rho, v) \neq \rho$ (and we can update our progress measure) and unsuccessful (or failed) otherwise. In other words: the set of stable vertices corresponds to the set of vertices at which lifting attempts would succeed. Finally, a vertex is called *dirty* if it has not been lifted since one of its successors has been lifted, and clean otherwise. As a result, unstable vertices are a subset of dirty vertices, and the difference is relevant because all previous implementations track *dirty* but not *unstable* vertices explicitly, and must attempt lifting of dirty vertices to determine whether they are stable or not.

The core algorithm can be described as a function taking a small progress measure that is appropriately initialized and a lifting strategy; see Algorithm 4.1. In this arrangement, the core algorithm notifies the lifting strategy of successful lifts, and the lifting strategy decides both the order in which vertices are lifted, and signals termination when the minimum fixed point has been reached.

This separation of concerns exists in most previous implementations of Small Progress Measures and is attractive because it keeps the core algorithm very simple, while still allowing a sophisticated approach to be used by the lifting strategy. Of course, this also means that the performance of the algorithm depends crucially on the strategy chosen.

In the following, I will describe the most commonly used lifting strategies, and how I implemented them in this framework. After that, I will describe some of the improvements I made to the core algorithm as outlined above.

Algorithm 4.1 Core Small Progress Measures Algorithm

```
1  SmallProgressMeasures(spm, lifting_strategy)
2  {
3      v = lifting_strategy.next()
4      while v != NO_VERTEX {
5          new_spm = lift(spm, v)
6          if spm[v] > new_spm[v] {
7              lifting_strategy.lifted(v)
8              spm[v] = new_spm[v]
9          }
10         v = lifting_strategy.next()
11     }
12 }
```

Algorithm 4.2 Linear Lifting Strategy

```
1  LinearLiftingStrategy
2  {
3      num_failed = 0
4      next_vertex = 0
5
6      lifted(Vertex v) {
7          num_failed = 0
8      }
9
10     Vertex next() {
11         if num_failed >= V {
12             return NO_VERTEX
13         } else {
14             num_failed = num_failed + 1
15             next_vertex = (next_vertex + 1) mod V
16             return next_vertex
17         }
18     }
19 }
```

4.2.2 Linear lifting strategy

Arguably the most basic strategy consists of trying to lift vertices in order of increasing indices, repeating the process as necessary, until all vertices in the graph have failed to be lifted in succession. Its advantages are that it is simple to implement and requires very little time and memory to select vertices. The main disadvantage is that it may spend a majority of the time attempting (and failing) to lift vertices in stable parts of the game graph. Algorithm 4.2 gives pseudo-code for a simple implementation of this algorithm.

(In Van de Pol and Weber's implementation, this approach is called *swiping*.)

A variation of the strategy is to iterate vertices in reverse order. This can be beneficial in cases where the graph has been constructed in a forward fashion, so that vertices tend to have higher indices than their predecessors, and therefore since changes in the graph propagate from vertices to their predecessors, iterating in reverse may cause more vertices to be lifted in one pass. For example, in a cycle graph with vertices numbered in cycle order, iterating in forward direction would lift only one vertex per pass, while lifting in backward direction would lift all.

There are a few things we can do to minimize the number of passes executed:

1. Calculate the ratio of "forward" and "backward" edges and choose the optimal direction

accordingly.

2. Permute vertex indices based on the order they are visited by depth-first or breadth-first search through the graph, which should create a more favourable edge ratio.

3. Alternate the direction of iteration on every pass.

The advantage of the third approach is that it does not require precalculation/preprocessing of the game graph. Compared to the first approach, it will create at most twice as much work. Permuting vertex indices can produce even better results, but even the second approach isn't guaranteed to yield optimal results.

4.2.2.1 Efficiency of the linear lifting strategy

An interesting observation about the linear lifting strategy is that it can be used to put not just an upper bound, but also a lower bound on the minimum number of lifting attempts which are necessary to solve a game.

For a minimal-length sequence of vertices which can be lifted to reach the minimum fixed point, it is permissible to insert extra vertices anywhere; the resulting sequence will still result in a solution. Since the linear lifting strategy (unlike some of the other, more sophisticated strategies) guarantees that between any pair of vertices in the optimal sequence strictly less than $|V|$ other vertices will be lifted, the total number of vertices that must be lifted is at least equal to the total number of lifting attempts (successful or not) performed by the linear lifting strategy divided by $|V|$.

Consequently, an optimal lifting strategy could improve upon the linear lifting strategy by at most a factor $|V|$.

4.2.3 Predecessor lifting strategy

A more sophisticated lifting strategy takes into account that a known-stable vertex can only become unstable after the progress measure vector of one of its successors has been updated. After all, the value for one vertex depends only on the values for its successors.

The predecessor lifting strategy uses a set data structure to store potentially unstable ("dirty") vertices, which is initialized to all vertices in the graph with progress measures less than $\top$. A vertex is arbitrarily removed from this set to be lifted, and if this succeeds, its predecessors are added back into the set. All vertices not in the set are necessarily stable, so when the set becomes empty, lifting is complete.

The advantage of this strategy is that it prevents a lot of unnecessary lifting that may occur with the linear lifting strategy. In fact, the ratio of unsuccessful to successful lifts is strictly limited by the maximum indegree of a vertex in the graph. Again, pathological cases can be constructed where this strategy performs poorly. For example, when the game graph is complete (i.e. $E = V \times V$) all vertices remain in the set until the game is completely solved. Fortunately, graphs with such properties arise rarely in practice.

The main disadvantage of the lifting strategy is that it requires maintaining the set of unstable vertices, which requires $O(|V|)$ extra memory, and extra time proportional to the average indegree of lifted vertices. In my solver, this set is implemented with a linear array of Boolean variables to indicate whether a vertex is in the set, and another array which acts as a circular queue of vertices, in order to allow constant time insertion and removal of vertices.

In PGSolver (which is limited to single-threaded operation) the predecessor lifting strategy is the only lifting strategy implemented. Van de Pol and Weber call this the *work list approach* but reject it in their multi-core solver because they are worried about the overhead involved in maintaining the necessary data structures in a concurrent setting. (This seems a bit premature: the predecessor lifting strategy seems simpler than the focus list approach described below, and the necessary data structures can easily be implemented in a lock-free manner.)

Algorithm 4.3 Predecessor Lifting Strategy

```
1  PredecessorLiftingStrategy
2  {
3      queued = Vector<bool>(V)
4      queue = Queue<Vertex>()
5      for v in [0..V) {
6          if not top(w) {
7          queued[v] = true
8             queue.push(v)
9         }
10     }
11
12     lifted(Vertex v) {
13         for w in predecessors(v) {
14             if not queued[w] and not top(w) {
15                 queued[w] = true
16                 queue.push(w)
17             }
18         }
19     }
20
21     Vertex next() {
22         if queue.empty() {
23             return NO_VERTEX
24         } else {
25             v = queue.pop()
26             queued[v] = false
27             return v
28         }
29     }
30 }
```

The pseudo-code for the predecessor lifting strategy is given in Algorithm 4.3 which shows that the main data structure used is a queue. Crucial to the implementation is that every vertex is queued at most once, which prevents attempted lifting of vertices which are not dirty, and additionally allows the queue to be implemented as a fixed-length array. An invariant maintained by this algorithm is that the set of vertices queued corresponds exactly to the set of dirty vertices; `next()` simply returns a dirty vertex until the queue becomes empty, at which point all vertices are necessarily stable.

There is some freedom of implementation when deciding which vertex from the set of potentially unstable vertices to lift next. My implementation uses no external information, but can be configured to extract vertices from the queue in a first in, first out (FIFO) or last in, last out (LIFO) manner, in which case the queue works like a stack. Extracting vertices in LIFO order may have the benefit of providing better cache locality, even when it doesn't reduce the number of lifts. The benchmark results presented later will show the performance difference between the two policies in practice.

4.2.4 Focus list approach

The *focus list approach* is a strategy designed by Van de Pol and Weber to mitigate some of the problems of the linear lifting strategy without introducing complicated data structures that may be hard to parallelize efficiently. Solving occurs by alternation of two phases of operation.

First, the algorithm iterates linearly over the vertices of the graph, putting vertices which are successfully lifted on a fixed-size work list. When the work list is full, the second phase begins: vertices on the work list are assigned an initial credit value and are selected for lifting again. When lifting a vertex succeeds, its credit is incremented linearly (adding a constant); if it fails, it is decreased exponentially (dividing by a constant). A vertex whose credit drops below a certain threshold is removed from the work list. When the work list is empty or a predetermined amount of lifting attempts have been performed, the work list is cleared and the first phase is resumed.

The idea behind the work list is that it captures unstable parts of the graph. When processing the work list, either lifting attempts will have a high success rate, or the list will be cleared quickly, allowing new vertices to be selected. Limiting the number of attempts at this time ensures that if little progress is made in one part of the game (i.e. despite high lifting success rate, vertices do not stabilize) work will soon continue on some other part of the game.

Since Van de Pol and Weber were the only ones to implement this strategy, there is not much empirical evidence available about its performance, though they report in [38] that it performs much better than linear lifting in some cases, and not much worse in most other cases, which suggests it meets its design goals. Compared to the predecessor lifting strategy, it has the advantage of being able to postpone difficult parts of the game (by recycling the work list when it is not cleared quickly enough) which may help in some hard cases.

The main downside of the strategy is that it depends on various constants (work list size, initial credit, credit increment, credit divisor) to work well. In Weber's implementation these are empirically derived while testing on random games, but it is unclear if and how these values should change for other types of games.

The implementation of the focus list lifting strategy is described in Algorithm 4.4. For clarity, the algorithm is presented as if the strategy calls the `lift()` method directly, instead of being called from the core algorithm; this is the inverse of how the strategy is implemented in reality. (The interaction between the core algorithm and the lifting strategy would best be implemented as a pair of coroutines, but unfortunately few programming languages support coroutines properly.)

Some of the parameters in this algorithm are fixed: the initial credit (here: 2) assigned to vertices on the focus list; the linear increment when lifting succeeds (again, 2); and the exponential decay when it doesn't (here: $\frac{1}{2}$). With these parameters, which were taken from van de Pol and Weber's implementation, a vertex is selected for lifting at least three times, but will stay on the focus list longer if lifting succeeds occasionally (more precisely, it needs to be successfully lifted at least once every three attempts to stay on the list). The remaining two parameters are `max_size` and `max_attempts`; the maximum size of the focus list and the maximum number of lifting attempts

Algorithm 4.4 Focus List Lifting Strategy

```
1  FocusListLiftingStrategy
2  {
3      phase = 1
4      num_attempts = 0
5      num_failed = 0
6      next_vertex = 0
7      focus_list = Queue<Pair<Vertex, Int>>
8
9      while true {
10         num_attempts = num_attempts + 1
11         if (phase == 1) {
12             if lift(next_vertex) {
13                 num_failed = 0
14                 focus_list.push(<next_vertex, 2>)
15             } else {
16                 num_failed = num_failed + 1
17             }
18             next_vertex = (next_vertex + 1) mod V
19             if num_failed == V {
20                 break
21             }
22             if num_attempts == V or focus_list.size() == max_size {
23                 /* N.B. focus_list is non-empty because
24                         num_failed < num_attempts */
25                 phase = 2
26                 num_attempts = 0
27             }
28         } else {  /* phase == 2 */
29             v, credit = focus_list.pop()
30             if lift(v) {
31                 focus_list.push(<v, credit + 2>)
32             } else if (credit > 0) {
33                 focus_list.push(<v, credit / 2>)
34             }
35             if focus_list.empty() or num_attempts == max_attempts {
36                 focus_list.clear()
37                 phase = 1
38                 num_attempts = 0
39             }
40         }
41     }
42 }
```

performed in phase 2 before switching back to phase 1. These parameters can vary, but are chosen as $\frac{|V|}{10}$ and $|V|$ (respectively) by default.

In terms of overhead, the focus list implements a circular buffer with an array of vertex/cost pairs. The size of this array is equal to `max_size` and this is the main factor in the memory used by the strategy. Otherwise, the focus list lifting strategy has little overhead. In particular, the runtime overhead is smaller than with the predecessor lifting strategy. When the focus list is small, the algorithm additionally benefits from being able to keep the active parts of the game in cache memory.

4.2.5 Maximum measure propagation

The maximum measure propagation strategy is a context-aware variant of the predecessor lifting strategy. It returns a vertex from the queue for which the minimum successor (for Even-controlled vertices) or the maximum successor (for Odd-controlled vertices) is maximal over all queued vertices. In other words: it picks a vertex such that the value assigned by a successful lifting attempt is as large as possible. The rationale behind this strategy is that since the final progress measure for a vertex does not depend on the order of lifting, it makes sense to try to propagate higher values first rather than waste time on updating lower values, which might be overwritten later.

It is not difficult to imagine scenarios where this order of lifting provides benefit. For example, whenever a vertex is known to be won by player Odd (when it is assigned $\top$) all vertices in its attractor set will be set to $\top$ before any other vertices are lifted, since $\top$ is the maximal value in the graph. With the normal predecessor lifting strategy, some of these vertices might have been lifted multiple times before inevitably finally reaching top value.

Maximum measure propagation can be viewed as a context-aware variant of predecessor lifting in that it still returns vertices from a queue, but rather than using insertion order to determine which vertex to lift next, it considers the values assigned to vertices so far.

Three variations of this idea are implemented:

1. Maximum measure metric: prioritize vertices with *greatest* successor value (described above)
2. Maximum step metric: prioritize vertices with *greatest difference* compared to their successor
3. Minimum measure: prioritize vertices with *least* successor value

As with predecessor lifting, ties between queued vertices can be broken in two ways based on insertion order. (Predecessor lifting itself can be viewed as the zeroth variant that does not prioritize queued vertices at all.)

A disadvantage of the context-aware lifting strategy is that it incurs more overhead, since a priority queue data structure must be maintained in order to efficiently select vertices for lifting. An in-place binary heap of vertex indices is used for this purpose, while the predecessor lifting strategy required only a circular buffer. The overhead incurred by maintaining the heap can be alleviated somewhat by removing the insertion-order tie-breaking rule (so that queued vertices are only partially ordered).

The in-place heap itself causes no memory overhead compared to a circular buffer, but requires more work to keep it balanced, especially since the proper position of a vertex in the heap depends not only on the value assigned to it, but also on the values assigned to its successor vertices. That means that whenever a vertex is removed from the queue to be lifted, the heap structure may be invalidated at multiple locations. To resolve this, the predecessors of the lifted vertex must be located in the heap and be moved up (for the maximum measure and step metric) or down (for the minimum measure metric) in the heap as necessary to restore the heap property. This necessitates storing the position of queued vertices in the heap. Finally, restoring the heap property requires that vertices are processed in the correct order based on their position in the heap: when moving vertices up, vertices higher in the heap should be processed first.

This process is illustrated in Figure 4.1. The first diagram shows a balanced binary heap. The balance is destroyed in the second diagram after the three labelled nodes have their values

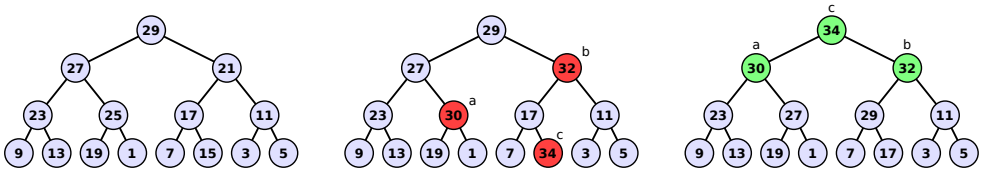


Figure 4.1: Updating the binary heap

increased. To restore the balance it is necessary to move node b up before node c, because b lies on the path from c to the root node. The third diagram shows the heap after rebalancing.

From the above description, it should be clear that maximum measure propagation (like its variants) is quite a bit more complicated than simple predecessor lifting, both in terms of implementation difficulty and the amount of computational overhead added to each lifting attempt. However; this might be worth it if the final progress measure can be computed more quickly. This will be assessed later on.

4.3 Improvements

Apart from choosing better lifting strategies, improvements to the core algorithm are possible too. I will discuss the improvements I implemented below; some are novel, and others have been previously implemented.

4.3.1 Eliminating failed lifting attempts

When implementing some of the more sophisticated lifting strategies (such as the predecessor- and maximum-measure strategies) it became apparent that the core algorithm as described in Algorithm 4.1 (and as implemented in all tools that I know of) is not as efficient as it could be. To determine if a vertex is stable, it must be lifted. In order to lift it, the minimum (or maximum) successor must be found, and its progress measure vector must be compared with that of the predecessor. Many of these comparisons can be avoided if the core algorithm keeps track of the relevant minimum/maximum successor for each vertex.

For example, if there is a vertex v controlled by Even, and one of its successors (say: w) is lifted, this affects v's stability if (and only if) w is the current minimum successor for v. Without retaining this information, however, the only way of knowing whether v's progress measure vector is too low is to re-examine the vectors assigned to each of v's successors.

Failed lifting attempts can be eliminated completely by storing for each vertex whether it is stable and what its current minimum/maximum successor is. This requires $\mathrm{O}(V)$ extra space in the core algorithm, but simplifies lifting strategy implementation. Additionally, at the end of the algoritm, the minimum successors form an optimal strategy for Even.

This observation effectively turns the core algorithm inside out: instead of checking the successors of a vertex to see if it can be lifted, we only lift known-unstable vertices and then check its predecessors to see if any of them are destabilized by the change. This may first appear to be a rather trivial variation, and while it is true it doesn't change the worst-case performance of the algorithm, the fundamental difference is that it eliminates many redundant progress measure vector comparisons.

This idea is described somewhat more formally as Algorithm 4.5. Note that `spm.less_than(v, w)` compares a the progress measure value assigned to v using $\leq_{\phi(v)}$ (if $\phi(v) \equiv 0 \mod 2$) or $<_{\phi(v)}$ otherwise, as is usual to determine if a vertex v is liftable by its successor value w. Like Algorithm 4.1, this procedure uses an object tracking progress measures and a lifting strategy as arguments. However, unlike the former algorithm, it keeps explicit track of stable and unstable vertices, notifying the lifting strategy (with `push`) whenever a vertex becomes unstable. The lifting strategy still has control over the order in which unstable vertices are lifted, but can easily avoid

Algorithm 4.5 Improved Small Progress Measures Algorithm

```
1  SmallProgressMeasures2(spm, lifting_strategy)
2  {
3      stable   = { 0, 0, 0 .. }
4      strategy = { 0, 0, 0 .. }
5
6      for v : game.graph.V {
7          if game.player(v) == 0 {
8              strategy[v] = spm.get_min_vertex(game.graph.successors(v))
9          } else {
10             strategy[v] = spm.get_max_vertex(game.graph.successors(v))
11         }
12         if spm.less_than(v, strategy[v]) {
13             lifting_strategy.push(v)
14         } else {
15             stable[v] = 1
16         }
17     }
18
19     while v = lifting_strategy.next_vertex() {
20         spm.lift_to(v, spm[strategy[v]])
21         stable[v] = 1
22         for u : game.graph.predecessors(v) {
23             if game.player(u) == 0 {
24                 if strategy[u] == v {
25                     strategy[u] = spm.get_min_vertex(game.graph.successors(u))
26                 } else {
27                     continue    // minimum successor unchanged
28                 }
29             } else {
30                 if (strategy[u] == v) or (spm[strategy[u]] < spm[v]) {
31                     strategy[u] = v
32                 }  else {
33                     continue    // maximum successor unchanged
34                 }
35             }
36             if stable[u] {
37                 if spm.less_than(u, strategy[u]) {
38                     stable[u] = 0
39                     lifting_strategy.push(u)
40                 }
41             } else {
42                 lifting_strategy.bump(u)
43             }
44         }
45     }
46     return strategy
47 }
```

trying to lifting stable vertices. To allow advanced lifting strategies (in particular, those based on maximum measure propagation) to keep their internal data structures consistent, the core algorithm also notifies the lifting strategy (with `bump`) when the minimum/maximum successor of an unstable vertex, or the associated progress measure value, changes.

It should be clear that the amount of work done per lifting attempt is roughly equal on average for both formulations of the core algorithm: while the original formulation requires examining all successors of the vertex being lifted, this variant examines its predecessors instead. However, since this variant algorithm eliminates all failed lifting attempts (which make up a considerable part of lifting attempts in most test cases) it is faster in practice.

I will call the earlier formulation of the core algorithm the "standard framework" since it matches Jurdziński's description and is implemented in earlier solving tools, and call the variation presented here the "improved framework" since it turns out (as will be shown in Chapter 6) that moving the responsibility of tracking stable vertices from the lifting strategy to the core algorithm is a strict improvement, at least for lifting strategies that are at least as sophisticated as the predecessor lifting strategy.

Clearly this improved core algorithm is more complicated than the algorithm presented earlier. This disadvantage is ameliorated by the observation that lifting strategies can be made simpler, since the improved framework absolves them of the responsibility to track dirty/stable vertices. Of course, this applies only to lifting strategies that were sophisticated to begin with, which is why the improved framework is mainly useful in conjunction with the predecessor and maximum measure propagation lifting strategies.

4.3.2 Optimization after lifting to top

The implementation of `lift` follows the description given in section 4.1, but with one important enhancement: whenever a vertex with odd priority p is lifted to $\top$, the value of M_p is decreased by 1. As a result, during solving, M is effectively determined by the cardinalities of only those vertices with progress measures less than $\top$. Note that this may cause some progress measure vectors to temporarily exceed the new vector bounds M', but this will be resolved when the vertex is lifted, as it necessarily must (see below).

This enhancement is not described by Jurdziński and not implemented in any of the other solvers based on Small Progress Measures, to my knowledge; all of these assume M to be constant during solving. It was invented specifically to remedy cases where the dual solving approach (described in 4.3.4) would perform worse compared to the approach of solving the normal and dual game in succession. These cases arise because in the latter approach vertices won by Even (in the normal game) are removed from the dual game before solving, which reduces the complexity of the second problem. With the two-sided approach no vertices are ever removed, which could cause the dual game to have significantly higher bounds on the progress measure codomain M.

Besides eliminating this disadvantage in the dual-solving approach, the enhancement also benefits the regular algorithm.

4.3.2.1 Correctness

In Small Progress Measures, non-$\top$ vectors encode an optimal strategy for Even; informally, the elements of the vectors encode how many vertices with dominant odd priorities can be visited consecutively in a winning play before encountering a vertex with a dominant even priority. Since vertices which are assigned $\top$ value (i.e. which are won by Odd) cannot be part of a winning play for Even, they will be excluded from this count, and therefore the least fixed point will necessarily have all vectors within the reduced bounds.

To fit this optimization into Jurdziński's framework, we should introduce a smaller set of progress measures $M'(\rho)$ the bounds of which are dependent on the count of non-$\top$ vertices in the

partially-constructed progress measure ρ:

$$M'_i(\rho) = \begin{cases} \{0\} & \text{if } i \equiv 0 \mod 2 \\ \{0..\left|\phi^{-1}(i) \setminus \rho^{-1}(\top)\right|\} & \text{if } i \equiv 1 \mod 2 \end{cases}$$

Clearly, $M'(\rho) \subseteq M$. Then, $\mathit{Prog}'(\rho, v, w)$ can be defined like $\mathit{Prog}(\rho, v, w)$ but drawing from $M'(\rho)$ instead of M, and similarly $\mathit{Lift}'(\rho, v)$ can be defined in terms of $\mathit{Prog}'(\rho, v, w)$ instead of $\mathit{Prog}(\rho, v, w)$. At this point, Jurdziński's theorems apply equally to $\mathit{Prog}'(\rho, v, w)$. In particular, the monotonicity of the lifting function is preserved and thus fixed point iteration can still be used to find a minimal fixed point of $\mathit{Lift}'(\rho, v)$ that yields an optimal solution for Even.

A more intuitive way to see that decreasing the progress measure bounds is permissible, even without fixing any vectors that are now out of range, is to consider that if vertex $\rho(v) = \top$ for some vertex v, then $v \in W_1$ and $\mathit{Attr}^1(\{v\}) \subseteq W_1$ and we could solve the subgame induced by $V \setminus \mathit{Attr}^1(\{v\})$ with associated bounds M'' instead. However, since ${M''}_p \subseteq M'_p(\rho)$ and since we compute a minimum fixed point in both cases, the final progress measures vectors must be the same for corresponding vertices in both games. This guarantees that the final value of all vectors will lie in ${M''}^{\top}$. (Note that attractor set computation achieves the same effect as propagating $\top$ values as usual.)

4.3.3 Game and graph preprocessing

Instead of improving SPM itself, there are a few transformations that can be applied to the parity game before passing it to the solver. These preprocessing operations are intended to be relatively cheap compared to solving the game itself (at least in the worst case). All practical tools implement at least some of these.

4.3.3.1 Loop removal

One peculiarity of Small Progress Measures is that it may spend a lot of time lifting vertices with high priority values which lie on a cycle. A typical example of this occurs when an Odd-controlled vertex has a high, odd priority and a loop (i.e. an edge from the vertex to itself). The final progress measure for such a vertex will obviously be $\top$ (since Odd will keep the token at this vertex indefinitely) but the progress measure at this vertex will change only in small increments, requiring many lifts to reach its final value.

Something similar occurs for vertices of odd priority which are controlled by Even. Although these cannot be initialized to $\top$ (they may still be won by Even) we can discard the loop edge to speed up propagation of values from successor vertices through this vertex. Four cases in total can be distinguished, as summarized in Table 4.1:

Player	Priority	Winner	Value	Loop edge	Other edges
Even	even	Even	0	kept	removed
Even	odd	undecided	0	removed	kept
Odd	even	undecided	0	removed	kept
Odd	odd	Odd	$\top$	kept	removed

Table 4.1: Preprocessing of vertices with loops

(Note that when a vertex has only one outgoing edge, then it can be considered to belong to either player; in this case it's most beneficial to assign it to the player corresponding to the parity of its priority.) There are different ways to use this information; the simplest is to remove the useless edges and initialize the progress measure vector to $\top$ for the fourth case, and then solve the rest of the game as usual.

It should be noted that in two of the four cases, it is known to which winning set the vertex belongs, and the associated strategy is to use the loop back to the vertex. A simple extension is

then to collect sets of these vertices for each player, and extend these to their attractor sets, to obtain a larger subset of vertices that are completely solved and can be removed from the game entirely. The resulting game graph then has no more loops.

4.3.3.2 Winner-controlled cycle removal

In his solver, Freark van der Berg introduced a more sophisticated version of the loop removal process described above. He searches for cycles of vertices controlled by Odd with odd priority, using a depth-first-search limited to a relatively shallow depth (8, in his experiments). This results in detection of small cycles only but despite this limitation, his benchmark results show that this already provides significant benefits over the removal of loops only; at least in large unclustered random games, where small cycles are common.

This idea can be improved upon in several ways:

1. Search for cycles of any size (not just small ones).
2. Don't limit the search to fixed parity vertices: only the parity of the least priority in a cycle matters; if that matches the controlling player, the priorities for the other vertices are irrelevant.
3. Search for cycles controlled by Even as well as Odd.

The last generalization does not reduce the progress measure bounds, but it is still useful because it is a quick way to solve part of the winning set of player Even.

The general process will be called *winner-controlled cycle removal*; i.e. removal of all cycles which are entirely controlled by the player corresponding to the dominant priority. To find these cycles, depth-first searching through all vertices in the graph would be too slow, but fortunately there is a more efficient solution, which is similar to the approach used to verify strategies.

To detect all i-cycles controlled by player p (where $p \equiv i \mod 2$) in the graph, construct the subgraph (V^i, E^i) induced by the vertices controlled by p with priority greater than or equal to i, i.e. $V^i = \{v \in V : \phi(v) \geq i \wedge (v \in V_p \vee |vE| = 1)\}$ and $E^i = E \cap \left(V^i \times V^i\right)$. Note that vertices controlled by $1 - p$ with outdegree 1 are considered controlled by p for this purpose and that the resulting graph may contain vertices without any edges; it need not be a proper parity game graph.

To find i-cycles, decompose the graph (V^i, E^i) into strongly connected components. Vertices with priority i that occur in a connected component with at least one edge (i.e. either the component contains at least two vertices or a single vertex with a loop) are part of a cycle with dominant priority i. If we collect one such vertex per component (if one exists) and then compute the attractor set for these vertices, we have identified all vertices which either lie on p-controlled i-cycles or in their attractor set.

Strategies can be constructed as follows. For each vertex in the initial vertex set, pick an arbitrary successor *in the same component*. The rest of the strategy is generated as part of attractor set computation. Finally, we can extend the winning regions identified in the subgame to the attractor set in the global game. All vertices identified this way are solved and may be removed from the game.

It may appear redundant to compute attractor sets twice, but it is necessary to construct correct strategies. The first computation is necessary to construct the i-cycles that are winning for player p, creating a valid strategy for all solved vertices, while the second calculation extends the solution as far as possible, which is necessary to guarantee the remaining subgame is a proper game.

To find winning cycles for all priorities, the process outlined above is simply repeated for each priority in the game. Every iteration requires constructing a subgraph with the appropriate vertices, decomposing it into strongly connected components, computing the attractor set in that subgraph, again in the original game, and finally computing a subgame with the remaining vertices. Using appropriate data structures, each of these operations takes at most time linear in the number of the edges in the graph, for a total runtime complexity of $O(dE)$.

Algorithm 4.6 Winner-controlled cycle removal

```
1  winner_controlled_cycles(ParityGame game, Strategy strategy)
2  {
3      Set<Vertex> solved
4      Priority p = 0
5      while p < game.d {
6          ParityGame subgame = game.make_subgame({ v in game.graph.V
7              where (v not in solved) and (game.priority(v) >= p) and
8                    ( game.player(v) == p mod 2 or
9                      game.graph.outdegree(v) == 1 ) })
10         Set<Vertex> won = {}
11         for scc : decompose(subgame.graph) {
12             for (v,w) : scc.edges() {
13                 if subgame.priority(v) == p {
14                     if game.player(v) == p mod 2 {
15                         strategy[v] = w
16                     } else {
17                         strategy[v] = NO_VERTEX
18                     }
19                     won.insert(v)
20                     break
21                 }
22             }
23         }
24         make_attractor_set(subgame, p mod 2, won, strategy)
25         solved.add(won)
26         make_attractor_set(game, p mod 2, solved, strategy)
27         p = p + 1
28     }
29     return solved
30 }
```

This is considerably more than the $O(E)$ time required for winner-controlled loop removal only, but the potential benefits are much larger too, making this preprocessing step useful in practice. Although cycle removal makes loop removal obsolete (since loops are simply very short cycles) it may still be beneficial to run the loop-removal routine first to remove winner-controlled loops (and vertices in their attractor sets) more quickly than the more general procedure could.

Due to the similarity to solving single player games, it seems likely that the approach described in [18] can also be adapted to perform winner-controlled cycle removal more efficiently, but due to time constraints, this was not further explored.

4.3.3.3 Priority compression

For Small Progress Measures it is desirable that the maximum priority used in the game is as low as possible, since the maximum priority influences both the worst-case runtime and the length of the progress measure vectors. Often the priorities used in a game are numbers from 0 to d (the index of the game, exclusive). However, when some priority values are unused, the range of priorities used can be reduced by a process called *priority compression.*

To compress priorities in a game $\Gamma = (V_0, V_1, E, \phi)$, we find the first unused priority (say, p) in the game. If $p = d$ then all priorities from 0 through $d - 1$ (inclusive) are used and we are done. Otherwise, let q be the least priority greater than p that is used.

If $p = 0$ then we can decrement all priorities by q and, if q is odd, we additionally switch the roles of players Even and Odd. Formally, we construct a game $\Gamma' = (V_0', V_1', E, \phi')$ where $V_x' = V_y$ and $x = y - p \mod 2$, $\phi'(v) = \phi(v) - q$.

If $p > 0$ then we keep $V_0' = V_0$ and $V_1' = V_1$. However, if $p \equiv q \mod 2$ then we get rid of the unused priorities and lump the vertices with priority q in with those with priority p. After all, p and q have the same parity, and since none of the values in between are used, they can be interchanged without affecting the outcome of the game. More formally, in this case we define $\phi'(v)$ as:

$$\phi'(v) = \begin{cases} \phi(v) & \text{if } \phi(v) \leq p \\ \phi(v) + p - q & \text{if } \phi(v) > p \end{cases}$$

Since $p - q \equiv 0 \mod 2$ this preserves parity: $\phi'(v) \equiv \phi(v) \mod 2$. Finally, if $p > 0$ and $p \not\equiv q \mod 2$ then we can remove the unused priorities, but we do not want to equate p and q which have different priorities:

$$\phi'(v) = \begin{cases} \phi(v) & \text{if } \phi(v) \leq p \\ \phi(v) + p - q + 1 & \text{if } \phi(v) > p \end{cases}$$

Again, since $p - q + 1 \equiv 0 \mod 2$ this preserves the parity of ϕ while removing the unused priorities between p and q. Repeating this process to eliminate all the gaps results in a game which uses priorities from 0 to $d - 1$ (inclusive). It is not too difficult to perform the entire operation in $O(V)$ time.

Further compression can be achieved by applying this operation on each strongly-connected component of the game graph individually. This is permissible because every infinite play must necessarily stay in a single component eventually. However, it is more common to apply priority compression after decomposition into strongly connected components, which is independently useful but achieves the same effect with regards to priority compression.

4.3.3.4 Priority propagation

Finally, a simple preprocessing step that can be applied to speed up solving with Small Progress Measures (and possibly other algorithms as well), is a process called *priority propagation* (a term suggested in [12] though the concept already appears in [1]): we can replace the priority assigned to a vertex with the greatest priority among its successors, if the latter is less than the former, without affecting winning sets or optimal strategies in the game, and similarly for the predecessors of a vertex. (Friedmann & Lange distinguish the two cases by calling them *backwards* and *forwards propagation* respectively, though there seems to be little reason to apply one process but not the other.)

This can be seen to be correct by considering that in a finite graph, if a play passes through a vertex infinitely often, it must also pass through at least one of its successors infinitely often. When a vertex's priority is greater than all of its successors', then it cannot be the dominant priority of the play, so it can be safely decreased. The same argument applies to predecessors, of course.

The rationale for applying this transformation before running SPM is that it tends to shift the distribution of priorities towards lower values (since lower values can replace higher values, but not the other way around), which could reduce the number of different possible small progress measure values (i.e. the size of the set M) and make vector comparisons/updates slightly faster, since the prefixes operated on are shorter on average.

In practice the performance gain derived from priority propagation appears to be small, as has also been noted by Keiren in [28]. Friedmann & Lange report that *"empirically, the use of priority propagation turns out to be harmful"*.

4.3.3.5 Vertex reordering

The vertices of a graph are not intrinsically ordered. However, algorithms that iterate over vertices one by one (such as the linear lifting strategy) must assign some order to the vertices, at least implicitly.

With Small Progress Measures, information is propagated backwards against the direction of edges: when a vertex is lifted, this may cause some of its predecessors to become unstable and suitable for lifting too. As a result, the linear lifting strategy is more efficient when vertices are ordered such that predecessors are lifted after their successors. In order to make such algorithms more efficient, my tool supports several options to reorder vertices before running the solving algorithm:

1. Re-order vertices by depth-first search order.
2. Re-order vertices by breadth-first search order.
3. Reverse the order of vertices.
4. Shuffle vertices randomly.

Reversing the order of vertices is useful when the graph has been constructed in a forward fashion, which occurs for example when exploring a state space explicitly. These reordering operations can also be combined; for example, it may make sense to use reverse search order to maximize the ratio of backward to forward edges.

Although vertex reordering makes little difference in practice, it is a cheap preprocessing pass that is useful to have as an option.

4.3.4 Two-sided SPM

A noteworthy variant of the Small Progress Measures algorithm was implemented by Oliver Friedmann in PGSolver, although it has not been described in literature. In this case, the algorithm is not run for one player at a time, but for both players at once. In Friedmann's implementation this means that a single vertex is selected, and it is lifted both in the normal game and its dual, and if either attempt succeeds, the predecessor vertices are queued. Effectively, the queue is shared between the two games. Conceivably, the two games could be processed independently, even in parallel on separate machines.

Periodically (in the current implementation after every $|V|$ lifting attempts) information about winning vertices is exchanged between the games, by conservatively identifying vertices in one game that are won by the player for which we are computing the strategy, and setting the progress measures of these vertices to $\top$ in the corresponding dual game for his opponent. These vertices are identified as follows.

Without loss of generality, assume we are computing the strategy for player Even. Then, we add a mark to every vertex for which any of the following conditions apply:

1. Its progress measure is $\top$
2. The vertex is controlled by Even and either:
   (a) all of its successors are marked; or
   (b) its progress measure is less than (or equal to, if the vertex priority is odd) all of its unmarked successors
3. The vertex is controlled by Odd and either:
   (a) one of its successors is marked; or
   (b) its progress measure is less than (or equal to, if the vertex priority is odd) one of its successors

Here, "less" means less with respect to the priority of the current vertex. Note that these conditions are consistent with the usual criteria for lifting vertices, where the marked vertices are considered to be set to $\top$. The interpretation of the marks is that marked vertices may be won by Odd, and

unmarked vertices are known to be won by Even. The unmarked vertices therefore are a subset of the winning set of Even, and the corresponding vertices in the dual game being solved for Odd can be set $\top$. The same process can, of course, be applied for Odd in the dual game to get vertices which should be set to $\top$ in Even's game.

To compute a maximal set of marked vertices, a similar approach as in the main SPM algorithm is used: all vertices are queued, checked against the properties mentioned above, and whenever a mark is added to a vertex, its predecessors must be re-examined, so they are queued again. The difference with the main SPM algorithm is that a mark can be added only once, which bounds the total processing time to the number of edges in the graph.

Propagating this information every $|V|$ lifting attempts amortizes the overhead involved against the work done lifting vertices, so that the worst-case complexity remains unchanged.

There is some similarity with Schewe's "big steps" approach [35] to solving parity games, which also tries to find more-easily-identifiable subsets of winning regions first in order to reduce the remainder of the problem to simpler terms. Both approaches provide a way to avoid "getting stuck" repeatedly lifting vertices by very small increments. In Schewe's scheme this is achieved by using reduced progress measure bounds to find small winning regions, obtaining only a partial solution using Small Progress Measures, and applying the recursive algorithm (see chapter 5) to solve the remainder of the game.

The two-sided approach solves games entirely with Small Progress Measures and is therefore simpler than Schewe's, but doesn't improve performance in the worst case. (It should be noted however, that in PGSolver the two approaches are combined: the "big steps" solver uses the SPM implementation with the two-sided optimization.)

4.3.4.1 Advantages and disadvantages

Jurdziński's solution approach is asymmetric: it solves the game only from the perspective of player Even. To find a strategy for Odd, we can re-run the algorithm on the dual game for the opponent's winning set, which may be somewhat easier because the game graph is smaller. Because of this asymmetry, it may be quicker to start from the dual game in the first place (effectively solving for Odd in the original game) but we don't know in advance whether this is true. An advantage of Friedmann's approach is that we do not need to guess, since we work on both the game and its dual at once. The extra amount of work done in this case is at most equal to the amount of work done for the easiest game.

The second advantage of Friedmann's two-sided approach is that it can propagate information between games even when neither is completely solved. This is useful because even in cases where both sides are equally difficult, they each may contain small regions that can be solved relatively easily and then be used to simplify the other game.

There may be rare cases where neither of these benefits apply. In such cases, up to twice as much effort may be spent on lifting vertices, and some additional overhead is incurred by searching for stable regions in the games. Since the cost of this search can be amortized against the time spent on lifting, the worst-case slowdown can be brought arbitrarily close to a factor of 2.

4.3.4.2 Implementation differences

The observation that the two-sided approach should use (at worst) about twice as much time and space as the standard algorithm may appear obvious, but it actually relies on a suitable implementation of the algorithm. Perhaps surprisingly, PGSolver can take orders of magnitude longer to solve games with the two-sided approach than without it. The cause of this is that PGSolver omits construction of subgames entirely, even after one of the sides is completely solved, and consequently does not benefit from a reduction in progress measure bounds that occurs when constructing a subgame for the harder part of the game. This problem is rectified by reducing progress measure vector bounds whenever vertices are lifted to $\top$ (rather than calculating these bounds once, before starting solving) which is the optimization described in subsection 4.3.2.

Another difference between PGSolver and my implementation is that PGSolver uses a single lifting strategy for both the normal and dual game, and when lifting vertices, recomputes progress measures for both games at once. Although that works, it seems plausible that the sets of liftable vertices in the normal and dual game will diverge, so it seems more reasonable to instantiate separate lifting strategies for the two games in order to do more useful work.

Finally, in an attempt to propagate information about losing vertices earlier, my solver attempts $|V|$ lifts in one game, before propagating information and switching to the dual game (rather than performing $2\,|V|$ lifting attempts in total before propagating information in both directions).

Of these three differences, the first is crucial for optimal performance, while the other two are at best small improvements over Friedmann's implementation.

Chapter 5

Zielonka's recursive algorithm

An entirely different approach to solving parity games arises from a constructive proof of the existence of memoryless strategies by Wiesław Zielonka in [41]. It is presented solely as a proof, not as an efficient algorithm, but it has been shown (e.g. in [12] and in [28]) to work well as a solving algorithm too.

Zielonka's formulation of the algorithm is based on previous work by McNaughton [33]. The main difference is that McNaughton limits himself to finite graphs, while Zielonka concerns himself with infinite graphs as well. Because of their fundamental similarity, Zielonka's algorithm is sometimes referred to as McNaughton's algorithm in literature on parity games. These algorithms are functionally identical.

For finite graphs, there is a constructive proof by induction on the number of different priorities and the size of the game graph. When there is only one priority in use, the game is trivially solved, and in particular this occurs when the graph consists of a single vertex (this is the basis for the induction). We can then solve any game by reducing either the size of the graph or decreasing the number of different priorities, as follows.

Suppose without loss of generality (after applying priority compression if necessary) we start with a game Γ with $d > 1$ and some vertices with priority 0. We can compute the attractor set U of these vertices for player Even ($U = Attr^0(\Gamma, \phi^{-1}(0))$, take its complement V' ($V' = V \setminus U$) and call the induced subgame Γ' ($\Gamma' = \Gamma|V'$). Since there are no vertices with priority 0 left in Γ', it has at most $d-1$ different priorities and we can invoke the induction hypothesis to solve Γ', yielding winning sets W'_0 and W'_1 and associated strategies.

When $W'_1 = \emptyset$ the entire game is won by Even, which follows from the fact that the token must either be in U infinitely often, where Even can keep attracting it to a 0-priority vertex, or it

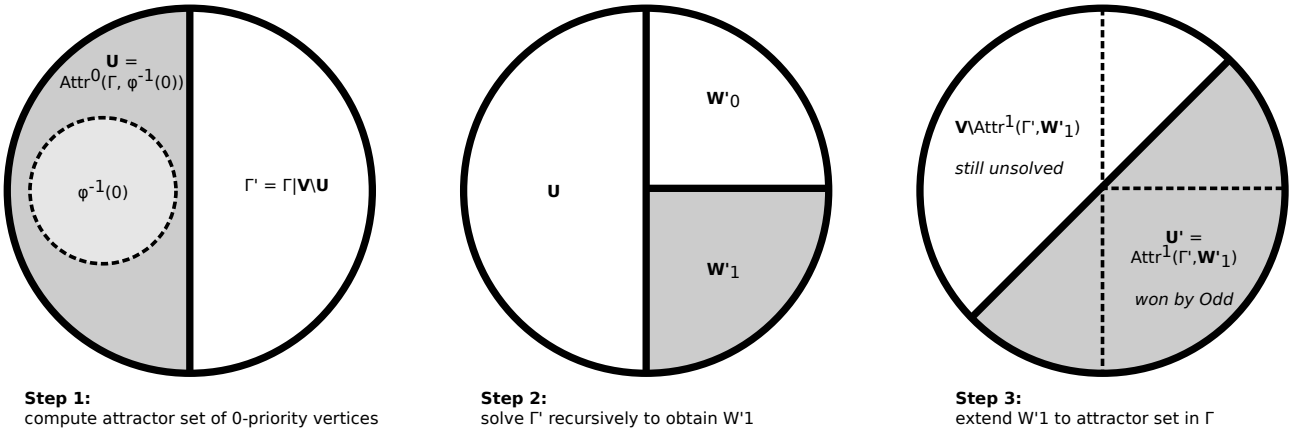


Figure 5.1: One iteration of Zielonka's algorithm

must stay in W_0' indefinitely, which is by definition winning for Even. However, when $W_1' \neq \emptyset$ then Odd can possible escape from W_0' to W_1' via U. We know that $W_1' \subseteq W_1$ because V' is a trap for Even in Γ and therefore any winning strategy for Odd in Γ' is winning in Γ too. We can calculate $U' = Attr^1(\Gamma, W_1')$ giving $W_1' \subseteq U' \subseteq W_1$ and $V \setminus U'$ is a trap for Odd in Γ (as the complement of an Odd-attractor). That means Odd has a winning strategy in U' and it suffices to solve the strictly smaller subgame $\Gamma|(V \setminus U')$ for which we again invoke the induction hypothesis.

Figure 5.1 shows the steps taken in an iteration of Zielonka's algorithm. In a game with d priorities and $|V|$ vertices, we can reduce the game by one vertex less than $|V|$ times, each time solving a subgame Γ' with a strictly smaller d, giving a worst-case runtime of $O(V^d)$. In [11], Friedmann presents a class of games requiring exponential time to solve, with $\Omega(fib(V))$ as a lower bound (where $fib(i)$ returns the i-th Fibonacci number).

5.1 Implementation

The proof structure above can be translated into a procedure that recursively solves games with decreasing number of priorities, while iteratively constructing smaller subgames. This approach is preferable to a completely recursive procedure, because it bounds the maximum recursion depth to d and thereby avoids the risk of overflowing the runtime stack.

Rather than applying priority compression between recursive invocations, we consider the least inversion of parity in the game, where an inversion of parity is a priority p such that $\{v \mid \phi(v) < p\} \neq \emptyset$ and $\{v \mid \phi(v) < p \ \wedge \ \phi(v) \equiv p \mod 2\} = \emptyset$. If there are no inversions of parity, then all priorities assigned to vertices have the same parity, and the corresponding player trivially wins the entire game. Otherwise, all vertices with priority less than p can be implicitly compressed into a single value; in the implementation this is done by adding all vertices with priority less than p to a minimum-priority attractor set. This has the same effect as applying priority compression on each subgame.

The recursive algorithm is presented in Algorithm 5.1. Recall that `strategy` maps vertices to successor vertex indices (where the value `NO_VERTEX` is used to indicate that a vertex is not in the controller's winning set). In the implementation this is a simple array of vertex indices, which requires that subgame construction preserves vertex indices.

The execution of the main algorithm depends on several, simpler helper functions:

- `least_inversion` returns the least inversion in the game, as described above, by analysing the cardinality of priorities in use. If there is no inversion, the game index `d` is returned instead.
- `make_attractor_set` takes a vertex set and extends it to its attractor set with respect to the given player, as described earlier. For all vertices that are added to the set (but not those in initial set) the corresponding strategy is also assigned, so that play according to the strategy leads to vertices in the initial set.
- `get_complement` returns the complement of a given vertex set, with respect to the vertex set of a graph.
- `make_subgame` constructs the subgame induced by the given vertex set. Since in Zielonka's algorithm subgames are constructed from the complement of an attractor set, this always results in a proper game. (For purposes of the algorithm outlined above, it is assumed that vertex identifiers are preserved, so that assignments to the global strategy array are consistent with the global game.)

From a high-level point of view, the algorithm operates by first assuming that all vertices in `min_prio_attr` are won by the corresponding player and then recursively solving the rest of the game. If the rest of the game is won by the same player, the procedure is complete; otherwise, the opponent has a non-empty winning set, and can win from any vertices in its attractor set as well. This attractor set is removed from the game, and the procedure is repeated with the remainder, until it becomes empty or entirely won by one player.

Algorithm 5.1 Zielonka's Recursive Algorithm

```
1  solve_recursively(ParityGame game, Strategy strategy)
2  {
3      Priority p = least_inversion(game)
4      while p < game.d {
5          Set<Vertex> min_prio = { v in game.graph.V where game.priority(v) < p }
6          Set<Vertex> min_prio_attr = make_attractor_set(game, (p-1) mod 2, min_prio, strategy)
7          Set<Vertex> unsolved = game.graph.get_complement(min_prio_attr)
8          if |unsolved| == 0 { break }
9          ParityGame subgame = game.make_subgame(unsolved)
10         solve_recursively(subgame, strategy)
11         Set<Vertex> lost = { v in subgame.graph.V where strategy.winner(v) == p mod 2 }
12         if |lost| == 0 { break }
13         Set<Vertex> lost_attr = make_attractor_set(game, p mod 2, lost, strategy)
14         Set<Vertex> remaining = game.graph.get_complement(lost_attr)
15         game = game.make_subgame(remaining)
16         p = least_inversion(game)
17     }
18     for v in game.graph.V {
19         if game.priority(v) < p {
20             if game.player(v) == game.priority(v) mod 2 {
21                 strategy[v] = any successor of v
22             } else {
23                 strategy[v] = NO_VERTEX
24             }
25         }
26     }
27 }
```

5.1.1 Strategy computation

The strategy for vertices in `min_prio_attr` \ `min_prio` is assigned during attractor set computation. The strategy for vertices in its complement are assigned by the recursive invocation of `solve`. The strategy for vertices in the attractor set of vertices lost to the opponent are overwritten by the next attractor set computation; this part of the strategy is then final, and the vertices in this set are removed from the graph.

The while-loop exits in three cases: when the game has no more inversions, when the attractor set of the minimum priority vertices covers the entire game graph, or when the vertices outside this attractor set are all won by the player corresponding to the minimum priority vertices. In all of these cases, the remaining game has no inversions (i.e. it is a single-parity game). At this point, the only vertices for which the strategy has not been assigned, are those with priority less than `p`. For those, it suffices to pick an arbitrary successor in the remaining part of the game (which is done in the for-loop at the end) to complete the strategy.

5.1.2 Termination

In this algorithm every recursive invocation removes at least one inversion from the game (when `solve` is called recursively, all vertices with priority less than `p` are removed, therefore `p` is not an inversion in the subgame) so the number of inversions in the initial game is an upper bound on the recursion depth. Further more, every complete iteration of the while loop removes at least two vertices from the game (one in `lost` and another in its attractor set, overlapping `min_prio_attr`) until the game is empty or completely solved, so the maximum number of iterations in the while loop is $\frac{|V|}{2}$. Combined, these observations guarantee the algorithm terminates.

Chapter 6

Empirical evaluation

An empirical evaluation of the performance of the algorithms described so far is necessary to determine how well they work in practice. However, meaningful benchmark results can only be obtained when these algorithms are applied to reasonably complex test data. Games which are too easily solved by preprocessing techniques or other, simpler methods should be excluded from the test set.

We will use two main sources of games: clustered random games, and some of the games used earlier by Friedmann & Lange (described in detail in [12]). The rationale is that clustered random games are an unlimited source of reasonably-hard games (when generated properly) while including some of the more structured games from Friedmann & Lange simplifies comparison of the results presented here with earlier reports.

6.1 Random games

Random games are characterized by the size of the vertex set $|V|$, the average outdegree o, (or, equivalently, average indegree), and the number of priorities d. For each vertex, the outdegree is chosen uniformly at random between 1 and $2o - 1$ (to guarantee each vertex has at least one outgoing edge) and then adjacent vertices are chosen uniformly at random from the vertex set without duplicates and excluding the vertex itself (to exclude the possibility of loops). Finally, a priority for the vertex is chosen uniformly at random between 0 and $d - 1$ (inclusive).

Graphs generated in this way typically consist of one large, strongly-connected component, and several small (usually single-vertex) components. For very small graphs there is even a risk of disconnecting the game graph entirely! To remedy this, after graph generation strongly-connected components are identified and those components that have either no incoming or no outgoing edges (i.e. they lie at the top or bottom of the hierarchy of strongly-connected components) are connected to each other in a cycle in random order. This guarantees the output consists of a single, strongly-connected component (at the cost of introducing some extra edges).

6.1.1 Clustered random games

To create games with a little less uniform graph structure (which more closely resemble instances arising in practice, as well as being less likely to be solved completely by the cycle-removal preprocessor) it is useful to structure the game graph around clusters of some fixed size. To generate a game with $|V|$ vertices and clusters of size C, first $\left\lfloor \frac{|V|}{C} \right\rfloor$ clusters of size C are randomly-generated using the procedure described in the previous subsection. Then, clusters are grouped together, at most C at a time, and connected together according to a randomly-generated game with the same parameters.

For example, if $|V| = 50$ and $C = 10$, then 5 clusters of 10 vertices each are generated, as well as one 5-vertex top-level game that is used to connect these clusters: if there is an edge from, say,

vertex 1 to 2 in the top-level game, then an edge is added from a random vertex in the first cluster to a random vertex in the second cluster. If $|V| > C^2$ then a third level is needed to combine groups of clusters, and so on, for a hierarchy of $\lceil {}^{C}\log |V| \rceil$ levels in the final graph.

Since all games generated were strongly-connected, the combined graph will be strongly-connected as well. However, it does contain a few more edges, limited to a factor $\frac{C}{C-1}$ (as the sum of a geometric series). Consequently, if few extra edges are desired, the cluster size should not be chosen too small.

6.2 Cases from *Solving Parity Games in Practice*

Friedmann & Lange's benchmarks include four kinds of games:

1. *Decision procedures:* two subclasses of games constructed from the validity problem of recursively defined μ-calculus formulae (ϕ and ϕ').
2. *Elevator verification:* two model checking problems, consisting of checking the fairness of two similar elevator models (one fair, G; one unfair, G').
3. *Towers of Hanoi*: a simple model checking problem which verifies whether a position in the solitary Towers of Hanoi game is solvable.
4. *Random games*: large games generated randomly.

Not all of these are useful for benchmarking. The third class of games can be solved entirely using either decomposition into strongly connected components (as Friedmann and Lange also reported) or by owner-controlled cycle removal, and is therefore excluded from the test set.

The fourth class of games could be interesting, but the exact parameters used to generate these random games (especially how they are clustered) are not provided. It should be noted that unclustered random games can typically be solved completely using cycle removal (which is a preprocessing operation that was not available to Friedmann and Lange). Instead, we will use clustered random games as described in the Subsection 6.1.1. One notable difference with the random games used by Friedmann and Lange is that our games will consist of a single strongly-connected component only.

This leaves the first two classes which each include two kinds of games that can in turn be varied in size. The decision procedure games are generated from recursively-defined μ-calculus formulae using a separate tool called MLSolver [13]. The elevator verification games are generated with a tool from the PGSolver suite. To prevent implementation details of these tools influencing the results, the vertices of the game graphs are shuffled before each experiment. This ensures the graph structure is retained while removing the effect that the particular vertex ordering produced by these tools might have had on the performance of various lifting strategies.

The properties of the games (such as size of the game graph, and number of different priorities) is summarized in Table 6.1. A more detailed description of how these games are structured is available in [12].

6.2.1 Limitations

It should be noted that these cases alone do not form a very comprehensive benchmark set. The main limitation is that the first three classes (ϕ, ϕ' and G) do not require lifting any vertices to $\top$ to solve the games; instead Odd's winning region can be extracted using loop removal with attractor set computation. For the fourth class of games (G') this applies to Even's winning set instead. This means that these games may be relatively easy to solve, at least using Small Progress Measures.

As a consequence, improvements like the two-sided approach and decrementing progress measure bounds after lifting to $\top$ are expected to provide little benefit on these cases. These improvements can be better evaluated on the more demanding (clustered) randomly-generated cases.

Game	Vertices	Edges	Priorities	Largest component
ϕ_2	435	584	9	131
ϕ_3	2,677	3,530	11	368
ϕ_4	13,815	18,330	13	1,068
ϕ_5	60,674	79,591	15	2,658
ϕ_6	259,170	340,982	17	6,515
ϕ_7	1,011,979	1,316,640	19	15,991
ϕ_8	3,939,223	5,128,098	21	36,606
ϕ'_{10}	4,590	6,520	11	1,812
ϕ'_{50}	21,910	31,320	11	8,652
ϕ'_{100}	43,560	62,320	11	17,202
ϕ'_{500}	216,760	310,320	11	85,602
ϕ'_{1000}	433,260	620,320	11	171,102
ϕ'_{2000}	866,260	1,240,320	11	342,102
ϕ'_{5000}	2,165,260	3,100,320	11	855,102
G_5	15,684	26,354	3	10,973
G'_5	16,356	38,194	3	11,456
G_6	108,336	180,898	3	78,818
G'_6	111,456	287,964	3	81,121
G_7	861,780	1,431,610	3	643,756
G'_7	876,780	2,484,252	3	655,073
G_8	7,744,224	12,810,736	3	5,894,533
G'_8	7,814,016	24,093,264	3	5,948,026

Table 6.1: Properties of test cases from *Solving Parity Games in Practice*

6.3 Benchmark platform

All benchmarks were performed on a Linux system with Intel Xeon E5520 processors (8 MB cache, 2.26 GHz clock speed) and 24 GB of memory (far more than needed for any test case).

6.4 Results on random games

Tables 6.2 and 6.3 show how the different lifting strategies perform on unclustered and clustered random graphs. The configurations differ on four (not completely independent) axes:

1. *Lifting strategy*: linear lifting ("lin"), predecessor lifting ("pred"), maximum measure propagation ("max"), minimum measure propagation ("min"), and maximum step propagation ("step").

2. *Direction/extraction order:* the linear lifting strategy can either iterate in forward direction (F) or alternate direction on each pass (A). The predecessor lifting strategy can either use queue-order (Q) or stack-order (S), while the maximum/minimum measure and maximum step strategies can also use the internal heap order (H). These variations are described in Subsection 4.2.

3. *Framework*: either the traditional framework or the improved framework (I) described in 4.3.1. The linear lifting strategy only supports the traditional framework, while the predecessor lifting strategy is the only one that supports both.

4. *Two-sided approach:* all strategies can use the two-sided solving approach (T) described in Subsection 4.3.4.

Five different classes of random games were used, each with 10 priorities and outdegree 3, but with varying graph sizes and cluster sizes. In the results, `4000/16` denotes a graph generated as

		4000/0		8000/0		16000/0		4000/16		4000/64	
		#solved	avg. lifts	#solved	avg. lifts	#solved	avg. lifts	#solved	avg. lifts	#solved	avg. lifts
Linear	F	27	5.290	25	7.997	25	28.621	0	0.000	0	0.000
Linear	A	27	7.702	25	11.710	25	37.968	0	0.000	0	0.000
Predecessor	Q	30	2.033	29	5.932	28	18.813	0	0.000	4	472.543
Predecessor	S	29	1.451	28	0.699	22	2.425	0	0.000	0	0.000
Predecessor	Q I	30	0.891	30	3.114	29	9.669	0	0.000	4	179.287
Predecessor	S I	29	0.253	29	0.287	25	0.718	0	0.000	0	0.000
Max. Measure	Q I	29	0.128	29	0.079	27	0.236	0	0.000	1	533.683
Max. Measure	S I	30	0.087	30	0.133	29	0.251	0	0.000	0	0.000
Max. Measure	H I	30	0.254	30	0.088	26	0.242	0	0.000	1	782.083
Max. Step	S I	30	0.195	30	0.230	28	0.343	0	0.000	1	449.607
Max. Step	Q I	30	0.189	30	0.220	28	0.303	0	0.000	1	407.224
Max. Step	H I	30	0.187	30	0.226	28	0.294	0	0.000	1	464.070
Min. Measure	Q I	25	36.135	24	65.144	9	109.908	0	0.000	0	0.000
Min. Measure	S I	26	40.427	24	61.187	8	143.396	0	0.000	0	0.000
Min. Measure	H I	26	42.547	23	57.114	9	153.542	0	0.000	0	0.000
Linear	F T	29	0.679	27	0.352	26	1.242	1	802.259	5	272.809
Linear	A T	29	0.961	28	0.482	26	1.260	1	854.012	5	326.424
Predecessor	Q T	30	0.049	29	0.055	30	0.188	17	105.114	21	22.095
Predecessor	S T	30	0.107	30	0.158	28	0.306	0	0.000	5	503.767
Predecessor	Q I T	30	0.030	30	0.031	30	0.078	18	46.409	21	9.023
Predecessor	S I T	30	0.049	30	0.067	28	0.224	0	0.000	10	304.465
Max. Measure	Q I T	30	0.033	30	0.049	30	0.114	0	0.000	3	383.092
Max. Measure	S I T	30	0.032	30	0.044	30	0.103	0	0.000	1	126.888
Max. Measure	H I T	30	0.032	30	0.051	30	0.127	0	0.000	6	347.106
Max. Step	S I T	30	0.047	30	0.063	30	0.135	0	0.000	9	296.409
Max. Step	Q I T	30	0.048	30	0.065	30	0.138	0	0.000	6	274.866
Max. Step	H I T	30	0.046	30	0.063	30	0.135	0	0.000	8	268.926
Min. Measure	Q I T	30	0.043	30	0.043	30	0.124	16	133.194	20	38.009
Min. Measure	S I T	30	0.041	30	0.043	30	0.135	14	121.260	18	37.710
Min. Measure	H I T	30	0.044	30	0.042	30	0.155	16	137.207	19	44.734

Table 6.2: SPM performance on random graphs (in million lifts)

discussed in subsection 6.1.1 with 4000 vertices and cluster size 16. For each class 30 instances were randomly generated, and each solver configuration was allowed to perform up to 10^9 lifting attempts before being aborted. Apart from initializing vertices with loops to $\top$ (the weakest form of loop removal described in Subsection 4.3.3.1), no preprocessing was applied.

Tables 6.2 and 6.3 show how many instances in each class could be solved successfully by the given configuration, and also the average number of lifts (in millions of attempts) and time taken (in seconds), respectively. Both the number of lifts and the amount of time varied greatly between configurations, so it is generally not meaningful to compare values from these columns except in those cases where all instances in a class were completely solved.

6.4.1 Discussion

First, it is interesting to see that clustered random games are a lot more difficult to solve than larger unclustered games (at least, using Small Progress Measures) and that the more levels of clustering there are (i.e. the smaller the initial cluster size is) the more difficult these games become.

On the direction axis, it turns out that forward swiping works best for linear lifting. Perhaps surprisingly, the alternating variant requires significantly more lifting attempts (though well below the theoretical maximum of twice the number required for the forward swiping alternative). Manual inspection of the benchmark results reveals that the alternating variant, on average, requires fewer successful lifting attempts but fails more lifting attempts, which results in a net loss in efficiency.

For the predecessor lifting strategy, queue-like (rather than stack-like) extraction order tends to solve the most cases. Since this order has the additional advantage that it is guaranteed to perform no worse than linear lifting (in the worst case) it seems reasonable to prefer queue-like

		4000/0		8000/0		16000/0		4000/16		4000/64	
		#solved	avg. time	#solved	avg. time	#solved	avg. time	#solved	avg. time	#solved	avg. time
Linear	F	27	2.967	25	2.832	25	3.019	0	0.000	0	0.000
Linear	A	27	3.758	25	2.522	25	3.832	0	0.000	0	0.000
Predecessor	Q	30	0.832	29	2.133	28	3.521	0	0.000	4	48.956
Predecessor	S	29	3.744	28	3.123	22	5.183	0	0.000	0	0.000
Predecessor	Q I	30	0.424	30	2.163	29	3.638	0	0.000	4	22.101
Predecessor	S I	29	1.439	29	2.376	25	2.126	0	0.000	0	0.000
Max. Measure	Q I	29	9.308	29	6.507	27	21.550	0	0.000	1	452.351
Max. Measure	S I	30	1.996	30	14.583	29	53.557	0	0.000	0	0.000
Max. Measure	H I	30	28.658	30	5.341	26	11.586	0	0.000	1	460.012
Max. Step	S I	30	8.550	30	36.330	28	29.250	0	0.000	1	542.782
Max. Step	Q I	30	8.088	30	40.261	28	29.791	0	0.000	1	515.101
Max. Step	H I	30	4.096	30	16.013	28	22.719	0	0.000	1	257.965
Min. Measure	Q I	25	69.551	24	129.567	9	195.020	0	0.000	0	0.000
Min. Measure	S I	26	89.297	24	111.767	8	286.991	0	0.000	0	0.000
Min. Measure	H I	26	60.572	23	74.569	9	191.763	0	0.000	0	0.000
Linear	F T	29	2.740	27	0.107	26	0.673	1	68.537	5	25.532
Linear	A T	29	3.791	28	2.431	26	0.644	1	74.418	5	28.675
Predecessor	Q T	30	0.438	29	0.041	30	4.027	17	19.126	21	10.727
Predecessor	S T	30	2.173	30	3.901	28	4.343	0	0.000	5	79.944
Predecessor	Q I T	30	0.238	30	2.083	30	2.850	18	18.751	21	5.544
Predecessor	S I T	30	0.163	30	2.118	28	3.124	0	0.000	10	71.695
Max. Measure	Q I T	30	0.957	30	3.428	30	16.774	0	0.000	3	421.754
Max. Measure	S I T	30	0.852	30	11.283	30	22.506	0	0.000	1	110.142
Max. Measure	H I T	30	0.658	30	5.367	30	7.714	0	0.000	6	212.710
Max. Step	S I T	30	1.539	30	13.520	30	28.391	0	0.000	9	450.345
Max. Step	Q I T	30	1.494	30	16.603	30	28.855	0	0.000	6	388.682
Max. Step	H I T	30	0.995	30	6.164	30	11.036	0	0.000	8	227.082
Min. Measure	Q I T	30	1.212	30	21.913	30	16.805	16	108.642	20	58.785
Min. Measure	S I T	30	1.145	30	6.181	30	16.363	14	108.993	18	58.838
Min. Measure	H I T	30	0.668	30	7.541	30	8.985	16	89.407	19	54.771

Table 6.3: SPM performance on random graphs (time in seconds)

behaviour. For the maximum measure lifting strategy and its variants, the results are less clear: in the one-sided framework using stack order results in at most one extra case being solved, but in the two-sided framework queue order performs better, but only by a small margin. Heap order never appears to be the best policy.

The results from the predecessor lifting strategy allow us to compare the traditional framework with the improved framework. That fewer lifting attempts are performed in the improved framework should come as no surprise, as it eliminates failed lifting attempts entirely, but what is more important, is the observation that in all cases the improved framework takes less time (roughly half as much) while solving the same number or more cases. This demonstrates that the "improved" framework is indeed an improvement in practice.

The two-sided approach turns out to be very effective on random games. Using any lifting strategy, the total number of solved games is increased, or the number of lifts required decreases by nearly an order of magnitude, or both. For some cases, these successes can be attributed to the fact that the two-way approach can solve the dual game before the normal game (in which case solving the dual game first would have benefited the one-sided approach too) but that cannot be the only reason why the two-sided approach works so well. For example, looking at the `4000/16` class of games, it seems statistically unlikely that 0 out of 30 games could be solved normally, while the dual of 17 of these games *can* be solved in time. Consequently, it seems more likely that identification of stable regions contributes significantly to the ability to solve these problems.

(It should be noted that the generated random games are not completely symmetric because the highest priority, 0, always corresponds to Even, while the lowest corresponds to Odd. However, this means the *dual* game should be harder to solve, at least in the worst case, as the upper bound on the number of lifts mentioned in Subsection 4.2 implies. For random games in practice, this line of reasoning also proves to be true: dual games are slightly harder than normal games, though

the margin gets smaller as the number of priorities increases.)

Finally, the minimum measure propagation strategy works well only with the two-sided approach, and in that case it outperforms the maximum measure and maximum step approaches, but not the predecessor lifting strategy. Indeed, the predecessor lifting strategy appears to be the overall most effective strategy for these kinds of games. In the improved framework and using queue order, it solves the most cases in every set of random games, while using relatively little time. All in all, these results do not bode well for the more sophisticated lifting strategies. However, random cases are rather synthetic and (even with clustering) not very similar in structure to parity game instances that might arise in practice.

6.5 Results on non-random games

To get a better view of how well different lifting strategies work on practical cases, we should also run benchmarks on the non-random cases described in Section 6.2. However, to reduce the number of variables involved in the analysis, we consider only queue-like extraction order and the improved framework for the predecessor lifting strategy, since the previous experiments have shown these to be the most efficient. As a result, these benchmarks focus on two aspects: the differences between lifting strategies, and the benefits of the two-sided solution approach.

The strategies reported on are the linear lifting ("lin"), predecessor lifting ("pred"), minimum-measure propagation ("min"), maximum-measure propagation ("max") and the maximum-step strategy ("step"). For linear lifting, the standard framework was used, and for the others, the improved framework described in Subsection 4.3.1. Each configuration was allowed to run for up to an hour.

Figures 6.1 and 6.2 show the results in number of lifting attempts required to solve the test cases for the decision procedures and elevator verification cases respectively.

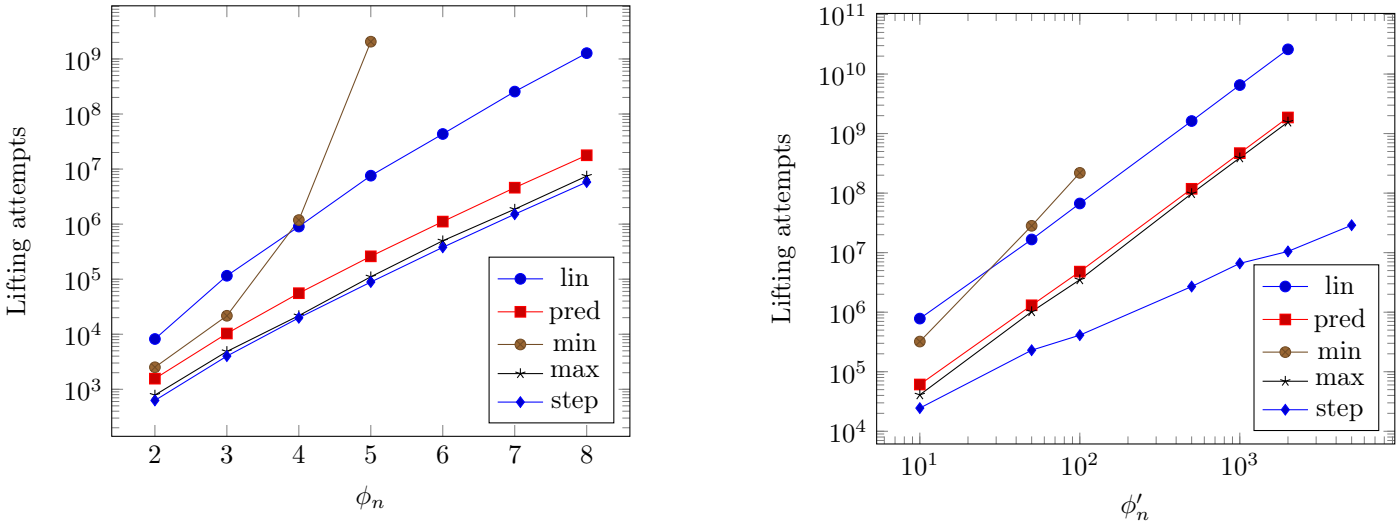


Figure 6.1: SPM lifting strategy performance on decision procedures (in lifting attempts)

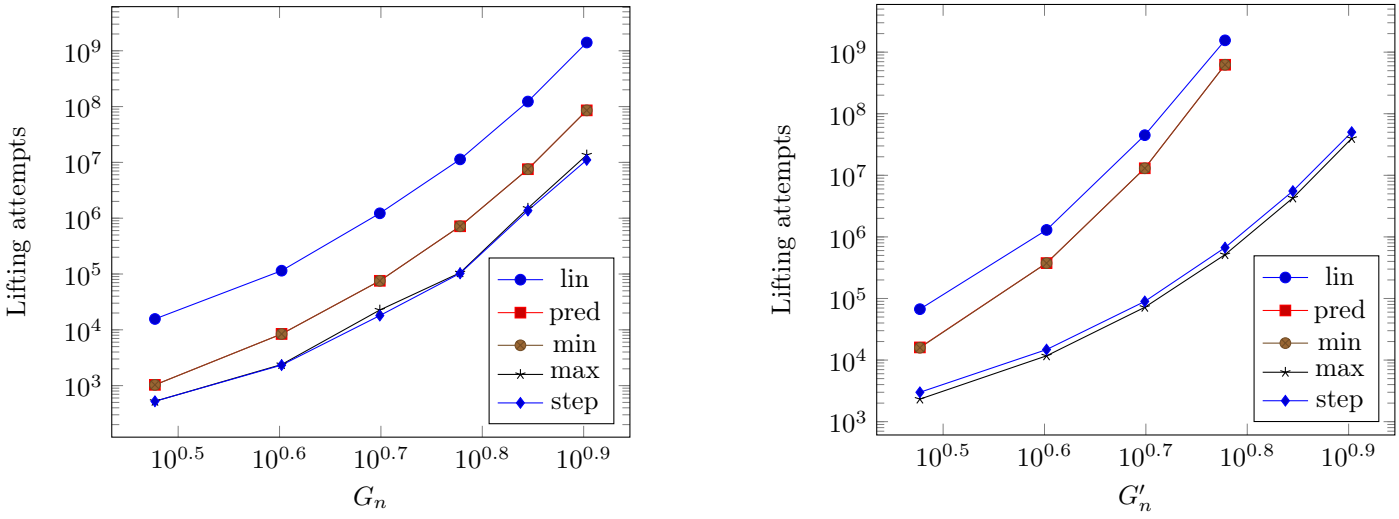


Figure 6.2: SPM lifting strategy performance on elevator verification (in lifting attempts)

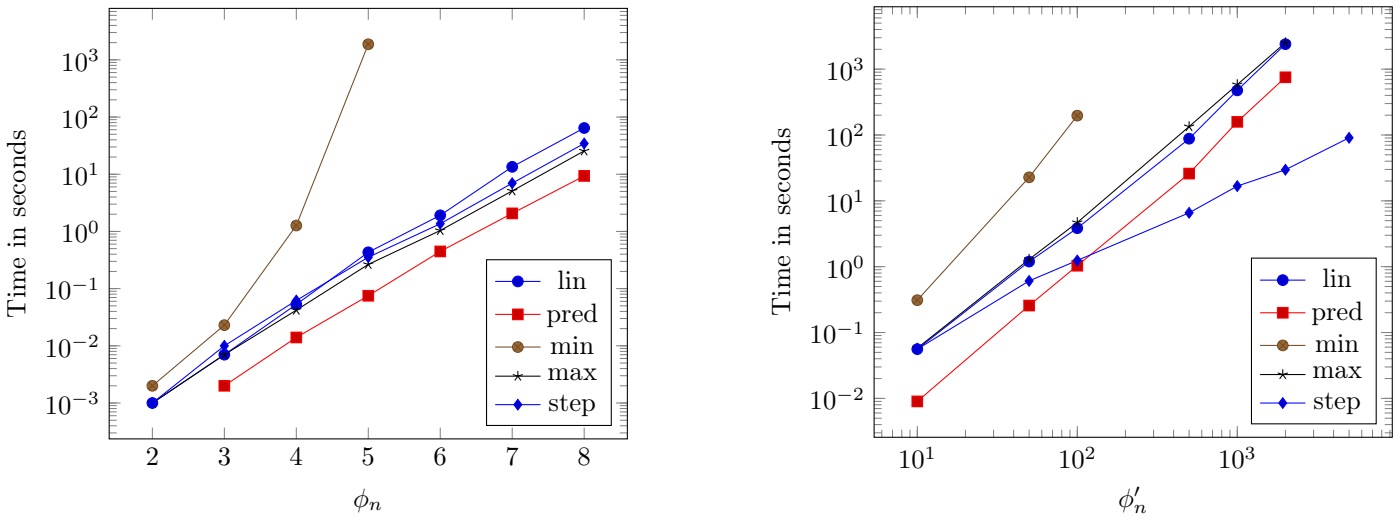


Figure 6.3: SPM lifting strategy performance on decision procedures (in time in seconds)

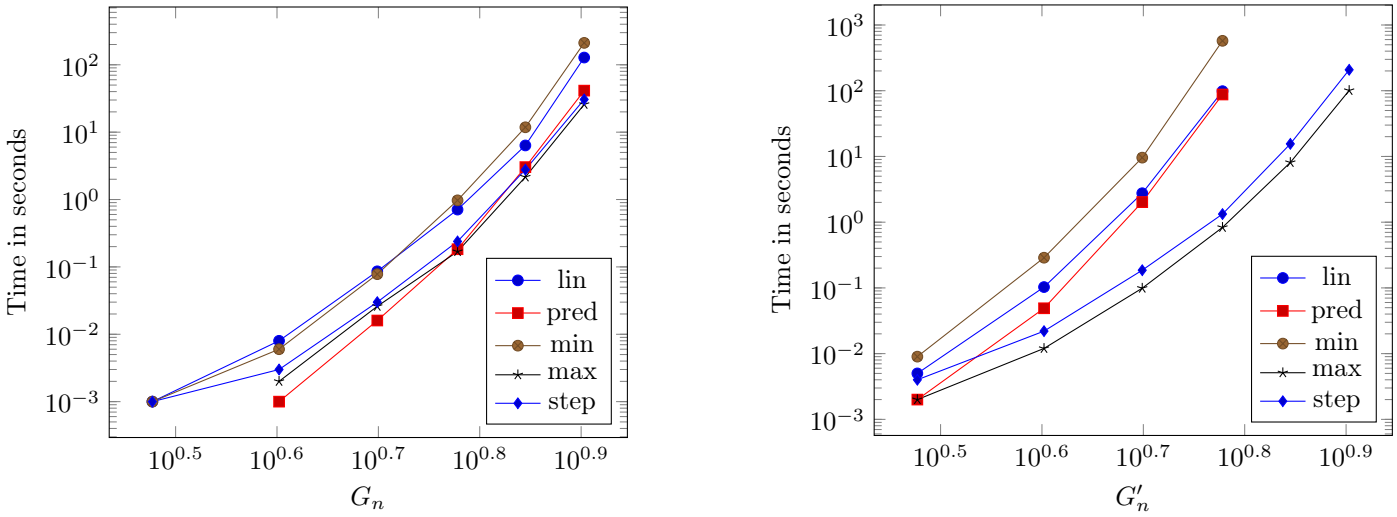


Figure 6.4: SPM lifting strategy performance on elevator verification (in time in seconds)

6.5.1 Discussion

As expected, the linear lifting strategy performs the most work, and times out on some of the more difficult cases (however, so do some of the more sophisticated alternatives). The minimum measure propagation strategy was not expected to work well in this configuration, and indeed it does poorly on the decision procedures. However, on the elevator cases it appears to work just as well as the predecessor lifting strategy.

Counting lifting attempts only, the maximum measure and maximum step lifting strategies are the best two. It seems fair to call the maximum step metric the overall winner, because even in cases where it is not the best strategy, it is fairly close to the best alternative (maximum measure propagation) and in the ϕ'_n cases it is significantly better than any other strategy.

6.5.1.1 Time versus lifts

Measuring the total time used to solve each case (instead of the number of lifting attempts) paints a slightly different picture, as is shown in Figures 6.3 and 6.4. The linear lifting strategy looks better compared to more sophisticated strategies, though (excepting minimum measure propagation) it is still the slowest. The predecessor lifting strategy now wins in the first case (ϕ_n) since the other strategies, while requiring fewer lifts, run considerably slower. However, for the second case (ϕ'_n) the maximum step strategy is still the winner since it scales much better to larger test cases. On the fair elevator case (G_n) differences in performance are small, but on the unfair elevator (G'_n) both the maximum measure and maximum step metric beat the simpler strategies by a large margin.

6.5.1.2 Two-sided approach

For completeness sake, the two-sided approach is also tested on these cases. As mentioned in Subsection 6.2.1 small gains are expected, but it is still useful to quantify the overhead of the two-sided approach due to switching between solving the game for player Even and Odd (and, as an implementation quirk, clearing the lifting strategy queue in between).

Figures 6.5 and 6.6 show the differences between one-sided and two-sided solving using different lifting strategies for one instance of each class of games. As expected, the two-sided approach is about twice as slow, except in the fourth case (unfair elevator verification, G') where it is much faster. However, the unfair elevator case is different from the others: for these games the dual is easier to solve than the normal game. Solving the dual game using the one-way approach is again faster than using the two-way approach, as the last bar chart in Figure 6.6 shows.

It is worth noting that in this last case, the SPM vector bounds give no indication that the dual game would be easier to solve. The elevator verification cases have only three priorities (0, 1 and 2) and in $G'8$ these occur 651 168, 1 793 672 and 5 369 176 times respectively. Using the formula presented in Subsection 4.2, an approximation of the maximum number of lifts for the normal game is therefore $1\,793\,672 \times (1\,793\,672 + 5\,369\,176)$ versus $651\,168 \times (651\,168 + 1\,793\,672 + 5\,369\,176 \times 5\,369\,176)$ for the dual game: the second value is orders of magnitude larger, yet the dual game is faster to solve in practice. This shows once again that worst-case estimations do not accurately predict how quickly games can be solved in practice.

From these charts it becomes clear that for the test cases under consideration (and unlike the random cases) the two-sided solving approach doesn't offer a significant improvement over the one-sided approach. Although it can be argued the two-sided approach is still useful when it is unknown which side of the game will be easier to solve (and in general this *is* hard to predict, as we saw in the G' cases) the same result can also be achieved (arguably in a simpler way) by solving the dual game independently in parallel.

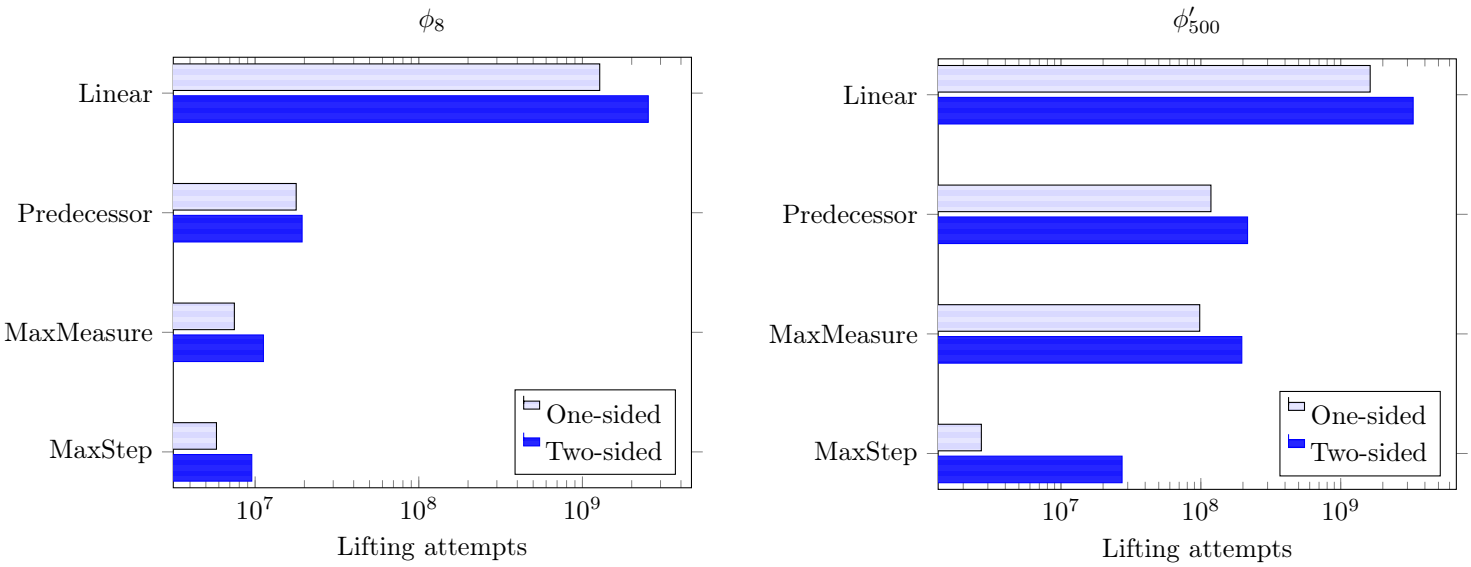


Figure 6.5: One-way vs two-way solving on decision procedures cases

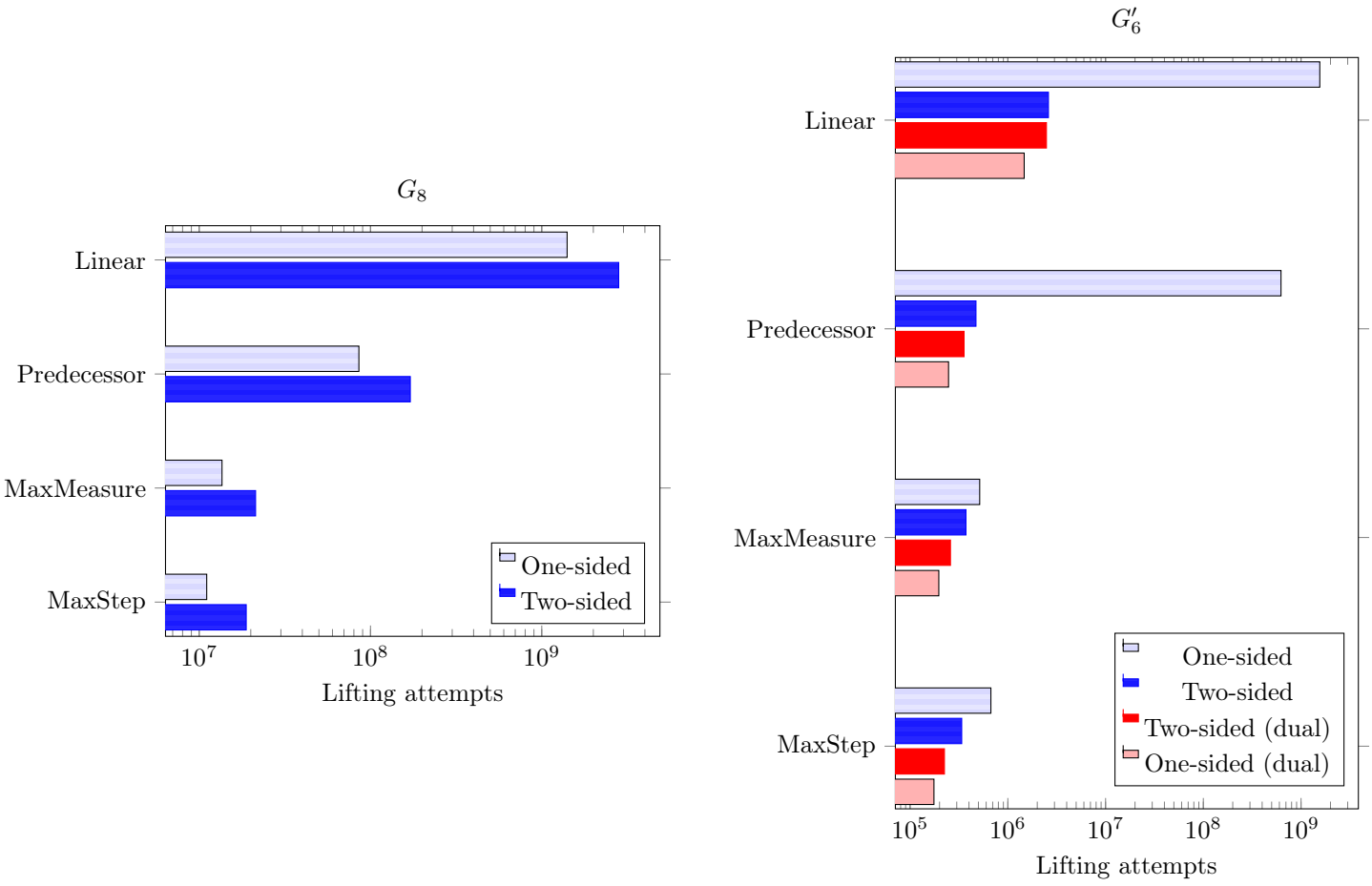


Figure 6.6: One-way vs two-way solving on elevator verification cases

6.6 Effectiveness of bounds reduction after lifting to top

The optimization described in Subsection 4.3.2 (reducing the bounds on the progress measure vectors after lifting a vertex to $\top$) can be evaluated on the same test cases. For the non-random cases, it turns out that vector updates never overflow, which should come as no surprise considering vertices are never lifted to $\top$ (except by propagating the $\top$ values assigned to vertices with loops) and as a result the optimization has no effect. The (normal) unfair elevator case, which doesn't have the above property, is the only exception: in those cases, the optimization reduces the number of lifts needed by about 2.5%.

The random graphs should be a little more interesting. We have performed the same experiments using the predecessor lifting strategy only, and only considering cases that can be solved with the optimization enabled. For each such case, the solver was run with the bounds reduction optimization disabled. Table 6.4 shows the total number of lifts required with the optimization disabled and enabled respectively (as well as the ratio between the two) per test case and solver configuration (in this case, whether or not the two-sided solving approach was used).

Graph	Cluster	Two-sided	Cases	Lifts (base)	Lifts (reduced)	Ratio
4 000	0	no	30	494 317 259	119 932 439	0.242622
8 000	0	no	30	1 513 641 376	602 220 938	0.397862
16 000	0	no	29	916 432 321	908 309 218	0.991136
4 000	64	no	4	4 890 723 902	790 592 155	0.161651
4 000	0	yes	30	374 374 889	44 696 952	0.119390
8 000	0	yes	30	2 657 833 500	388 786 325	0.146279
16 000	0	yes	30	4 201 582 569	669 228 047	0.159279
4 000	64	yes	21	4 939 903 038	596 019 198	0.120654
4 000	16	yes	18	35 295 413 593	1 791 402 821	0.050754

Table 6.4: Effectiveness of bounds reduction after lifting to top

The data in this table support two conclusions. First, that the optimization is more effective for the relatively compact-but-difficult clustered random cases than for the large-but-simple random games. Second, that it is more effective for the two-sided approach than the one-sided approach. For the two-sided approach, the optimization speeds up solving by a factor 6-10, while for the one-sided approach a factor 1-6 seems more typical.

It should be noted that the benefit of enabling the optimization varies greatly between individual cases, though it is never detrimental. For example, the third row of the table shows a set of 29 cases which benefit little from the optimization. On the other hand, one of the 18 cases reported in the last row required 27 billion lifts; nearly 30 times as much as the roughly 900 million lifts required with the optimization enabled. (Excluding this outlier would nearly double the ratio from 0.050754 to 0.107564 for the last row.)

Because of these large individual differences, the ratios reported should be taken with a grain of salt. Accurate results would require much larger test cases. However, it seems fair to conclude that these benchmark results confirm that bounds reduction after lifting to top is an effective way to speed up Small Progress Measures in practice, especially for the two-sided approach.

6.7 Effectiveness of cycle removal

Table 6.5 shows how preprocessing the game using cycle removal (Decycle) and/or decomposition into strongly connected components (DeSCC) affects the total running time for nonrandom test cases. For Small Progress Measures, the predecessor lifting strategy in the improved framework was used. The bold figures mark the lowest solution time among the four possible configurations. These results show that in most cases SCC decomposition is more effective than cycle removal. However, in the fair elevator case (G_8) cycle removal provides clear benefits.

	ϕ_8		ϕ'_{1000}		G_8		G'_6	
SPM		Decycle		Decycle		Decycle		Decycle
	9.387	26.299	157.785	176.285	41.132	46.792	87.100	83.483
DeSCC	**3.290**	23.392	**26.705**	27.704	38.377	**36.750**	**46.863**	51.489
	ϕ_8		ϕ'_{1000}		G_8		G'_8	
Zielonka		Decycle		Decycle		Decycle		Decycle
	350.226	24.059	0.317	0.897	11.475	**6.870**	**8.050**	10.894
DeSCC	**2.850**	22.920	**0.225**	0.939	11.148	10.673	16.382	16.014

Table 6.5: Effectiveness of cycle removal on nonrandom cases (time in seconds)

These results are not entirely unsurprising since the games for ϕ' contain no winner-controlled cycles except for a single vertex with maximum priority that contains a loop, so in these cases, cycle removal is equivalent to loop removal. The games for ϕ contain a similar loop, and also a small winner-controlled cycle for the other player with a large attractor set that allows roughly half of the game to be solved by the decycle solver. Surprisingly, this appears to benefit Zielonka's recursive algorithm, but not Small Progress Measures. However, cycle removal still is not very useful there, because better results can be achieved by only applying SCC decomposition. (This reinforces the idea, as suggested in [12] and [17], that Zielonka's algorithm performs best when combined with SCC decomposition.)

For the fair elevator case, it is interesting that cycle removal combined with SCC decomposition yields best results when solving with Small Progress Measures, but adding SCC decomposition only slows down Zielonka's algorithm in this case.

Finally, it should be noted that in all cases where the preprocessing increased the total solving time, manual inspection of the solver output showed that this increase can be entirely ascribed to the time spent on preprocessing. The time spent by the general solver never increases, so these preprocessing steps do not seem to increase the difficulty of the underlying games.

Table 6.6 shows the running times on random graphs of 10^7 vertices, average outdegree 3 and 10 different priorities. The clustered games are generated with cluster size 100. Running times are averaged over 10 trials per configuration with different random graphs (though resulting timings varied little). Since these games are much too large to solve with Small Progress Measures, which already gets stuck on graphs with a few hundred thousand vertices, benchmarks have been performed using Zielonka's recursive algorithm only, which tends to be very effective at solving random games.

In fact, Zielonka's algorithm is so effective that solving the unclustered random games directly is faster than applying any sort of preprocessing. SCC decomposition adds 14 seconds of preprocessing overhead (without benefit, since these games tend to consist of one large component) and although cycle removal can solve such games entirely without invoking the general solver, this still takes somewhat longer than solving the game directly.

The clustered random games are somewhat more difficult to solve, and here the preprocessing algorithms are beneficial. Roughly 90% of the game can be solved using cycle removal, and running SCC decomposition afterwards further simplifies the problem. In these cases, preprocessing operations actually speed up the solution process.

	Unclustered Random		Clustered Random	
Zielonka		Decycle		Decycle
	8.493	8.700	23.834	13.928
DeSCC	22.516	8.696	25.708	**12.715**

Table 6.6: Effectiveness of cycle removal on nonrandom cases (time in seconds)

To summarize, cycle removal is highly effective on random games, on which SCC decomposition is not effective at all. Zielonka's algorithm is already quite good at solving random games (especially of the unclustered variety) so this benefit applies primarily to Small Progress Measures. However, results on clustered random games show that even Zielonka's algorithm can benefit from these optimizations.

On nonrandom cases, the results are mixed: in only one of the four cases described here cycle removal caused a significant speed-up. However, the fact that such cases exist suggests that it is a useful optional feature.

Chapter 7

Conclusion

Most of my research has focussed on improving the performance of Jurdziński's Small Progress Measures algorithm as part of a practical solver tool. After designing and implementing various improvements to the most basic algorithm, and executing a number of benchmark experiments to assess their effects, I can draw several conclusions:

- Advanced lifting strategies which take information about the current progress measure into account (see Subsection 4.2.5) can significantly reduce the total number of lifting attempts needed to solve a parity game, although this does not always reduce the total amount of time required to solve a game. When solving for only one player at a time, the maximum step heuristic performs best. For Friedmann's two-sided solving approach, both the predecessor and the minimum measure propagation heuristics work well.
- As a compromise between simplicity and performance, the queue-based predecessor lifting strategy was consistently among the best methods available. Although it is not a new strategy, it has never been compared empirically with its alternatives. In most research prototypes, the even simpler yet otherwise inferior linear lifting strategy is used. The recommendation to implement the predecessor lifting strategy instead is supported by the benchmark results presented earlier.
- The core algorithm as described by Jurdziński is suitable mainly for the very simple linear lifting strategy. When implementing the predecessor lifting strategy, or one of the more advanced lifting strategies, it is benefical to reformulate the core algorithm to eliminate the possibility for failed lifting attempts (see Subsection 4.3.1). The empirical evaluation shows this roughly doubles its performance.
- The two-sided variant of Small Progress Measures (see Subsection 4.3.4), originally implemented by Oliver Friedmann in PGSolver but not described in literature, was also evaluated and shown to work well on particularly difficult cases such as clustered random games. It has been shown to benefit greatly from the optimization described in Subsection 4.3.2, so it is recommended that these two improvements are implemented together.

The main conclusion regarding Zielonka's recursive algorithm is that my benchmarks confirm earlier reports that this algorithm is very versatile, and quite good at solving even difficult test cases. It can additionally benefit from preprocessing techniques like SCC decomposition. Based on these benchmark results, and the fact that the algorithm is not much more complicated than Small Progress Measures, I conclude that Zielonka's algorithm is generally to be preferred over Small Progress Measures as a practical algorithm for solving parity games, despite its theoretically worse runtime complexity.

A new preprocessing operation called winner-controlled cycle removal (see Subsection 4.3.3.2) has been introduced as a generalization of loop removal. Besides being an effective way to solve unclustered random games, benchmarks have shown that it can speed up parity game solving using both Small Progress Measures and Zielonka's recursive algorithm.

7.1 Future work

During the project I identified many opportunities for further research that could not fit in the scope of my project. The most important topics will be listed below, with the hope that others will be able to further improve upon the state of the art.

7.1.1 Symbolic solving

Throughout this project, I worked with an explicit representation of the parity game graph. Although this representation is simple and efficient to work with, these graphs are often large and difficult to generate efficiently from higher-level problems, creating a performance bottleneck even before the parity game solver is invoked. When used as a vehicle for formal verification, generating and solving parity games symbolically might be a more practical approach.

Suitable symbolic representations would also allow infinite game graphs to be encoded, allowing entirely new classes of games to be solved. Zielonka's algorithm is already known to work on infinite graphs and can be adapted to a symbolic representation.

Some work in this direction has already been done. A symbolic approach to solving parity games derived from μ-calculus model checking problems is presented in [2], which has also been implemented and evaluated, but not directly compared with explicit approaches. A symbolic representation of Boolean equation systems based on Linked Decision Diagrams (LDDs) is introduced in [6], as well as a method to solve these structures by adapting Zielonka's recursive algorithm. Similarly, work is under way to add symbolic solving of parity games to the LTSmin model checker [26], using Zielonka's algorithm and a game representation based on Binary Decision Diagrams (BDDs).

These developments look promising. For future work, it seems useful to evaluate their practical performance in greater detail, comparing the performance between different symbolic approaches and with earlier explicit methods.

7.1.2 Concurrent and/or distributed solving

Although computer hardware is still getting exponentially faster over time, most performance gains recently have been effected by increasing the number of processor cores per chip, rather than increasing the base clock frequency of each core. This means that algorithms cannot be expected to scale up with the increase in processing power available unless they efficiently scale to multiple cores.

Currently, the best available parity game solvers run only sequentially. Although I have tried to develop concurrent (shared memory) and distributed (message passing) versions of my solvers, these did not exhibit comparable performance to their sequential counterparts, and they did not even scale very well. Localizing the work performed and minimizing the required communication (either through message passing or shared data structures) between threads or processes is difficult.

Although some previous work has been done parallelizing Small Progress Measures, and not entirely without success, the resulting solvers still could not outperform more sophisticated algorithms (or even more sophisticated variants of Small Progress Measures, such as those discussed in this report). Van de Pol and Weber report that their solver times out on uniform random graphs of about 100 000 vertices (which tend to be easy to solve using cycle removal or the two-sided approach). Discussing his parallel solver, Van der Berg concludes that: "*benchmarks show in its current implementation the algorithm is not up to par with other implementations, like MaxMeasures*" (referring to the sequential version discussed in this report). Bootsma's solver for the GPU does implement the two-sided approach, but even he concludes that "*the CPU implementation is still faster for the majority of the parity games tested*" (though it should be noted that the majority of his benchmark cases are fairly easy games).

The above algorithms are all based on Small Progress Measures, which is probably the easiest algorithm to parallelize, but does not always offer the best performance in practice. Even a fast, parallel implementation of Small Progress Measures may be outperformed by Zielonka's recursive

algorithm or e.g. Schewe's optimal strategy improvement algorithm. Parallelizing those algorithms may be more difficult but ultimately more useful.

The question of how to design efficient, non-sequential solvers that are competitive with sequential solvers, but that also scale well when more processor cores are allocated to them, remains open. Finding a satisfactory answer would be a likely topic of future work.

A somewhat easier task, to start with, would be to efficiently parallelize the various preprocessing steps that greatly help solving parity games in practice: decomposition into strongly-connected components, removal of winner-controlled loops, solving single-player games, construction of subgames, and so on. (Note that some of these subproblems, such as identifying strongly-connected components, have been studied before and may already have adequate parallel formulations.) Even when these subproblems are solved, integrating their solutions into an efficient solver without introducing sequential performance bottlenecks is a challenge on its own.

7.1.3 Improving strategy improvement

My project revolved mainly around Jurdziński's Small Progress Measures algorithm, and although I managed to make it work more efficiently in many ways, it is clear that this is not the preferred algorithm for solving parity games in practice. Parity game researchers (see Section 1.4: Related Work) typically suggest that algorithms based on strategy improvement should perform best in practice, but these algorithms have not been investigated here at all.

The baseline for performance of these algorithms is set by PGSolver. Since my work showed that with careful design and implementation, large performance gains over this baseline are possible with Small Progress Measures and Zielonka's algorithm, it seems reasonable to suspect that future work on (optimal) strategy improvement could yield similar results.

7.1.4 Better benchmark cases

Finally, empirical research on parity games seems to be hampered by the lack of a clearly established problem domain. Consequently, there is no commonly accepted set of benchmark cases to evaluate solvers by, which makes it difficult to compare the various different approaches fairly, and introduces the problem that the result of an empirical evaluation depends greatly on the benchmark set chosen by the evaluator.

Although different kinds of problems may call for different approaches (and thus different results in practice are not necessarily an indication of bad scientific work) an absolute, objective assessment of solvers is impossible without a predetermined, well-understood and commonly agreed-upon benchmark set. Such benchmark sets are common in other areas of research; for example, lossless data compression algorithms are typically benchmarked on the Calgary or Canterbury corpus. There is no equivalent benchmark set for parity game solvers. As a result, different researchers use different methods of evaluation, sometimes reaching different conclusions, which becomes problematic when test case details are underspecified and results are presented in a way that makes them appear more general than they really are.

It is not obvious what kind of games parity game solvers should be tested on. Many practitioners choose to benchmark on randomly generated games, which are easy to generate, but do not look like anything that arises in practice. The same objection applies to synthetic games designed to elicit worst-case performance in some of the published algorithms. On the other hand, many games that do occur in practice (by converting a model checking problem into a parity game, for example) appear to be structurally weak, even if they are large: they often have few different priorities, only a single controlling player, or can be simplified and solved mainly using preprocessing techniques rather than invoking the parity game algorithm proper.

To advance the state of the art, it would be useful to define the problem domain for parity game solvers more clearly, as well as to describe how test cases may be drawn from this domain, ideally resulting in a standard benchmark set that can be used to compare solvers in practice. Such a benchmark set would be invaluable for future empirical work.

Bibliography

[1] A. Antonik, N. Charlton, and M. Huth. Polynomial-time under-approximation of winning regions in parity games. *Electronic Notes in Theoretical Computer Science*, 225:115–139, 2009.

[2] M. Bakera, S. Edelkamp, P. Kissmann, and C.D. Renner. Solving μ-calculus parity games by symbolic planning. In *Model Checking and Artificial Intelligence*, pages 15–33. Springer, 2009.

[3] H. Björklund, S. Sandberg, and S. Vorobyov. A discrete subexponential algorithm for parity games. In *STACS 2003*, pages 663–674. Springer, 2003.

[4] S.C.C. Blom, J. Van de Pol, and M. Weber. LTSmin: Distributed and symbolic reachability. In *Computer Aided Verification*, pages 354–359. Springer, 2010.

[5] P.J.A Bootsma. Speeding up the small progress measures algorithm for parity games using the GPU. Master's thesis, 2013.

[6] T. Boshoven. A symbolic approach to PBES instantiation. Master's thesis, 2013.

[7] S. Cranen, J.J.A. Keiren, and T.A.C. Willemse. Stuttering mostly speeds up solving parity games. In *NASA Formal Methods*, pages 207–221. Springer, 2011.

[8] S. Cranen, J.J.A. Keiren, and T.A.C. Willemse. A cure for stuttering parity games. In *Theoretical Aspects of Computing–ICTAC 2012*, pages 198–212. Springer, 2012.

[9] E.A. Emerson. Model checking and the mu-calculus. *DIMACS series in discrete mathematics*, 31:185–214, 1997.

[10] E.A. Emerson and C.S. Jutla. Tree automata, mu-calculus and determinacy. In *Foundations of Computer Science, 1991. Proceedings., 32nd Annual Symposium on*, pages 368–377. IEEE, 1991.

[11] O. Friedmann. Recursive Solving Of Parity Games Requires Exponential Time. Unpublished. URL: `http://www2.tcs.ifi.lmu.de/~friedman/papers/recursive_lower_bound.pdf`.

[12] O. Friedmann and M. Lange. Solving parity games in practice. *Automated Technology for Verification and Analysis*, pages 182–196, 2009.

[13] O. Friedmann and M. Lange. MLSolver, 2013. URL: `https://github.com/tcsprojects/mlsolver`.

[14] O. Friedmann and M. Lange. PGSolver, 2013. URL: `https://github.com/tcsprojects/pgsolver`.

[15] C. Fritz. *Simulation-based simplification of omega-automata.* PhD thesis, University of Kiel, 2005.

[16] C. Fritz and T. Wilke. Simulation relations for alternating parity automata and parity games. In *Developments in Language Theory*, pages 59–70. Springer, 2006.

[17] M. Gazda and T.A.C. Willemse. Zielonka's recursive algorithm: dull, weak and solitaire games and tighter bounds. In *Proceedings Fourth International Symposium on Games, Automata, Logics and Formal Verification*, 2013.

[18] J.F. Groote and M. Keinänen. A sub-quadratic algorithm for conjunctive and disjunctive boolean equation systems. In *Theoretical Aspects of Computing–ICTAC 2005*, pages 532–545. Springer, 2005.

[19] J.F. Groote, J.J.A. Keiren, A. Mathijssen, B. Ploeger, F. Stappers, C. Tankink, Y. Usenko, M. Van Weerdenburg, W. Wesselink, T.A.C. Willemse, and J. Van der Wulp. The mCRL2 toolset. In *Proceedings of the 17th Annual Symposium on Theoretical Aspects of Computer Science*. Springer, 2008. URL: `http://www.mcrl2.org/`.

[20] J.F. Groote and T.A.C. Willemse. Parameterised boolean equation systems. *Theoretical Computer Science*, 343(3):332–369, 2005.

[21] K. Heljanko, M. Keinänen, M. Lange, and I. Niemelä. Solving parity games by a reduction to SAT. *Journal of Computer and System Sciences*, 78(2):430–440, 2012.

[22] M. Jurdziński. Deciding the winner in parity games is in UP and co-UP. *Information Processing Letters*, 68(3):119–124, 1998.

[23] M. Jurdziński. Small Progress Measures for Solving Parity Games. In *Proceedings of the 17th Annual Symposium on Theoretical Aspects of Computer Science*, pages 290–301. Springer-Verlag London, UK, 2000.

[24] M. Jurdziński, M. Paterson, and U. Zwick. A deterministic subexponential algorithm for solving parity games. In *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 117–123. ACM, 2006.

[25] J. Kandziora. Playing Parity Games on the Playstation 3. In *10th Twente Student Conference on IT*, 2009.

[26] G. Kant. spgsolver - symbolic parity game solver, 2013. URL: `http://fmt.cs.utwente.nl/tools/ltsmin/doc/spgsolver.html`.

[27] G. Kant and J. Van de Pol. Efficient Instantiation of Parameterised Boolean Equation Systems to Parity Games. In *GRAPHITE*, pages 50–65, 2012.

[28] J.J.A Keiren. An experimental study of algorithms and optimisations for parity games, with an application to Boolean Equation Systems. Master's thesis, Eindhoven University of Technology, 2009.

[29] J.J.A. Keiren and T.A.C. Willemse. Bisimulation minimisations for boolean equation systems. In *Hardware and Software: Verification and Testing*, pages 102–116. Springer, 2011.

[30] N. Klarlund. Progress measures, immediate determinacy, and a subset construction for tree automata. *Annals of Pure and Applied Logic*, 69(2-3):243–268, 1994.

[31] 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*, volume 5, 2005.

[32] D.A. Martin. Borel determinacy. *The Annals of Mathematics*, 102(2):pp. 363–371, 1975. URL: `http://www.jstor.org/stable/1971035`.

[33] R. McNaughton. Infinite games played on finite graphs. *Annals of Pure and Applied Logic*, 65(2):149–184, 1993.

[34] J. Obdržálek. *Algorithmic analysis of parity games*. PhD thesis, University of Edinburgh, 2006.

[35] S. Schewe. Solving parity games in big steps. *FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science*, pages 449–460, 2007.

[36] S. Schewe. An Optimal Strategy Improvement Algorithm for Solving Parity and Payoff Games. In *Proceedings of the 22nd international workshop on Computer Science Logic*, pages 369–384. Springer, 2008.

[37] R.E. Tarjan. Depth-first search and linear graph algorithms. In *Switching and Automata Theory, 1971., 12th Annual Symposium on*, pages 114–121, 1971.

[38] J.C. Van de Pol and M. Weber. A multi-core solver for parity games. *Electronic Notes in Theoretical Computer Science*, 220(2):19–34, 2008.

[39] F. Van der Berg. Solving Parity Games on the Playstation 3. In *13th Twente Student Conference on IT*, 2010.

[40] J. Vöge and M. Jurdziński. A Discrete Strategy Improvement Algorithm for Solving Parity Games. In *Proceedings of the 12th International Conference on Computer Aided Verification*, pages 202–215. Springer-Verlag London, UK, 2000.

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

□