Exploring and implementing quasi-polynomial time algorithms for solving parity games

Mateusz Parafiński

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2019

Abstract

A recent breakthrough by Calude et al. gave rise to multiple quasi-polynomial time algorithms for solving parity games, beating the prior best known bound of $n^{O(\sqrt{n})}$. A natural question to ask is whether they can compete with exponential-time algorithms that are known to perform well in practice. This document attempts to answer this question and focuses on two quasi-polynomial parity game solvers – succinct progress measures by Jurdziński and Lazić and register games by Lehtinen.

We investigate both algorithms from a theoretical and practical perspective, giving an overview of each approach and discussing the design of our implementation of these algorithms in OCaml for the PGSolver library. The succinct progress measures algorithm is also evaluated against other solvers using procedurally generated parity games of different sizes and types.

The results suggest that despite their better time-complexity-theoretic upper bound, the algorithms analysed are not competitive against the best known solvers. Register games struggle with their quasi-polynomial space requirements while succinct progress measures perform similarly to small progress measures by Jurdziński but worse than the recursive or strategy improvement algorithms.

Acknowledgements

First and foremost, I would like to thank my supervisor, Kousha Etessami, for answering all my questions and insightful discussions about parity games and much more.

I would also like to thank Marcin Jurdziński and Karoliina Lehtinen for useful suggestions and comments about their work.

Last but not least, I would like to thank my family and friends for continuous support throughout the last four years. I would not have done it without you.

Contents

1 Introduction

Parity games are two-player zero-sum games on graphs in which each of the two players tries to satisfy their so-called *parity condition*. In 1991 Emerson and Jutla gave a proof of memoryless determinacy of parity games [EJ91] and since then they are actively studied, both theoretically and empirically.

Over the years many algorithms for solving parity games appeared in the literature, but it wasn't until 2017 that Calude et al. [CJK+17] presented the first quasi-polynomial one. Despite its impracticality caused by space requirements, it paved the way for other quasi-polynomial solutions which appeared soon after.

In this document, we discuss two such algorithms – *succinct progress measures* and *register games*, introduce their foundations, analyse the time and space requirements and propose an implementation in OCaml for the PGSolver library. Appendix D briefly explains the library structure and points to our implementation.

1.1 Goals

The goal of this document is to present, analyse and implement the aforementioned quasi-polynomial time algorithms for solving parity games, as well as address the question of their practicality and compare them with other existing methods.

1.2 Content

We start our journey by introducing parity games and stating their properties in section 2. In there, we also briefly discuss other notable parity game solving algorithms, their implementations and the importance of parity games.

In section 3 we talk in depth about the small progress measures [Jur00] and the succinct progress measures [JL17]. We also propose an implementation of the succinct progress measures algorithm, justifying the design decisions and discussing possible improvements.

The register games algorithm [Leh18] is introduced in section 4. We present the approach, give a few examples of it in practice and discuss various aspects of the implementation and the practicality of the algorithm. We also mention a major problem in the implementation – strategy synthesis – and suggest possible solutions based on a conversation with the author of the algorithm, Karoliina Lehtinen.

Finally, the comparison of the succinct progress measure algorithm with some of the well-known algorithms for solving parity games is done in section 5. We start by presenting our methodology and discuss the results of the experiments. This is followed up by a conclusion and suggestions for improving the implementation, partially influenced by an email conversation with Marcin Jurdziński, one of the authors of [JL17].

2 Background

This section introduces the notion of a parity game, both formally and informally. We describe how the game is played, what are the winning conditions and give a few examples. We also define what it means to solve the game and combine that with a discussion on memoryless determinacy of parity games, their most fundamental property.

To give a broader picture, we also briefly go through the history of parity game research, explain their importance and mention two practical tools for solving them.

2.1 Parity games

Let's start with a formal definition of a parity game.

Definition 2.1 (Parity game)**.** *A **parity game** is a two-player zero-sum game $G = (V, V_e, V_o, E, \pi)$ with the following components*

- *V, the set of vertices,*
- *E, the set of edges, $E \subseteq V^2$,*
- *V_e, V_o, disjoint subsets of vertices, one for each player (**Even** and **Odd**),*
- *$\pi : V \mapsto \mathbb{N}$, the priority function assigning each vertex a natural number.*

More informally, a parity game is a two-player game on a directed graph where each node has its given priority in the form of a natural number and belongs to exactly one of the two players which we will refer to as **Even** and **Odd**. Note that somewhat counter-intuitively, the priority of a node does not determine its owner – as we will see, the names of the players come from the definition of the winner of the game. Below we can see the first example of a parity game.

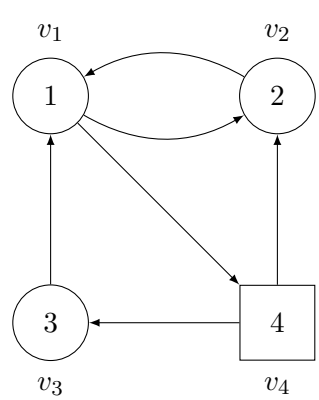

Figure 1: A parity game with priorities inside the nodes. Circle nodes belong to Odd and square nodes to Even.

Following definition 2.1, the parity game in figure 1 would be described as $G = (V, V_e, V_o, E, \pi)$, where

$$\begin{aligned} V &= \{v_1, v_2, v_3, v_4\} \\ V_e &= \{v_4\} \\ V_o &= \{v_1, v_2, v_3\} \\ E &= \{(v_1, v_2), (v_1, v_4), (v_2, v_1), (v_3, v_1), (v_4, v_2), (v_4, v_3)\} \\ \pi(v_i) &= i. \end{aligned}$$

2.1.1 Playing parity games

Let $G = (V, V_e, V_o, E, \pi)$ be a parity game. Each play of the game starts in some vertex $v_0 \in V$. The owner of that vertex chooses one of its successors, i.e. some v_1 such that $(v_0, v_1) \in E$ and moves to v_1. Subsequently, the owner of v_1 performs another move to one of its successors, say v_2, and so on. In this manner we create an infinite *play* $(v_0, v_1, v_2, \ldots)$.

In such a play, some of the vertices will occur infinitely often. Let $\hat{\pi}$ be the *maximum*[1] priority of one of the vertices occurring infinitely often, that is

$$\hat{\pi} = \max\{\pi(v) \mid v \text{ occurs infinitely often in the play}\}.$$

Then Even is declared a winner if $\hat{\pi}$ is even and Odd is declared a winner otherwise.

Example. Consider the game in figure 2. If the game starts at v_1, then there is only one possible infinite play, namely $(v_1, v_2, v_3, v_1, v_2, v_3, \ldots)$. Clearly, all three priorities occur infinitely often in such a play and the maximum one is $\hat{\pi} = \pi(v_3) = 3$, so Odd wins this game.

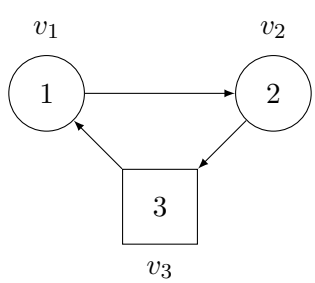

Figure 2: A parity game always won by Odd. Circle nodes belong to Odd and square to Even.

△

[1]Some versions of the game, as for example [Jur00] define the *minimum* priority occurring infinitely often to be the one determining the winner, however most of the recent papers use the definition presented above. Those two approaches are equivalent.

It's important to note that *every play has a winner*. To see that, let p denote the maximum priority in the game, $p = \max\{\pi(v) \mid v \in V\}$. Clearly, in an infinite play $(v_0, v_1, \ldots)$, the sequence $(\pi(v_0), \pi(v_1), \ldots)$ is bounded above by p. Thus there is some priority occurring infinitely often that is less than or equal to p and such a priority can be either even or odd, guaranteeing a win for one of the players.

Players could play the game in an arbitrary fashion, picking one of their successors arbitrarily at each move or they can follow some set of rules that determine their move for each possible situation appearing in a play. Such a set of rules, one for each player, is called a *strategy*. A player is said to *follow* a strategy if at every possible move they pick the next node according to the rules described by their strategy.

More formally, let S be the set of all possible finite sequences of nodes appearing in the game

$$S = \{(v_0, v_1, v_2, \ldots) \mid (v_i, v_{i+1}) \in E \text{ for all } 0 \leq i < k,\ k \in \mathbb{N}\}.$$

Also, let S_e, S_o be the subsets of S containing all the sequences with the last vertex belonging to V_e and V_o respectively, i.e.

$$S_e = \{(v_0, \ldots, v_k) \in S \mid v_k \in V_e\}$$
$$S_o = \{(v_0, \ldots, v_k) \in S \mid v_k \in V_o\}.$$

Clearly $S_e \cup S_o = S$ and $S_e \cap S_o = \emptyset$. With this in mind, we can define a strategy.

Definition 2.2. *A* ***strategy*** σ_e *for player Even is a mapping* $\sigma_e : S_e \mapsto V$ *such that for each* $(v_0, \ldots, v_k) \in S_e$ *it is true that* $(v_0, \ldots, v_k, \sigma_e(v_0, \ldots, v_k)) \in S$*. In other words, the mapping* σ_e *picks one of the successors of the last node for each finite play. A strategy* $\sigma_o : S_o \mapsto V$ *for Odd is defined equivalently.*

Consider a parity game $G = (V, V_e, V_o, E, \pi)$ with the starting vertex $v_0 \in V$. Recall that the Even wins the game if the highest priority occurring infinitely often is even and Odd wins otherwise. This allows us to define a *winning strategy* for player $p \in \{\text{Even}, \text{Odd}\}$ which is simply a strategy that guarantees a win for p. It turns out that parity games are **determined** [EM79, EJ91] – for every parity game starting at some particular vertex there is a (pure) winning strategy for one of the players.

2.2 Solving parity games

We can distinguish different variants of solving parity games. First, one can talk about the **coverage** of the solver. We say that a solver is *local* if we are only interested in finding the solution for a particular vertex $v \in V$, or *global* if the intention is to find the solution from every possible vertex in the game.

But what does it exactly mean to find the solution? This leads us to the notion of **solution quality**. We say that the solver *decides the winner* if the solution only gives us the winner for the nodes of interest. If the solution also returns a winning strategy, we say that the solver performs a *strategy synthesis* [JL17].

		Coverage	
		local	global
Solution quality	without strategy	Deciding the winner	Finding the winning positions
	with strategy	Local strategy synthesis	Strategy synthesis

Table 1: Different definitions of solving parity games.

It is not difficult to see that local and global solvers are interreducible – a global solver can be simulated by a local one by calling it at most $|V|$ times and a local solver can be simulated by a global one by solving the game globally and then returning the solution for the vertex we are interested in. Such constructions are feasible although rarely used in practice [FL09b].

In section 2.1.1, we have seen that parity games are determined, which means that from any starting vertex exactly one of the players has a winning strategy. Remembering that we defined a strategy for Even in the previous section as a map from S_e to V, we see that there are infinitely many finite plays that we have to assign a strategy to. So at first, it may seem a bit daunting to think about how to approach solving such a game. Luckily, we do not have to consider all the possible infinite plays to give a winning strategy for the winner. Parity games are characterised by what we call **memoryless determinacy** [EJ91] – if a player has a winning strategy from a node v, then that player also has a winning strategy from v *which does not depend on the history of play*[2] but only on v. More formally

Definition 2.3. *A strategy σ is **memoryless** if for any two finite plays $\nu = (v_0, v_1, \ldots, v_n)$ and $\omega = (w_0, w_1, \ldots, w_m)$, if $v_n = w_m$, then $\sigma(\nu) = \sigma(\omega)$.*

The fact that parity games are memorylessly determined is what allows us to seek for efficient algorithms for solving them. As we will see, parity games have been thoroughly analysed and many different algorithms for solving them are available.

2.2.1 History

One could argue that the first proof of memoryless determinacy of parity games appeared in 1979 in [EM79], where Ehrenfeucht and Mycielski proved that mean-payoff games are memorylessly determined. Nevertheless, the first direct proof of memoryless determinacy of parity games appeared in 1991 in [EJ91, Corollary 4.3]. Two years later, it was shown

[2] Such a strategy is called *memoryless* or *positional*.

that the model checking problem for μ-calculus is in NP $\cap$ CO-NP [EJS93, Theorem 4.3]. Since μ-calculus model checking is PTIME reducible to solving parity games [EJS93,Jur98], it follows that deciding the winner of a parity game is also in NP $\cap$ CO-NP[3]. After those results were published, other algorithms for solving parity games started appearing in the literature.

Zielonka's recursive algorithm

One of the first and most notable algorithms is the *recursive* algorithm due to Zielonka [Zie98]. The idea comes from Zielonka's proof of determinacy of parity games and is based on the notion of attractors.

This approach is notable not only because it is one of the first, but also because of its great performance on most instances of parity games. Even though the worst-case runtime of this algorithm is exponential [FL09b], it is known to perform better than any other solvers on average [FL09a, vD18].

Small progress measures

In the year 2000, Jurdziński published another approach for solving parity games using so-called *progress measures* [Jur00]. A progress measure is an assignment of values to vertices that has to satisfy specific properties. It can be shown that if such a mapping is found, then it automatically induces a strategy and the solution for the game.

The algorithm proposed by Jurdziński also yields exponential worst-case runtime [Jur00] and, as experimental data shows, this approach tends to perform worse than the recursive algorithm.

In section 3.1 we describe progress measures in depth, prove their necessity and sufficiency for solving parity games and then describe the algorithm for finding them.

Other solvers

Over the years different algorithms and approaches appeared in the literature. Some of them use the idea of strategy improvement, based on the strategy improvement algorithm for stochastic games by Hoffman and Karp [HK66]. These come in many variations including probabilistic strategy improvement [BSGV04], local strategy improvement [FL10] or the so-called optimal strategy improvement [Sch08].

Apart from the strategy improvement-based algorithms, there are various others such as the priority promotion algorithm [BDM16] or, for local solving, the algorithm by Stevens and Stirling [SS98].

[3] It is in fact in UP $\cap$ CO-UP [Jur98].

Quasi-polynomial breakthrough

In 2017 parity game solving experienced a breakthrough with the appearance of a quasi-polynomial time algorithm for solving parity games by Calude et al. [CJK+17]. This was followed up by Jurdziński and Lazić with their *succinct progress measure* algorithm [JL17] that is strongly rooted in the ideas from [Jur00] and inspired by the work of Calude.

A year later, another quasi-polynomial time algorithm came to light in the paper by Lehtinen [Leh18] which introduces a different approach referred to as a *register game*.

What is interesting is that still in 2018, Czerwiński et al. [CDF+18] published a paper in which they argue that all of the quasi-polynomial time algorithms presented here can be viewed as instances of the same general approach for solving parity games – so-called *separation approach*.

To this day, the idea of a polynomial time algorithm for solving parity games remains elusive.

2.2.2 Parity game solvers

Luckily, rich theory behind parity games isn't just theory. Online one can find various implementations of the most popular algorithms described in section 2.2.1. In here, we mention two robust parity game solvers – PGSolver and Oink.

PGSolver

PGSolver is a "collection of tools for generating, manipulating and – most of all – solving parity games" [FL09b]. Written in OCaml, it was created in 2009 by Oliver Friedmann and Martin Lange with the mission of facilitating the comparison of algorithms that have appeared in the literature over the years.

PGSolver provides many tools, including game generators that allow creating different parity games ranging from random games, through encodings of problems as parity games (e.g. `towersofhanoi`), to games difficult for particular solvers, e.g `jurdzinskigame` or `recursiveladder`. Apart from the generators, the library includes parity game transformers, obfuscators and a benchmarking tool that allows users to compare multiple algorithms on the same input.

Arguably the most important part of the library – the solver itself – comes fully featured too. Users can choose whether to allow for global optimisation that attempts to pre-process a game by means of priority compression, self-loop removal and other tricks that are believed to speed up the solving process in most instances [FL09b]. Apart from that, there are options to choose between global and local solvers as well as developer tools such as debug messages or solution verification.

Our implementations described in this document are incorporated into the PGSolver library (see appendix D) and make use of a lot of its functionality. To learn more about the features provided by PGSolver refer to [FL09b].

Oink

Oink is a new tool written in C++ that "aims to provide a high-performance implementation of parity game solvers" [vD18]. Oink was created in 2018 by Tom van Dijk and implements a narrow range of carefully chosen algorithms with the focus on optimising performance.

It is worth noting that van Dijk in [vD18] also conducts a comparative study of the succinct progress measures algorithm. Nevertheless, our work is completely independent of the work of van Dijk.

2.3 Importance

Parity games are being studied because of their theoretical and practical significance. A few main motivations for studying parity games mentioned in the literature include the following:

- Modal μ-calculus is considered to be "one of the most important logics in model-checking" [BW18] and we know that it is polynomial-time equivalent to solving parity games [EJS93].
- Parity games are closely related to other infinite duration games including mean payoff games [Jur98] or stochastic games [Pur14].
- Controller synthesis problems require solving parity games [FL09b].
- Parity games have a very interesting complexity theory status – they are one of a few naturally arising problems known to be in NP $\cap$ co-NP not known to be in P [Jur98].

3 Progress measures

Progress measures first appeared as a method for solving parity games in a paper by Jurdziński [Jur00]. In a sense, one could think of a progress measure as an indicator of even and odd cycles – cycles with the highest priority being even or odd respectively. Progress measures have some interesting properties that allow us to prove their existence is both sufficient and necessary for the existence of winning strategies in a game [Jur00], allowing us to solve the game by finding appropriate progress measures. The same holds for *succinct* progress measures which are additionally easier to find[4].

We start by defining small progress measures. We give a few examples of how they work and present an algorithmic way of finding them using fixed-point iteration. Then we present succinct progress measures with all their machinery and discuss the asymptotic speedup which comes with using those instead of small progress measures. At the end of the section, we present the implementation of the succinct progress measures algorithm in OCaml for the PGSolver library.

3.1 Small progress measures

Notation. We denote by $[n]$ the set $\{0, 1, ..., n-1\}$. We also define d to be the smallest even number at least as big as any priority in the game. Given a tuple A^k for some ordered set A, we define the ordering $>$ of the tuples from A^k to be lexicographic. We also enforce special indexing on tuples. Let $t \in A^k$. Then we write t as

$$t = (t_{2k-1}, t_{2k-3}, \ldots, t_3, t_1),$$

in particular, if $t \in \mathbb{N}^{d/2}$, then

$$t = (t_{d-1}, t_{d-3}, \ldots, t_3, t_1).$$

We also define a *p-truncation* of a tuple $t \in A^k$ to be t without all the t_i such that $i < p$ and denote it as $t|_p$. By () we denote an empty tuple.

Example. To demystify the ideas of lexicographic ordering and truncations, consider two tuples $(3, 2, 1)$ and $(3, 2, 0)$. With the lexicographic ordering we have $(3, 2, 1) > (3, 2, 0)$ because the first two entries from the left are the same and in the third entry $1 > 0$. As examples of truncations consider

[4] At least complexity-theoretically, see section 5.

$$
\begin{aligned}
(3,2,1)|_0 &= (3,2,1)\\
(3,2,1)|_1 &= (3,2,1)\\
(3,2,1)|_2 &= (3,2)\\
(3,2,1)|_3 &= (3,2)\\
(3,2,1)|_4 &= (3)\\
(3,2,1)|_5 &= (3)\\
(3,2,1)|_6 &= ().
\end{aligned}
$$

$\triangle$

3.1.1 Idea

Jurdziński's algorithm hinges on the idea of edge progressiveness. In the following description, it may help to think of the parity game as if we were Even and Odd was an opponent. A progressive edge is in a sense 'good for Even', meaning that Even wants to take such an edge. And naturally, if such an edge is 'good for Even', it is also 'bad for Odd'. Thus in a sense, we want to make sure that Odd has no choice other than taking a progressive edge. The progress measure then tries to find an assignment of integer tuples to vertices such that each node satisfies the progressiveness criteria.

In the following discussion $\mu : V \mapsto \mathbb{N}^{d/2}$ will be a map and one can think of it as a 'progress measure candidate'.

Definition 3.1 (Progressiveness)**.** *Let $G = (V, V_e, V_o, E, \pi)$ be a parity game and $\mu : V \mapsto \mathbb{N}^{d/2}$. An edge $(v, w) \in E$ is said to be **progressive** in μ if*

$$\mu(v)|_{\pi(v)} \geq \mu(w)|_{\pi(v)}$$

*and the inequality is **strict** if $\pi(v)$ is odd. A **node** $v \in V$ is said to be progressive in μ if the following holds depending on the owner of v:*

- *if $v \in V_e$, then **there is** a successor $w \in V$ of v such that (v, w) is progressive in μ,*
- *if $v \in V_o$, then **for all** successors $w \in V$ of v, the edge (v, w) is progressive in μ.*

This allows us to present the first definition of a progress measure.

Definition 3.2. *Let $G = (V, V_e, V_o, E, \pi)$ be a parity game. A **parity progress measure** $\mu : V \mapsto \mathbb{N}^{d/2}$ is a map such that each $v \in V$ is progressive in μ.*

We now consider two examples. In the first one, we find a parity progress measure for the graph and in the second we answer the question of whether a parity progress measure always exists.

Example. Consider the parity game depicted in figure 3.

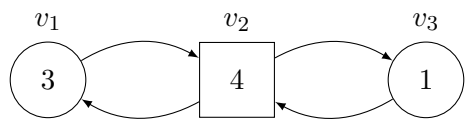

Figure 3: A parity game example. Square node belongs to Even, circle nodes to Odd.

As the highest priority occurring in the game is 4, we know that $d = 4$ (smallest even number not smaller than 4), so the parity progress measure μ will map vertices to $\frac{d}{2} = 2$-tuples. Let

$$\begin{aligned}\mu(v_1) &= (a_3, a_1)\\ \mu(v_2) &= (b_3, b_1)\\ \mu(v_3) &= (c_3, c_1),\end{aligned}$$

where $a_i, b_i, c_i \in \mathbb{N}$. If we want μ to satisfy the parity progress measure definition 3.2, we need to make each of v_1, v_2 and v_3 progressive. Each vertex belonging to Odd has only one outgoing edge and that edge has to be progressive. This yields the following inequalities

$$\begin{aligned}(a_3, a_1)|_3 &> (b_3, b_1)|_3 \qquad \text{(progressiveness of the edge } (v_1, v_2))\\ (c_3, c_1)|_1 &> (b_3, b_1)|_1 \qquad \text{(progressiveness of the edge } (v_3, v_2)).\end{aligned}$$

The vertex belonging to Even – v_2 – has two edges and it's enough if we make one of them progressive. So μ has to satisfy either

$$(b_3, b_1)|_4 \geq (a_3, a_1)|_4 \qquad \text{(progressiveness of the edge } (v_2, v_1))$$

or

$$(b_3, b_1)|_4 \geq (c_3, c_1)|_4 \qquad \text{(progressiveness of the edge } (v_2, v_3)).$$

All four inequalities above boil down to the system

$$\begin{aligned}a_3 &> b_3\\ (c_3, c_1) &> (b_3, b_1)\\ () &\geq ()\\ () &\geq ()\end{aligned}$$

which means that we require $a_3 > b_3$ and in the tuple (c_3, c_1) we can either choose $c_3 > b_3$ or $c_3 = b_3$ and $c_1 > b_1$ to satisfy the second constraint. Clearly, the third and fourth

constraints are satisfied, which means that v_2 is always progressive in this game.

Remembering that we are looking for non-negative integers, we can now set $a_3 = c_3 = 1$ and $a_1 = b_3 = b_1 = c_1 = 0$. Indeed, an example of a progress measure for this game would be

$$\begin{aligned}\mu(v_1) &= \mu(v_3) = (1, 0)\\ \mu(v_2) &= (0, 0).\end{aligned}$$

Alternatively, we could also have

$$\begin{aligned}\mu(v_1) &= (1, 0)\\ \mu(v_2) &= (0, 0)\\ \mu(v_3) &= (0, 1).\end{aligned}$$

△

The next example shows that a parity progress measure may not exist. Notice that the game in figure 3 is winning for Even from each vertex, whereas the one we are going to see in the next example (figure 4) is winning for Odd. This is not a coincidence – it can be shown that a parity progress measure exists if and only if the game is winning for Even from every vertex.

Example. As an example, consider the game in figure 4. In this game, there is just one possible move from both nodes and clearly Odd wins any play as both 1 and 0 will occur infinitely often and $1 > 0$.

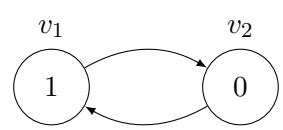

Figure 4: A parity game always won by Odd. Both nodes belong to Odd.

What happens if we try to find a parity progress measure for this example? Let μ be this parity progress measure. We know that $d = 2$ (smallest even number not smaller than top priority), so the co-domain of μ consists of 1-tuples. Let $\mu(v_1) = (a_1)$, $\mu(v_2) = (b_1)$. To make both nodes progressive, we need to find non-negative values a_1, b_1 such that

$$\begin{aligned}&(a_1)|_1 > (b_1)|_1 && \text{(progressiveness of the edge } (v_1, v_2))\\ &(b_1)|_0 \geq (a_1)|_0 && \text{(progressiveness of the edge } (v_2, v_1)).\end{aligned}$$

This can be rewritten as

$$a_1 > b_1$$
$$b_1 \geq a_1$$

Clearly, those two inequalities cannot hold at the same time. Therefore a parity progress measure for this game cannot exist. $\triangle$

How can one find such a progress measure algorithmically if it exists? It can be shown [Jur00, p. 7] that if we start with $\mu(v) = (0, \ldots, 0)$ for all $v \in V$ and iteratively increase the value of μ for the non-progressive nodes, we will always eventually reach a fixed point in which all the nodes are progressive[5]. Moreover, if we find such a parity progress measure μ, then we know that Even wins the game from every vertex [Jur00, Proposition 4].

However, as we have seen in the second example above, this doesn't help if the parity progress measure does not exist – in that case, the iterative process would never terminate. This observation incentivises to look for a generalised measure that will exist for *all* games. To that end, we will require some *upper bound* on the elements that μ maps to.

Additionally, we want to indicate somehow that if the upper bound is reached for some $v \in V$, we do not consider v as non-progressive anymore. Introducing an additional *top element* $\top$ to the co-domain of μ will help us achieve that.

Notation. We denote by V_i the set of all vertices of priority i, $V_i = \pi^{-1}(i)$ and let $n_i = |V_i|$ be the number of vertices of priority i. Then we define M_G as the following subset of $\mathbb{N}^{d/2}$

$$M_G = [n_{d-1} + 1] \times [n_{d-3} + 1] \times \ldots \times [n_1 + 1].$$

By $M_G^\top$ we denote the set $M_G \cup \{\top\}$ such that $\top > m$ for all $m \in M_G$. Moreover

$$\top|_i = \top \quad \forall i.$$

Now we can revisit definitions 3.1 and 3.2 and generalise them to work for any parity game and consequently allow us to find the winning sets for both players.

Definition 3.3 (Edge progressiveness)**.** *Let $G = (V, V_e, V_o, E, \pi)$ be a parity game and $\mu : V \mapsto M_G^\top$. Progressiveness of an edge $(v, w) \in E$ is defined as in definition 3.1 except for the case when $\mu(v) = \mu(w) = \top$. In that case, the edge is progressive regardless of the priority of v.*

The definition of node progressiveness does not need any adjustments. We can now define a generalised version of 3.2 – the *game parity progress measure* [Jur00, Definition 6].

Definition 3.4 (Game parity progress measure)**.** *Let $G = (V, V_e, V_o, E, \pi)$ be a parity game. A **game parity progress measure** is a map $\mu : V \mapsto M_G^\top$ such that every vertex $v \in V$ is progressive in μ.*

[5] *Only* if the parity progress measure exists!

From now on, for brevity, we will write GPPM when referring to a game parity progress measure.

Let's come back to the main problem we are concerned with – solving parity games. We want GPPMs to help us with that. And it can be shown that a 'special' GPPM that allows us to solve the game exists. However not every GPPM for a parity game is special.

First, notice that there always is a trivial GPPM μ for a parity game $G = (V, V_e, V_o, E, \pi)$ such that $\mu(v) = \top$ for all $v \in V$. This way every edge – and consequently every node – is progressive, satisfying definition 3.4. Unfortunately, in most cases, this is not the special GPPM we are looking for. To define the special GPPM, we need to impose an *order* on maps $\mu : V \mapsto M_G^\top$. With this order, the *least* GPPM will be the special one.

Definition 3.5. *We define a* ***partial order*** *on maps $V \mapsto M_G^\top$ as follows. Let μ, μ' be such maps. Then we write*

$$\mu \leq \mu' \iff \mu(v) \leq \mu'(v) \quad \forall v \in V.$$

Additionally, we write $\mu < \mu'$ if $\mu \leq \mu'$ and $\mu \neq \mu'$.

We can now say that the *least GPPM* is a map $\mu : V \mapsto M_G^\top$ such that μ is a GPPM and if $\mu' : V \mapsto M_G^\top$ is also a GPPM then $\mu \leq \mu'$. It can be shown that order 3.5 creates a *complete lattice* structure on the set of maps $V \mapsto M_G^\top$ [Jur00, p. 7]. Therefore, by Knaster-Tarski theorem, we know that *the least GPPM exists*. The theorem also gives us a simple way to compute this GPPM. We can use the same idea we have introduced before – start with a map that maps each vertex to an all-zero tuple and iteratively increase it until all nodes become progressive. By Knaster-Tarski theorem, this is guaranteed to stop at the least GPPM.

But why is the least GPPM special in the first place? Well, it can be shown that finding it is equivalent to finding the winning sets for both players and the strategy for Even [Jur00, Corollaries 7 and 8]:

Theorem 3.6. *The following two properties of GPPMs hold*

- *If $G = (V, V_e, V_o, E, \pi)$ is a parity game and $\mu : V \mapsto M_G^\top$ is the least GPPM, then the winning set of Even in G is*

  $$W_e = \|\mu\| = \{v \in V \mid \mu(v) \neq \top\}$$

  and the winning strategy $\tilde{\mu}$ for Even is to pick an edge progressive in μ from each vertex in $W_e \cap V_e$

  $$\tilde{\mu}(v) = \arg\min_{w \in V} \{\mu(w) \mid (v, w) \in E\} \quad \forall v \in W_e \cap V_e.$$

- *If $W_e \subseteq V$ is the set of winning vertices of Even in a parity game $G = (V, V_e, V_o, E, \pi)$, then there exists a GPPM μ (e.g. the least GPPM) such that*

$$\begin{aligned} \mu(v) &< \top \quad \forall v \in W_e, \\ \mu(v) &= \top \quad \forall v \in W_o = V \setminus W_e. \end{aligned}$$

3.1.2 Algorithm

In the discussion above we introduced GPPMs and showed that the least GPPM can be used to solve a parity game. We have also given an idea of how to find such a mapping – start with the smallest possible assignment of tuples to vertices and then iteratively make nodes progressive until all of them are progressive.

In this section, we formalise this idea. We start by introducing the lift and Lift operators that are the main building blocks of the algorithm. Then we present the pseudocode and analyse the performance of this method in the worst case.

Notation. Let μ be a mapping. By $\mu[v \to m]$ we denote a new mapping that behaves the same as μ for all elements apart from v and maps v to m. Formally

$$\mu[v \to m](u) = \begin{cases} \mu(u) & \text{if } u \neq v \\ m & \text{if } u = v \end{cases} \qquad \text{for all } u \text{ in the domain of } \mu.$$

Let's start by introducing the $\text{lift}(\mu, v, w)$ operator which given a mapping $\mu : V \mapsto M_G^\top$ (a small parity progress measure *candidate*) and an edge $(v, w) \in E$ returns the least $m \in M_G^\top$, $m \geq \mu(v)$ such that the edge (v, w) becomes progressive in $\mu[v \to m]$.

The lift operator is a tool for making an *edge* progressive. However, in definition 3.4 it's *node* progressiveness that we want to satisfy, so let's introduce another operator, Lift, that makes a node progressive.

Definition 3.7. *Let $G = (V, V_e, V_o, E, \pi)$ be a parity game and $\mu : V \mapsto M_G^\top$ a GPPM candidate. The* Lift *operator is a tool for making a node in μ progressive and is defined with the help of* lift *as follows*

$$\text{Lift}_v(\mu)(u) = \begin{cases} \mu(u) & \text{if } u \neq v \\ \min_{(v,w)\in E} \text{lift}(\mu, v, w) & \text{if } u = v \text{ and Even owns } v \\ \max_{(v,w)\in E} \text{lift}(\mu, v, w) & \text{if } u = v \text{ and Odd owns } v. \end{cases}$$

This should be quite intuitive – since the idea behind Lift is to make the node v progressive, we want to keep the mapping as it is for all nodes $u \neq v$ and focus on the value that v maps to. Remembering definition 3.1 of a progressive node, for Even, we want to find the minimum $m \in M_G^\top$ required to make *some* edge progressive, which is the minimum of all the lift operators, whereas for Odd we want to find the minimum $m \in M_G^\top$ required to make *all* edges progressive – and that is the highest of the lift operators for all

outgoing edges from v, because anything smaller will cause some of the edges to not be progressive.

Using these two operators we can now present the algorithm for finding the least GPPM. As mentioned before, the correctness of the algorithm comes from the fact that the order on the maps $V \mapsto M_G^\top$ creates a complete lattice. This, combined with the fact that the Lift operator is $<$-monotone [Jur00, Proposition 9], proves that the algorithm is correct. For the details refer to [Jur00, p. 7] and [JL17, Theorem 5].

Algorithm 1 Progress measure lifting algorithm

1: Initialize $\mu : V \mapsto M_G^\top$ to map each $v \in V$ to $(0, \ldots, 0)$
2: **while** There exists a $v \in V$ not progressive in μ **do**
3: $\quad \mu \leftarrow \mathrm{Lift}_v(\mu)$ ▷ make v progressive
4: **end while**
5: **return** $\|\mu\|$ as the winning set of Even, $\tilde{\mu}$ as the strategy for Even ▷ see theorem 3.6

Note that line 3 of the algorithm makes a non-progressive node v progressive but potentially also makes some of the predecessors of v non-progressive[6]. If a predecessor u wasn't progressive before, then notice that $\mathrm{Lift}_v(\mu)(v) \geq \mu(v)$, so u cannot become progressive. However, if u was progressive, then depending on the owner of u we have two cases to consider:

- If $u \in V_e$, then u becomes non-progressive if and only if there is **no** progressive edge from u, i.e. if (u, v) was the only progressive edge from u in μ and is no longer progressive in $\mathrm{Lift}_v(\mu)$.
- If $u \in V_o$, then u becomes non-progressive if and only if (u, v) becomes non-progressive.

This will become important later when we talk about the implementation of the succinct progress measures algorithm (section 3.3), because it allows us to only focus on the subset of V during each Lift operation.

Algorithm analysis

Algorithm 1 runs in time exponential in the number of priorities. If $G = (V, V_e, V_o, E, \pi)$ is a parity game and we let $n = |V|$, $m = |E|$ and d be defined as earlier (smallest even number not smaller than any priority in the game), then the runtime of the algorithm is [Jur00, Theorem 11]

$$O\left(dm \cdot \left(\frac{n}{\lfloor d/2 \rfloor}\right)^{\lfloor d/2 \rfloor}\right). \tag{1}$$

[6] We don't prove it here, but it's not difficult to see that *only* the predecessors can become non-progressive.

The biggest bottleneck of the algorithm is the number of times we need to lift, i.e. the size of $M_G^\top$. Recalling the definition of M_G from section 3.1.1, its size can be written as

$$|M_G| = \prod_{i=1}^{\lfloor d/2 \rfloor} (n_{2i-1} + 1).$$

Even though we are not going to prove that this is true, it's not difficult to see that this number is the highest when all n_{2i-1} are equal. We can also assume $n_i \neq 0$ for each $i \in [d]$, because if some priority i doesn't exist, then all priorities bigger than i can be scaled down by two[7]. Therefore $n = \sum_{i=0}^{d} n_i = \sum_{i=1}^{d/2}(n_{2i-1} + n_{2i-2}) + n_d \geq \sum_{i=1}^{\lfloor d/2 \rfloor}(n_{2i-1} + 1)$ and thus if we want all n_{2i-1} to be equal then $n_{2i-1} + 1 \leq \frac{n}{\lfloor d/2 \rfloor}$, so

$$\prod_{i=1}^{\lfloor d/2 \rfloor} (n_{2i-1} + 1) \leq \prod_{i=1}^{\lfloor d/2 \rfloor} \left(\frac{n}{\lfloor d/2 \rfloor} \right) = \left(\frac{n}{\lfloor d/2 \rfloor} \right)^{\lfloor d/2 \rfloor}$$

as desired. The runtime bound (1) comes from the fact that the Lift operator can be implemented to run in time $O(d \cdot \text{out-degree}(v))$ for each $v \in V$.

For a game in which d is constant, this is a polynomial time algorithm, however in most games of interest d is not a constant and it's not too difficult to show that priorities can only be compressed without changing the properties of the game to make d at most linear in n.

For a more elaborate analysis of the runtime refer to [Jur00] which gives a slightly better bound and suggests some optimisations that can improve the runtime even further. It also introduces a special type of parity games which induces the worst-case runtime of the lifting algorithm 1, regardless of the details of the lifting policies.

3.2 Succinct progress measures

Notation. In this section, the value d, tuple indexing, p-truncations and progressiveness are defined as in section 3.1. By η we denote the number of vertices with odd priority in a parity game. We denote an empty binary string by ϵ.

A breakthrough happened in 2017 when Calude et al. [CJK+17] published a paper that gave a quasi-polynomial time algorithm for solving parity games. Unfortunately, the algorithm also requires quasi-polynomial space which makes it impractical. However, inspired by the ideas from that paper, Jurdziński and Lazić came up with an improved version of progress measures [JL17] that guarantee quasi-polynomial worst-case runtime with only quasi-linear space requirement, creating a potential competitor for other algorithms.

This approach very closely resembles the one presented in section 3.1, the only change being the structure of the co-domain μ – succinct progress measures use *tuples of binary*

[7] This is one of the things that PGSolver does during priority compression [FL09b, Section 2.4.3].

strings as the co-domain. We start by presenting the structure those and give the definition of a succinct progress measure. Then we show that, similarly to GPPMs, succinct progress measures are also sufficient and necessary for solving parity games. Everything is accompanied by multiple examples.

At the end of the section, we present an algorithm for finding the least succinct progress measure using fixed-point interation, an example of how the algorithm works in practice and discuss its time and space complexity.

3.2.1 Idea

Succinct progress measures rely on binary strings, which serve the same role as natural numbers in GPPMs (3.4). We start by defining a special ordering of binary strings.

Definition 3.8. *Let s and s' be binary strings $s, s' \in \{0,1\}^*$ and b a binary digit $b \in \{0,1\}$. We define the ordering $<$ of binary strings as follows*

$$0s < \epsilon, \quad \epsilon < 1s \quad \text{and} \quad bs < bs' \iff s < s'.$$

Similarly to GPPMs, succinct progress measures map each vertex to a tuple, called an *adaptive counter*, that has certain restrictions on size and the total length of binary strings inside it.

Definition 3.9 (Adaptive counter)**.** *Let $g, h \in \mathbb{N}$. A **g-bounded adaptive h-counter** is an h-tuple of binary strings whose total length is at most g. Adaptive counters are ordered **lexicographically** as extension of the order from definition 3.8.*

By $B_{g,h}$ we denote the set of all g-bounded adaptive h-counters and following the notation from [JL17], by $S_{\eta,d}$ we denote the set of $\lceil \lg \eta \rceil$-bounded adaptive i-counters for $0 \leq i \leq d/2$:

$$S_{\eta,d} = \bigcup_{i=0}^{d/2} B_{\lceil \lg \eta \rceil, i}.$$

Example. $B_{1,2}$, the set of all 2-tuples of binary strings whose total length is at most 1 is

$$B_{1,2} = \{(0,\epsilon), (\epsilon,\epsilon), (1,\epsilon), (\epsilon,0), (\epsilon,1)\}.$$

and the set of all lg(2)-bounded adaptive counters of length at most 2 is

$$S_{2,4} = \bigcup_{i=0}^{2} B_{\lceil \lg 2 \rceil, i} = \{(), (0), (\epsilon), (1), (0,\epsilon), (\epsilon,\epsilon), (1,\epsilon), (\epsilon,0), (\epsilon,1)\}$$

and is ordered as follows

$$() < (0) < (0,\epsilon) < (\epsilon) < (\epsilon,0) < (\epsilon,\epsilon) < (\epsilon,1) < (1) < (1,\epsilon).$$

△

This allows us to define a *succinct progress measure*. As one may guess, it maps vertices to the elements of $S_{\eta,d}$ with an additional $\top$ element that, similarly to GPPMs, indicates that we no longer want a node to be considered as non-progressive.

Definition 3.10. *Let $G = (V, V_e, V_o, E, \pi)$ be a parity game. A* ***succinct progress measure*** *μ is a map from V to $S^{\top}_{\eta,d} = S_{\eta,d} \cup \{\top\}$ such that each node $v \in V$ is progressive in μ, that is*

- *If $v \in V_e$, then* ***there exists*** *a successor w of v such that*

$$\mu(v)|_{\pi(v)} \geq \mu(w)|_{\pi(v)}$$

  and the inequality is strict if $\pi(v)$ is odd, $\mu(v) < \top$ and $\mu(w) < \top$.

- *If $v \in V_o$, then* ***for all*** *successors w of v*

$$\mu(v)|_{\pi(v)} \geq \mu(w)|_{\pi(v)}$$

  and the inequality is strict if $\pi(v)$ is odd, $\mu(v) < \top$ and $\mu(w) < \top$.

Example. Consider the game presented in figure 5. We deduce that $d = 4$, $d/2 = 2$ and $\eta = 2$, $\lceil \lg \eta \rceil = 1$. Therefore we are looking for 1-bounded adaptive i-counters with $0 \leq i \leq 2$, so let $\mu : V \mapsto S^{\top}_{2,4}$ be a succinct progress measure.

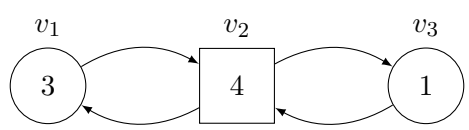

Figure 5: A parity game example. Circle nodes belong to Odd and square to Even.

Notice that $\mu(v_1)|_4 = \mu(v_2)|_4 = \mu(v_3)|_4 = ()$ regardless of what the actual mapping μ is and hence v_2 will be progressive regardless of what value we assign to $\mu(v_2)$. Let's assign the smallest one possible, i.e.

$$\mu(v_2) = ().$$

Now, to make (v_1, v_2) progressive, we need to satisfy the inequality

$$\mu(v_1)|_3 > \mu(v_2)|_3 = ().$$

This means that $\mu(v_1)$ has to be of length at least 1. Let $\mu(v_1) = (0)$. Similarly to make (v_3, v_2) progressive, we need to satisfy the inequality

$$\mu(v_3)|_1 = \mu(v_3) > \mu(v_2)|_1 = (),$$

so again, any $\mu(v_3)$ of length at least 1 will do. Let $\mu(v_3) = (0)$. This gives us the following succinct progress measure

$$\mu(v_1) = (0)$$
$$\mu(v_2) = ()$$
$$\mu(v_3) = (0).$$

△

Recall the definition of partial GPPM ordering 3.5 and let's consider the same ordering for succinct progress measures (SPMs). It can be shown with a bit of ingenuity [JL17, Sections 3.4 and 3.5] that SPMs have the same properties as GPPMs and we can use them to solve parity games. The proof relies on the fact that GPPMs exist and applies the **succinct tree coding lemma** (appendix A) to show that one can work with SPMs instead of GPPMs to solve parity games.

Theorem 3.11. *The following two properties of SPMs hold*

- *If $G = (V, V_e, V_o, E, \pi)$ is a parity game and $\mu : V \mapsto S^{\top}_{\eta,d}$ is the least SPM, then the winning set of Even in G is*

$$W_e = \|\mu\| = \{v \in V \mid \mu(v) \neq \top\}$$

  and the winning strategy $\tilde{\mu}$ for Even is to pick an edge progressive in μ from each vertex in $W_e \cap V_e$

$$\tilde{\mu}(v) = \arg\min_{w \in V}\{\mu(w) \mid (v, w) \in E\} \quad \forall v \in W_e \cap V_e.$$

- *If $W_e \subseteq V$ is the set of winning vertices of Even in a parity game $G = (V, V_e, V_o, E, \pi)$, then there exists an SPM μ (e.g. the least SPM) such that*

$$\mu(v) < \top \quad \forall v \in W_e,$$
$$\mu(v) = \top \quad \forall v \in W_o = V \setminus W_e.$$

3.2.2 Algorithm

An algorithm very similar to the one we have seen in section 3.1.2 works for finding the least SPM – start with the smallest possible assignment of adaptive counters to vertices and apply the lifting procedure. By Knaster-Tarski theorem this is guaranteed to reach a fixed-point which is the least SPM we are after.

For completeness, let's remind ourselves of the Lift and lift operators. Let $G = (V, V_e, V_o, E, \pi)$ be a parity game and $\mu : V \mapsto S^{\top}_{\eta,d}$. The $\text{lift}(\mu, v, w)$ operator returns the smallest $m \in S^{\top}_{\eta,d}$ such that $m \geq \mu(v)$ and (v, w) is progressive in $\mu[v \to m]$. The rules for finding $\text{lift}(\mu, v, w)$ for SPMs are not obvious and can be found in appendix B.

The Lift operator makes a node progressive in a similar manner as for GPPMs.

$$\mathrm{Lift}_v(\mu)(u) = \begin{cases} \mu(u) & \text{if } u \neq v \\ \min_{(v,w)\in E} \mathrm{lift}(\mu, v, w) & \text{if } u = v \text{ and Even owns } v \\ \max_{(v,w)\in E} \mathrm{lift}(\mu, v, w) & \text{if } u = v \text{ and Odd owns } v. \end{cases}$$

With this, we are now ready to present the succinct progress measure lifting algorithm which is the point of interest in the implementation section 3.3.

Algorithm 2 Succinct progress measure lifting algorithm

```
Initialize μ : V ↦ S^⊤_{η,d} to map each v ∈ V to ()
while There exists a v ∈ V not progressive in μ do
    μ ← Lift_v(μ)                                   ▷ make v progressive
end while
return ‖μ‖ as the winning set of Even, μ̃ as the strategy for Even ▷ see theorem 3.11
```

Example. Consider the game G in figure 6. Let's run through the steps of lifting algorithm 2 for finding the succinct progress measure μ for G.

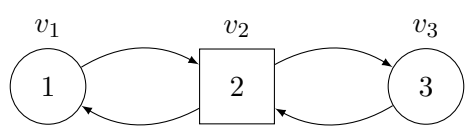

Figure 6: The parity game G. Circle nodes belong to Odd and square to Even.

We start by noting that $d = 4$ and $\eta = 2$, so we are looking for 1-bounded adaptive i-counters with $0 \leq i \leq 2$. The steps of the algorithm can be seen in table 2.

Update	Rule	$\mu(v_1)$	$\mu(v_2)$	$\mu(v_3)$	Non-progressive
Start	–	()	()	()	v_1, v_3
v_1	Rule 1	(0)	()	()	v_3
v_3	Rule 1	(0)	()	(0)	v_2
v_2	$\pi(v_2)$ even	(0)	(0)	(0)	v_1, v_3
v_1	Rule 1	$(0, \epsilon)$	(0)	(0)	v_3
v_3	Rule 3	$(0, \epsilon)$	(0)	(ϵ)	

Table 2: The steps of the lifting procedure 2. The 'Update' column indicates which node we are making progressive in the current step. The 'Rule' column indicates which rule from appendix B is used for the update.

The last row contains the least SPM. The strategy for Even in this game is to always go to v_1 from v_2. △

Algorithm analysis

The biggest bottleneck of the algorithm is, as before, the number of necessary lifts, i.e. $|S_{\eta,d}|$. The runtime analysis of algorithm 2 requires various mathematical tools including e.g. estimates for binomial coefficients. Curious readers are encouraged to see [JL17, Sections 4 and 5]. The main result proved there is that if $d = \omega(\log \eta)$, then

$$|S_{\eta,d}| = O\left(d\eta^{\lg(d/\lg \eta)+1.45}\right),$$

and consequently, the number of lifts required is not exponential as it was the case with GPPMs (3.1.2) but quasi-polynomial in the number of priorities.

To give a bound on the runtime of algorithm 2 itself, we note that each Lift operator can be implemented to work in time $O(\deg(v) \cdot \log \eta \cdot \log d)$ [JL17, Theorem 7] and consequently the runtime is bounded by

$$O\left(\sum_{v \in V} \deg(v) \cdot \log \eta \cdot \log d \cdot |S_{\eta,d}|\right) = O(m \log \eta \cdot \log d \cdot |S_{\eta,d}|).$$

It is also worth mentioning that space requirements are indeed quasi-linear – the only thing required to be stored is the map itself which assigns to each of $|V| = n$ vertices a $\lceil \lg \eta \rceil$-bounded adaptive i-counter with $0 \leq i \leq d/2$. By the succinct tree coding lemma (appendix A), each such adaptive counter can be represented using at most $\lceil \lg \eta \rceil \lceil \lg d \rceil$ bits, so the total space requirement is

$$O(n \log n \cdot \log d).$$

3.3 Design and implementation

In this subsection, we bring together everything mentioned so far in section 3 as a piece of OCaml code for the PGSolver library. We discuss the implementation of the succinct progress measure algorithm. For brevity, we sometimes just write 'progress measure', omitting 'succinct', but all mentions of progress measures in this section refer to succinct progress measures described in section 3.2.

We start with a quick introduction into OCaml which the algorithm is written in and a brief overview of the `Paritygame` module from the PGSolver library. We then move on to the implementation, where we present parts of the code and justify the decisions made.

Appendix D contains a small guide on how to find the implementation in the submitted folder.

3.3.1 Background

For clarity and ease of understanding the code in this section, we start with an introduction into OCaml and the `Paritygame` module from PGSolver.

OCaml

OCaml is a multi-paradigm programming language developed at Inria over the last 20 years [A+]. Programs written in OCaml can be purely functional as in Haskell, however OCaml also provides imperative and object-oriented features.

In comparison to languages such as C or C++, OCaml can be considered high-level with its automated garbage collection and no distinction between passing by value or reference – everything is passed by value unless we are considering mutable types. As we will also see when discussing the design, a sophisticated type system introduced in OCaml allows for a very easy representation of many structures.

The choice of OCaml for implementing SSPM can be justified by extensive library support in the form of PGSolver (see section 2.2.2). Having access to its features allowed focusing entirely on the intricacies of the algorithm – all the rest (parity game implementation, pre-processing etc.) was already taken care of.

All the interpreter examples in this section start with a `#`. Moreover, whenever the output of a call is displayed, we omit all the irrelevant information, such as types, for readability. Also, in favour of clarity, the fragments of OCaml code shown here do not contain the debug messages, however, the actual implementation does.

Last but not least, in many interpreter examples we use the **pipe** symbol, which in OCaml is `|>` and does something similar to the `|` used in Unix environments.

The `Paritygame` module

This is the most important fragment of the PGSolver library for the rest of this section. We give a short introduction of its capabilities.

There are 7 manipulable 'entities' in the module

- nodes,
- sets of nodes,
- parity games,
- solutions,
- strategies,
- players and
- priorities.

For each of those, there are multiple functions that allow to get, set or iterate the members of the given entity.

It is worth noting here that a few of the aforementioned types are implemented simply as *arrays*. Consequently, we can think of the set of vertices V of the game as $V = \{v_0, v_1, \ldots, v_{|V|}\}$, where the subscript represents the index in the array. And for example, both solution (`solution` type) and strategy (`strategy` type) are represented as arrays where the ith entry corresponds to the solution (strategy) for v_i.

Last but not least, the parity game itself (`paritygame` type) is also represented as an array that maps each node to a tuple composed of its priority, owner, predecessors, successors and an optional description.

3.3.2 Implementation

The algorithm attempts to mimic the structure presented by Jurdziński and Lazić in their algorithm described in section 3.2. We treat binary strings (`BString` module) as building blocks used to create adaptive counters (`AdaptiveCounter` module). The progress measure itself (`ProgressMeasure` module) is implemented as a map from nodes to their adaptive counters. The `ProgressMeasure` module is also where the bulk of logic resides with its lifting methods `lift` and `lift_`.

The whole structure is brought together in the module-free `solve` method that given a parity game builds a progress measure from the ground up. As mentioned in section 3.2, the algorithm proposed by Jurdziński and Lazić returns the nodes winning for Even and his strategy. We are, however, interested in *strategy synthesis for both players* 2.2, so first, the winning strategy for one player is computed and then, to compute the strategy of the other player we construct and solve a subgame from the original input.

3.3.2.1 `BString` module

Bit strings are the smallest building blocks of the whole system. Each adaptive counter is built as an array of those. While solving the game, bit strings are very frequently being manipulated during `lift` calls where we may want to either remove all elements from the back up to the first 0 (`cut` method) or append a 1 followed by zeros up to a certain length (`extend` method), so it is very important that these manipulations can be performed efficiently.

Each `BString` can have one of two types, as indicated in the line

```
type t = BString of bool list | Empty
```

The distinction between an empty bit string and a non-empty one is only made to facilitate testing. It would be totally possible (and readable) to work with `BString []` instead of `Empty`, but having the empty bit string as a separate type allows for easy conversion to a string representation as presented in the `show` method[8]:

[8]The empty binary string is actually presented as ϵ in the code, however the `minted` package for LaTeX does not allow for unicode characters in strings.

```
let show = function
| Empty -> "e"
| BString list -> [...]
```

As can be seen above, each bit is represented as a boolean instead of an integer. However, as we have seen in section 3.2, bit strings contain 0s and 1s and that's how we are going to refer to them here – 0 will represent the boolean value `false` and 1 the boolean value `true`, so a `BString` with content `[true;true;false;true]` will be written as 1101.

It is also worth mentioning that the initial implementation used arrays instead of lists to represent bit strings. However, as tests have shown, the difference in performance between the two implementations is unnoticeable and the list version is much clearer. Moreover, using immutable data structures has its irrefutable advantages – we have the guarantee that at no point the bit string we create can be modified as a side effect of another operation.

Manipulation

Apart from the functionality implemented in methods `length`, `create` and `is_empty`, which are for the most part wrappers around the usual list operations and the `is_max` method that checks whether given `BString` is maximal of its length (i.e. composed only of 1s), there are two main methods that allow us to manipulate bit strings.

First of those is `extend` which, given a `BString` `b` and an integer `n` creates a bit string of length `n` by appending $10\ldots0$ at the end of `b`, to the total length of `n`. A bit string of length not smaller than `n` is returned without any changes. This method is used in two of the cases of the `lift` method from the `ProgressMeasure` module – if the `AdaptiveCounter` is not full or if the last element of the `AdaptiveCounter` is composed only of 1s (rules 2 and 4 in appendix B). Below we can see an example call to `extend`:

```
# let b = BString.create [true; true; false];;
# BString.print b;;
110
# BString.extend b 6 |> BString.print;;
110100
```

The second one is the `cut` method that takes a `BString` `b` and returns a shortened version of it by removing, starting from the end, all the elements up to (and including) the first 0. For example:

```
# let b = BString.create [true; false; false; true; true; true];;
# BString.print b;;
100111
# BString.cut b |> BString.print;;
10
```

The `cut` method is also used in multiple cases of `lift` (rule 3 in appendix B).

Comparison

Adaptive counters need to be constantly compared while lifting – we need to find either the minimum (for player Even) and the maximum (for player Odd) lifted value of an edge. And what is compared underneath are the elements of adaptive counters – bit strings. That's why we need a way of comparing two `BString` variables following definition 3.8. Note that `compare` takes two bit strings `a` and `b` and returns an integer. This integer is positive (in this case 1) if `a > b`, 0 if `a = b` and negative (in this case -1) if `a < b`:

```
let compare bstr1 bstr2 =
match bstr1, bstr2 with
| Empty, Empty           -> 0
| BString list, Empty    -> if List.hd list then 1 else -1
| Empty, BString list    -> if List.hd list then -1 else 1
| BString l1, BString l2 ->
  let rec compare_rec l1 l2 =
    match l1, l2 with
    | [], []             -> 0
    | hd :: tl, []       -> if hd then  1 else -1
    | [], hd :: tl       -> if hd then  -1 else 1
    | h1 :: t1, h2 :: t2 ->
      if h1 && not h2 then 1
      else if not h1 && h2 then -1
      else compare_rec t1 t2
  in
  compare_rec l1 l2
```

The first three cases of the outer `match` statement correspond to the base case of the comparison, namely $0s < \epsilon$ and $\epsilon < 1s$, and the last case implements the comparison of non-empty bit strings recursively, with the same base cases as above. As we will see when discussing adaptive counters in section 3.3.2.2, this comparison is what the `AdaptiveCounter.compare` method relies on.

3.3.2.2 `AdaptiveCounter` module

Adaptive counters are crucial in the succinct progress measures algorithm. As we have seen in section 3.2, the number of possible values each adaptive counter can take is the determining factor of the runtime of the algorithm and so the manipulation of those should be as efficient as possible. Since `lift` requires access to both the beginning and end elements of the `AdaptiveCounter` and there are several cases in which in-place changes can be beneficial for performance, a list implementation did not seem like a great idea. Therefore each `AdaptiveCounter` can be of one of two types – either `Top` or a non-top element represented as an array of `BString`:

```
type t = ACounter of BString.t array | Top
```

Furthermore, as the value d of the smallest even integer at least as big as any priority is an intrinsic part of the representation of `AdaptiveCounter` in that it determines how trimming should be done, the *module itself* has a variable `d` that stores a reference of this value. Naturally, there is also a method to set the value of `d` at the beginning of the algorithm `set_d`.

Creation and basic helpers

We have a few helper functions, namely `empty`, `create`, `create_top`, to create appropriate adaptive counters and two functions for two different notions of the *length* of the adaptive counter. The first of those, `length` is simply a wrapper around the `Array.length` function that returns the number of binary strings inside the adaptive counter (i.e. the size of the tuple) and additionally throws an error when called on the `AdaptiveCounter.Top` element. The second one, `length_BStr` is used to determine the sum of lengths of all the `BString` objects inside the adaptive counter:

```
# let ac = AdaptiveCounter.create [BString.create [true]; BString.Empty];;
# AdaptiveCounter.print ac;;
(1,e)
# AdaptiveCounter.length ac;;
2
# AdaptiveCounter.length_BStr ac;;
1
```

As all rules described in appendix B depends on k – the last index of the adaptive counter $(s_{d-1}, s_{d-3}, \ldots, s_k)$, we also have a function `last_index` that will return k given an `AdaptiveCounter`. This is done using the formula

$$\text{last index} = (d-1) - 2(l(a)-1)$$

where $l(a)$ denotes the length of the adaptive counter a. Note that if we have an empty adaptive counter, a call to this function will return $d+1$ and that is completely fine – this index is used for trimming and determining whether the adaptive counter is full in the lift method and for those cases $d+1$ works perfectly fine as an indicator of an empty adaptive counter.

Last but not least, the module contains functions such as `is_max`, `is_empty` with obvious functionality and `show` and `print` for debugging purposes.

Modification and access

The `AdaptiveCounter` module has a wide range of methods for manipulation and access of elements required by `ProgressMeasure.lift`. Firstly, we have the `trim` method that given an `AdaptiveCounter` a and a number p returns the p-truncation of a as described in section 3.1. A special case of truncating the top element is also handled as in the paper by returning the top element itself without any modifications. It is also worth mentioning

that if no actual trimming happens – either in the case of an empty adaptive counter or the last index being greater than p, a new `AdaptiveCounter` is returned as a *copy* of the original adaptive counter. That's because we are usually modifying trimmed adaptive counters in the `lift` method, so we need to create a copy to make sure we are not changing the contents of the original adaptive counter.

Apart from `trim` described above, there is also a special trimming operation that given an adaptive counter removes all trailing empty bit strings, called `trim_to_last_nonempty`:

```
# let ac = AdaptiveCounter.create [BString.create [true]; BString.Empty;
↪  BString.Empty];;
# AdaptiveCounter.print ac;;
(1,e,e)
# AdaptiveCounter.trim_to_last_nonempty ac |> AdaptiveCounter.print;;
(1)
```

Finally, we can add an element at the end of an adaptive counter by using the `append` method, remove the last element by using the `remove_last` method and set the element at index n by using the `set` method or the last element by using the `set_last`. Note that the `set` method uses normal array indexing, not the adaptive counter indexing, so e.g. setting the first element would be done by passing $n = 0$. In this implementation, we are only using `set_last` and `set` was only added to facilitate testing.

Comparison

One of the most important parts of the module is the ability to compare different adaptive counters with each other, allowing us to determine which edges (and consequently which nodes) are progressive according to definition 3.3. The `compare` method takes two adaptive counters and returns 1 if the first adaptive counter is bigger, 0 if they are equal and -1 if the second adaptive counter is bigger.

```
let compare ac1 ac2 =
  match ac1, ac2 with
  | Top, Top -> 0
  | Top, ACounter arr -> 1
  | ACounter arr, Top -> -1
  | (ACounter arr1 as ac1), (ACounter arr2 as ac2) ->
    let i = ref 0 in
    let ret = ref 0 in
    while !ret = 0 && !i < length ac1 && !i < length ac2 do
      ret := BString.compare arr1.(!i) arr2.(!i);
      i := !i + 1
    done;
    if !ret = 0 && length ac1 < length ac2 then -1
    else if !ret = 0 && length ac1 > length ac2 then 1
    else !ret
```

There are three special cases that we needed to consider in the method – either one or both adaptive counters being the top element. Those base cases are implemented following the definition of the top element – two top elements are equal and a top element is bigger than any non-top adaptive counter. Next, we have the case of two non-top elements in which case we perform an operation similar to the comparison described for `BString` module, however this time, as we are working with arrays, the comparison is done in an imperative way. We iterate through both adaptive counters and at each index we check whether the binary strings in the two adaptive counters differ. If they do, we return the result of `BString.compare` on those (i.e. 1 if the first adaptive counter had a bigger element at that position etc.). If the adaptive counters being compared are the same at all positions, the return value is determined by analysing their length – it can be the case that one is a prefix of the other.

Next up, the `trim_compare` method given two adaptive counters and a number p, compares the p-truncations of the adaptive counters. This method is used only in the `ProgressMeasure.is_edge_progressive` method described below, but it improves the runtime considerably in comparison to calling `trim` every time we want to check edge progressiveness. This is due to the fact that by performing only an implicit truncation, we do not need to create any new adaptive counters and consequently save time on memory allocation of a new array and copying the elements.

Finally, there are `max` and `min` which given two adaptive counters return the bigger (or respectively smaller) of the two based on the result of `compare`.

3.3.2.3 `ProgressMeasure` module

The `ProgressMeasure` module combines all the work mentioned above to create an implementation of a succinct progress measure described by Jurdziński and Lazić. We implement the succinct progress measure in a natural way as an array in which the ith entry corresponds to the adaptive counter of the ith node (v_i), following the convention from the `Paritygame` module described in section 3.3.1. It also seemed appropriate to associate the lifting methods (`lift` and `lift_` corresponding to the Lift and lift operators described in section 3.2.2) with the `ProgressMeasure` module. The notions of a progressive edge and node are also defined here, in the `is_edge_progressive` and `is_node_progressive` methods. Finally, after the algorithm finishes one can extract the winning set and the winning strategy for the player from the progress measure using `get_winning_set` and `get_winning_strategy`.

Initialisation

At the beginning of solving a parity game, a progress measure is created using the `create` function, which, given a parity game and its set of vertices, performs multiple initialisation procedures for the progress measure. First, the maximum priority in the game is determined and the value d of the smallest even number not smaller than any priority in the game is saved in a reference variable. This value is also saved in the `AdaptiveCounter`

module using the `AdaptiveCounter.set_d` method mentioned in section 3.3.2.2. Next, the number η of odd priorities is determined and stored in a reference variable `eta` and since in the algorithm $\lceil \lg \eta \rceil$ determines the maximum length of an adaptive counter, we also save this value in the `max_len` variable. Finally, an array which assigns to each node an empty adaptive counter is created and returned.

Lifting

The implementation of the lifting methods closely resembles the one in section 3.2.2. Due to naming restrictions in OCaml, the Lift operator is called `lift` in the implementation and lift is called `lift_`. The former takes as arguments the progress measure, the parity game and the node we want to lift and, based on the result of `lift_` on its successors, adjusts the progress measure with an appropriate value of an adaptive counter to make the given node progressive:

```
let lift pm pg node =
  let module AC = AdaptiveCounter in
  let module PG = Paritygame in
  let neighbours = PG.pg_get_successors pg node |> PG.ns_nodes in
  let newAC =
  if PG.pg_get_owner pg node = PG.plr_Even then (** Minimum **)
    List.fold_left (fun acc x -> AC.min acc (lift_ pm pg node x)) AC.Top
    ↪  neighbours
  else (** Maximum **)
    List.fold_left (fun acc x -> AC.max acc (lift_ pm pg node x)) AC.empty
    ↪  neighbours
  in
  set_AC pm node newAC
```

The `lift_` method is where most of the magic described in appendix B happens. As arguments, it takes the progress measure, the game, the node v and the neighbour w and returns an appropriate minimal adaptive counter in order to make the edge between the node and its neighbour progressive. As in the paper, the code is split into different cases:

- The first thing we check is whether the adaptive counter of the neighbour is $\top$. If it is, then we always want to return $\top$ – that is the only possibility to make the edge progressive.
- Otherwise, we check the parity of the priority of v. If it's even, we know that for the edge to be progressive, it's enough if the adaptive counter of v is not smaller than the trimmed adaptive counter of the neighbour. Therefore we can simply return the maximum of the trimmed adaptive counter of the neighbour and the original adaptive counter of v.
- Otherwise, the priority of v is odd. In this case, we perform all the checks described in appendix B. One that is not explicitly mentioned in there and worth noting is

when $\mu(w)|_{\pi(v)}$, trimmed to the last non-empty element turns out to be empty (i.e. $\mu(w)|_{\pi(v)}$ was of the form $(\epsilon, \epsilon, \ldots, \epsilon)$), then the smallest adaptive counter bigger than $\mu(w)|_{\pi(v)}$ is necessarily the top element and that's what we return.

The `lift_` method is where most of the helper operations implemented in the `BString` and `AdaptiveCounter` modules are used – most rules require manipulations of both adaptive counters and binary strings.

Checking node and edge progressiveness

As mentioned before, the `ProgressMeasure` module has two helper functions for determining edge and node progressiveness. `is_edge_progressive` takes the adaptive counters of two nodes and compares their trimmed versions using the `AdaptiveCounter.trim_compare` method. There is a special case of two top elements, in which the edge is always progressive as described in definition 3.3:

```
let is_edge_progressive pm pg n1 n2 =
  let module AC = AdaptiveCounter in
  let ac1 = get_AC pm n1 in
  let ac2 = get_AC pm n2 in
  let p = Paritygame.pg_get_priority pg n1 in
  if AC.is_max ac1 && AC.is_max ac2 then true
  else if p mod 2 = 0 then AC.trim_compare ac1 ac2 p >= 0
  else AC.trim_compare ac1 ac2 p > 0
```

The second one, `is_node_progressive`, makes use of `is_edge_progressive` and based on the owner of the node, checks either for the existence of a progressive edge or that all the edges are progressive:

```
let is_node_progressive pm pg node =
  let owner = Paritygame.pg_get_owner pg node in
  let successors = Paritygame.pg_get_successors pg node in
  if owner = Paritygame.plr_Even then
    (** Even -> exists a progressive edge **)
    Paritygame.ns_exists (fun succ -> is_edge_progressive pm pg node succ)
    ↪  successors
  else
    (** Odd -> all edges progressive **)
    Paritygame.ns_forall (fun succ -> is_edge_progressive pm pg node succ)
    ↪  successors
```

Recovering the solution

Once lifting of the succinct progress measure μ is finished and all nodes are progressive, the winning sets and the strategy of one player can be recovered using `get_winning_set`

and `get_winning_strategy`. The implementation of those is based on theorem 3.11, however for efficiency purposes, `get_winning_strategy` does not pick the neighbour w of v that necessarily minimises $\mu(w)$, but instead it picks any neighbour w' such that (v, w') is progressive in μ – any such neighbour will do. Note, however, that there is no way easy to recover the strategy for the other player from this computation [FL09b]. For that, we need to solve an inverted game as described in section 3.3.2.4.

3.3.2.4 Algorithm

PGSolver uses strategy synthesis (section 2.2) as its definition of solving a parity game, so to count the game as solved we need to find two separate progress measures – one for Even and one for Odd.

There are two approaches to solving the game and recovering the strategies for both players. The first one, used in our implementation, solves two separate games to determine the winning sets and strategies for both players. Alternatively, one can find the progress measure for both players simultaneously – for the details see section 5.3.1. One strong advantage of using the first approach is that once the main structure of the progress measure is implemented, the algorithm itself is clear and concise, whereas in the double-solve approach one needs to modify the actual structure of the progress measure.

The core of this algorithm is the `solve` function. It decides what order of solving the game is the best and also combines the strategies for both players from the solution. As always, let $G = (V, V_e, V_o, E, \pi)$ be the parity game. The idea behind `solve` hinges on one property of the progress measures – finding a progress measure for one player P returns that player's winning set W and strategy σ which automatically also gives us the winning set of the other player, namely $V \setminus W$. Unfortunately, even though we know the winning set of the other player, we don't know their strategy. For that, we need to solve another game. However, we know that this strategy will certainly not pick any nodes in W, because those nodes are all winning for P. Therefore we can guilt-free remove W from the game and create an induced subgame $G[V \setminus W]$. Solving this subgame will determine the strategy for the other player.

Algorithm 3 `solve` function pseudocode

$G = (V, V_e, V_o, E, \pi)$ is the parity game to be solved
even $\leftarrow$ number of even nodes
odd $\leftarrow$ number of odd nodes
if *even* $>$ *odd* **then**
    (W_e, W_o), $\sigma_e \leftarrow$ `solve'` G for Even
    subgame $\leftarrow G[W_o]$
    $\sigma_o \leftarrow$ `solve'` *subgame* for Odd
else if *even* $\leq$ *odd* **then**
    (W_e, W_o), $\sigma_o \leftarrow$ `solve'` G for Odd
    subgame $\leftarrow G[W_e]$
    $\sigma_e \leftarrow$ `solve'` *subgame* for Even
end if
return $(W_e, W_o), (\sigma_e, \sigma_o)$

We can see that `solve` uses multiple helper functions to actually solve the game. The main one, `solve'`, implements the Lifting Algorithm 1. The way the set of non-progressive nodes is updated is justified in section 3.1.2.

```
let solve' pg =
  let module PG = Paritygame in
  let module PM = ProgressMeasure in
  (** Initialise the progress measure **)
  let nodes = PG.collect_nodes pg (fun _ _ -> true) in
  let pm = PM.create pg nodes in
  (** Get all non-progressive **)
  let nonprog = ref (PG.ns_filter (fun node -> PG.pg_get_priority pg node
  ↪  mod 2 = 1) nodes) in
  (** Call lift on non-progressive until all progressive **)
  while not (PG.ns_isEmpty !nonprog) do
    let non_prog_node = PG.ns_some !nonprog in
    nonprog := PG.ns_del non_prog_node !nonprog;
    PM.lift pm pg non_prog_node;
    (** Add predecessors that became non-progressive to the list **)
    let pred = PG.pg_get_predecessors pg non_prog_node in
    PG.ns_iter (fun node ->
      if PG.pg_get_owner pg node = PG.plr_Odd then
      begin
        if not (PM.is_edge_progressive pm pg node non_prog_node)
        then nonprog := PG.ns_add node !nonprog
      end
      else
      begin
        if not (PM.is_node_progressive pm pg node)
```

```
      then nonprog := PG.ns_add node !nonprog
    end) pred;
done;
let sol = PM.get_winning_set pm pg in
let str = PM.get_winning_strategy pm pg nodes in
sol, str
```

The last thing we need for algorithm 3 to work is a way to create and solve the subgame for the second player. This is done using the `invert` function which creates a new game with 1 added to the priority of each node and inverting the roles of the players. Additionally, we use a helper function `subgame_by_node_filter` from the `Paritygame` module described in section 3.3.1 which allows to filter out unnecessary nodes from the game.

The `solve` method is wrapped in the Universal Solver from the PGSolver library that does some pre-processing and initial checks that attempt to simplify the game by reducing its size and lowering the priorities. For more detail on this, refer to [FL09b, Section 2.4.3].

A question one may ask is 'Why does the algorithm choose which game to solve based on the number of even and odd priorities?'. The answer comes straight from [JL17] and the analysis of the performance of the algorithm in section 3.2.2. The runtime of SSPM depends on the number of odd priorities η. Therefore, it seems wise to pick η to be as low as possible in the first run of the algorithm. This doesn't necessarily guarantee better runtime but is certainly an 'educated guess' of what could be faster.

4 Register games

Register games are yet another approach for solving parity games in quasi-polynomial time, introduced by Karoliina Lehtinen in [Leh18].

We start with the definition of a register game and an example of such a game being played. Then we move on to the algorithmic aspect of this method where we explain a systematic approach of performing necessary transformations and present a pseudocode algorithm of how it can be carried out in practice.

The section ends with a discussion on a few problems with the naïve approach we present. We also give a few suggestions of how those issues can be handled.

4.1 Idea

Notation. In this section, every game we consider has a starting vertex and thus a **winning strategy** will refer to a strategy starting at that vertex. Therefore in each game exactly one player has a winning strategy. We say that two games are **winning-strategy-equivalent** if the existence of a winning strategy for the player p in one implies the existence of a winning strategy for p in the other.

The paper by Lehtinen [Leh18] introduces an apparently different approach from Jurdziński's succinct progress measures presented in section 3.2. The parity game $G = (V, V_e, V_o, E, \pi)$ is enhanced with so-called *registers* that keep track of the highest vertex priority encountered since their last reset. This results in a *register game* that is winning-strategy-equivalent to the original game G. This register game can be subsequently transformed into a winning-strategy-equivalent parity game. Such a parity game is significantly (quasi-polynomially) bigger than G but its priorities are bounded by $O(\log n)$, which allows us to use a solver exponential in the number of priorities (such as the succinct progress measures 3.2) to solve the game in quasi-polynomial time [Leh18, Section 5].

We start by introducing the notion of registers. The game $G = (V, V_e, V_o, E, \pi)$ is enhanced by some number k of registers which can hold numbers from 0 up to the highest priority of the game. We will refer to the content of the registers as $r = (r_1, \ldots, r_k)$.

We can now define the *k-register game* as follows. The structure of the underlying game graph is the same as that of G and the game starts at some vertex $v_0 \in V$. Players move between vertices of the graph in the same manner as in the underlying parity game, however, additionally, we have two new rules. First, after each move (v, w), the **registers are updated** – if the content of the ith register, r_i, was smaller than $\pi(w)$ (the priority of the vertex that we move to), then it gets updated to $\pi(w)$. Additionally, player Even **controls the registers**. This means that before each move, if we are at vertex v, then no matter who v belongs to, Even can (but doesn't have to) *reset* one of the registers, say r_i. This does two things

- Produces an output of $2i$ if r_i is even and $2i + 1$ if r_i is odd.

- Changes r_i to 0 and 'moves it to the beginning' of r, shifting everything else.

The **dual** k-register game in which Odd controls the registers is defined equivalently except that resetting r_i produces an output of $2i - 1$ if r_i is odd instead of $2i + 1$. From now on, if we talk about a k-register game without specifying the player controlling the registers, we mean the game in which Even controls the registers.

The game starts at some pre-determined vertex v_0 with $(0, \ldots, 0)$ as the register contents. We say that the player controlling the registers wins if they manage to produce an infinite series of reset outputs with the same parity condition as for the original game, i.e. the highest output occurring infinitely often is of their parity. If the highest output occurring infinitely often is of the opposing parity or the sequence of outputs is finite, the opponent wins.

Playing the game, as well as the idea of resetting the registers are much easier to explain and understand with an example.

Example. As an example, we will show a few initial steps of a play of a 2-register game on the graph from figure 7.

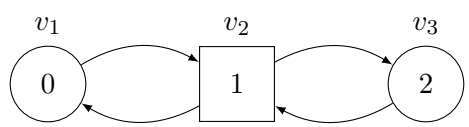

Figure 7: A parity game with three vertices. Circle nodes belong to Odd and square to Even.

vertex	v_1	v_1	v_2	v_2	v_3	v_3	v_2	v_2	v_3
action	*start*	$v_1 \to v_2$	reset r_2	$v_2 \to v_3$	reset r_2	$v_3 \to v_2$	skip	$v_2 \to v_3$	reset r_1
output	–	–	5	–	4	–	–	–	2
r_1	0	1	0	2	0	1	1	2	0
r_2	0	1	1	2	2	2	2	2	2

Table 3: Fragment of a play of the 2-register game obtained from figure 7, to be read from left to right. Register contents in each column represent the register contents *after* the action from that column was executed.

△

k-register games have some remarkable properties [Leh18, Lemma 3.3 and 3.4]:

1. If Odd has a winning strategy in the underlying parity game G, then she also has a winning strategy in the corresponding k-register game for any k.
2. If Even has a winning strategy in a parity game G with k priorities, then Even has a winning strategy in the corresponding k-register game.

Notice that those two facts already give us an equivalence relation between the winning strategies in the original game and the register game with an appropriate amount of registers in the following way. We know that a winning strategy for Even in a parity game G with k priorities implies a strategy in the corresponding k-register game. For the other way around, notice that if Even has a winning strategy in a k-register game for some k, then Odd definitely doesn't, because at most one player can have a winning strategy as mentioned at the beginning of the section. Therefore, by the contrapositive of the first property above, Odd cannot have a winning strategy in the underlying parity game G. Therefore Even must have a winning strategy in G. We have therefore shown that G and the k-register game are winning-strategy-equivalent when k is equal to the number of distinct priorities of G [Leh19].

We will see in section 4.1.1 that each k-register game can be represented as a parity game with priorities bounded by $2k+1$. Let $R_e^k(G)$ denote the parity game obtained by converting a k-register game in which Even controls the registers[9]. Using this notation, we can define the *register-index* of a parity game.

Definition 4.1 (Register-index, [Leh18, Definition 3.5])**.** *The **register index** of a parity game G is the least integer k such that Even wins both G and $R_e^k(G)$ or Odd wins both G and $R_o^k(G)$. This amounts to the least k such that $R_e^k(G)$, $R_o^k(G)$ and G have the same winner.*

The main result that gives us a quasi-polynomial time algorithm for solving parity games using register games is the fact that the register index of a parity game $G = (V, V_e, V_o, E, \pi)$ is bounded above by $\log(|V|)+1$ [Leh18, Theorem 4.7], which means that it suffices to solve $R_e^{\log(|V|)+1}(G)$ to find the winner in the original game G.

4.1.1 Conversion from G to $R_e^k(G)$

To be able to talk about the implementation of this approach, we have to discuss how to encode a k-register game as a parity game. In here, we are going to explain the idea intuitively with an example, for gory formal details see [Leh18, Definition 3.1]. Notice that the result of our encoding will have priorities on *edges*, but this can be easily converted to a game with priorities on vertices in time linear in the size of the input.

Example. Consider the game G in figure 8. We will show how to convert this game into a 1-register game $R_e^1(G)$. First off, at each vertex, we may either have 1 or 0 as the content of our register, implying the following 4 vertices of $R_e^1(G)$ – $(v_1, 0), (v_1, 1), (v_2, 0)$ and $(v_2, 1)$, where the second entry denotes the contents of the register.

[9] $R_o^k(G)$ refers to the parity game representation of the dual k-register game.

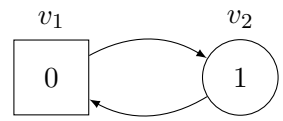

Figure 8: A parity game with two vertices. Square belongs to Even, circle to Odd.

This would be fine, however, we also need to somehow encode the fact that at each vertex we can *reset* the contents of the register. We can do this by splitting each vertex into two stages – 'pre-reset' and 'post-reset'. The 'pre-reset' vertices, denoted by $(v, r, 0)$, $v \in V$, $r \in \{0, 1\}$, will be the ones where Even can make the choice of resetting r and the 'post-reset' ones, denoted by $(v, r, 1)$, $v \in V$, $r \in \{0, 1\}$, will be the ones after the reset has potentially happened, but before the owner of v has made a move to one of its successors. With this, we can define the set of vertices $\mathcal{V}$ of $R_e^1(G)$ as follows:

$$\mathcal{V} = \{(v, r, t) \mid v \in \{v_1, v_2\}, r \in \{0, 1\}, t \in \{0, 1\}\}.$$

What about the edges? From each 'pre-reset' vertex we can either *skip* resetting or *reset* the register, whereas from each 'post-reset' vertex $(v, r, 1)$, we can *move* to some 'pre-reset' vertex $(v', r', 0)$, where v' is the successor of v in G and r' is the updated value of the register according to the rules described in section 4.1. This gives us the following three disjoint subsets of the set of all edges:

$$\begin{aligned}
E_{skip} &= \{((v, r, 0), (v, r, 1)) \mid (v, r, t) \in \mathcal{V}\} \subseteq \mathcal{V} \times \mathcal{V} \\
E_{reset} &= \{((v, r, 0), (v, 0, 1)) \mid (v, r, t) \in \mathcal{V}\} \subseteq \mathcal{V} \times \mathcal{V} \\
E_{move} &= \{((v, r, 1), (v', r', 0)) \mid (v, v') \in E,\ r' = \max(r, \pi(v'))\} \subseteq \mathcal{V} \times \mathcal{V}.
\end{aligned}$$

The last thing to consider is the priority assignment. First, note that at each register reset, we output either 2 if the register contains 0 and 3 if it contains 1. Therefore, each edge $((v, i, 0), (v, 0, 0)) \in E_{reset}$ should have priority $2 + (i \mod 2)$.

By assigning priority 1 to all edges in E_{skip} and E_{move} we ensure that Even cannot win if he never resets. This is because if Even never uses any edges from E_{reset}, the highest priority occurring infinitely often will be 1 and consequently Odd will be the winner. We can see the resulting game in figure 9.

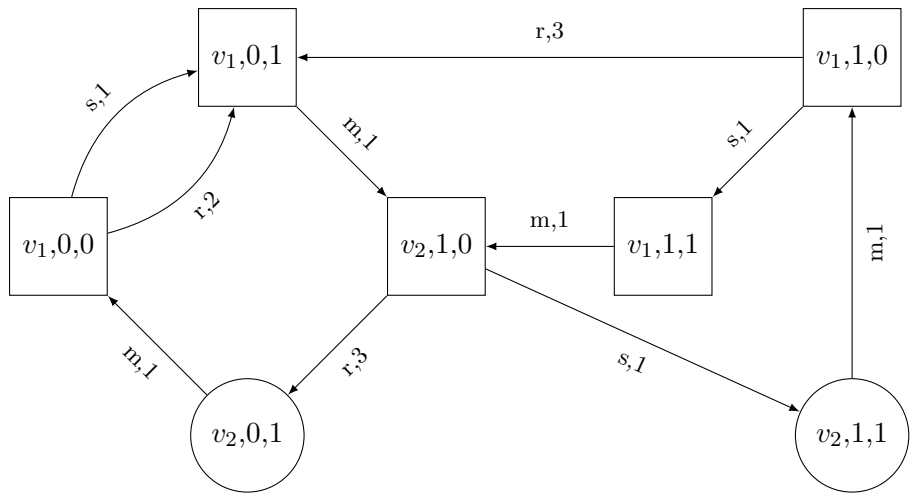

Figure 9: $R_e^1(G)$ with priorities on edges. Square nodes belong to Even and circle ones to Odd. 'm', 's' and 'r' correspond to E_{move}, E_{skip} and E_{reset} edges respectively.

△

4.2 Algorithm

Notation. Let π be a priority function. Following the notation from [Leh18], we are going to write I for the co-domain of π, i.e. $\pi : V \mapsto I$.

In this section, we go over the design and the implementation of the algorithm for converting a parity game G to $R_e^k(G)$. We start with the naïve version and the sketch of the implementation. This is followed by a few comments about the problems of this approach and suggestions for improvement.

The section ends with a discussion about the problem of recovering the strategy from $R_e^k(G)$, which turns out to be a challenge in and of itself.

Appendix D explains where to find the actual implementation of the algorithm in the folder submitted.

4.2.1 Naïve algorithm

The implementation closely resembles the example described in section 4.1.1. The biggest difference is that we are looking for a version conforming to definition 2.1, so we need priorities in vertices instead of edges. We start by creating all the possible vertices $\mathcal{V}_R$ – those include the vertices described in 4.1.1 as well as the reset vertices $\mathcal{V}_r$.

$$\mathcal{V} = \{(v, r, t) \mid v \in V, r \in I^k, t \in \{0, 1\}\},$$
$$\mathcal{V}_r = \{(v, r, 0, i) \mid (v, r, 0) \in \mathcal{V}, i \in \{1, \ldots, k\}\},$$
$$\mathcal{V}_R = \mathcal{V} \cup \mathcal{V}_r.$$

And for the assignment of vertices to players we have $\mathcal{V}_o = \{(v, r, 1) \in \mathcal{V} \mid v \in V_o\}$ and $\mathcal{V}_e = \mathcal{V}_R \setminus \mathcal{V}_o$. Next, we create all the edges. E_{skip} and E_{move} are defined as before and E_{reset} contains edges going into and out of $\mathcal{V}_r$:

$$
\begin{aligned}
&E_{skip} = \{((v, r, 0), (v, r, 1)) \in \mathcal{V} \times \mathcal{V}\},\\
&E_{move} = \{((v, r, 1), (v', r', 0)) \in \mathcal{V} \times \mathcal{V} \mid (v, v') \in E, r' = (\max(r_1, \pi(v')), \ldots, \max(r_k, \pi(v')))\},\\
&E_{pre-reset} = \{((v, r, 0), (v, r, 0, i)) \in \mathcal{V} \times \mathcal{V}_r \mid i \in \{1, \ldots, k\}\},\\
&E_{post-reset} = \{((v, r, 0, i), (v, r', 1)) \in \mathcal{V}_r \times \mathcal{V} \mid r' = (0, r_1, r_2, \ldots, r_{i-1}, r_{i+1}, \ldots, r_k)\},\\
&E_{reset} = E_{pre-reset} \cup E_{post-reset}.
\end{aligned}
$$

The last thing to consider is the assignment of priorities to vertices. Starting with the reset nodes, remembering that resetting r_i outputs $2i$ if r_i is Even and $2i + 1$ otherwise, we can simply assign that value to each vertex in $\mathcal{V}_r$:

$$\pi_R(v, r, 0, i) = \begin{cases} 2i & \text{if } r_i \text{ is Even} \\ 2i + 1 & \text{if } r_i \text{ is Odd} \end{cases} \qquad \forall (v, r, 0, i) \in \mathcal{V}_r.$$

To all other vertices we can assign priority 1 - we want to ensure that Even has to visit the reset vertices infinitely often in order to win

$$\pi_R(v, r, t) = 1 \quad \forall (v, r, t) \in \mathcal{V}.$$

To sum this process up, we present the algorithm for solving a parity game using the register game transformation

Algorithm 4 Register game solver

1: $G = (V, V_e, V_o, E, \pi)$ is the game to be solved
2: Create $\mathcal{V}$, $\mathcal{V}_r$, E_{skip}, E_{move}, E_{reset} and π_R
3: $R_e^{\lceil \log |G| \rceil + 1}(G) \leftarrow (\mathcal{V}_R, \mathcal{V}_e, \mathcal{V}_o, \mathcal{E}, \pi_R)$
4: **Solve** $R_e^{\lceil \log |G| \rceil + 1}(G)$ **using any parity game solver**
5: $(\mathcal{W}_e, \mathcal{W}_o)$, (ζ_e, ζ_o) are the winning sets and strategies respectively in $R_e^{\lceil \log |G| \rceil + 1}(G)$
6: Recover the winning sets (W_e, W_o) for G from $(\mathcal{W}_e, \mathcal{W}_o)$
7: Recover the winning strategies (σ_e, σ_o) for G from (ζ_e, ζ_o)
8: **return** $(W_e, W_o), (\sigma_e, \sigma_o)$

How do we represent and store $R_e^{\lceil \log |G| \rceil + 1}(G)$ in a systematic way? At least in PG-Solver, we want to store all the vertex information in an array. For that, we need a way to map the vertices from $\mathcal{V}_R$ to indices in the array. We are looking for a mapping $f : \mathcal{V}_R \mapsto \mathbb{N}$ which assigns a natural number (i.e. array index) to each vertex in $\mathcal{V}_R$. We won't present the whole idea here – curious readers are encouraged to have a look at the `Converters` module. Instead, we are going to give a brief overview of the conversion from $\mathcal{V}$ to $\mathbb{N}$.

Let $V = \{v_0, \ldots, v_{n-1}\}$ and consider the following mapping

$$f(v_i, r, t) = 2i|I^k| + g(r) + t|I^k|,$$
$$g(r) = g(r_1, \ldots, r_k) = r_1 + |I|r_2 + |I|^2 r_3 + \ldots + |I|^{k-1} r_k.$$

This way f gives us a bijection between $\mathcal{V}$ and $\{0, \ldots, |\mathcal{V}| - 1\}$ and allows to store and operate on the vertices just by looking at their index in the array. We can do a similar thing with $\mathcal{V}_r$ to get a full mapping.

Analysis

We are interested in the performance of algorithm 4 in practice. Consider a parity game $G = (V, V_e, V_o, E, \pi)$ with $|V| = 10$ and $\pi : V \mapsto I = \{0, \ldots 9\}$. What are the respective sizes of $\mathcal{V}$ and $\mathcal{V}_r$? It's enough to have the register index of $k = \lceil \log(|G|) \rceil + 1 = 5$, so we have

$$|\mathcal{V}| = 2|I^k||V| = 2 \cdot 10^5 \cdot 10 = 2 \cdot 10^6,$$
$$|\mathcal{V}_r| = 2k|I^k||V| = 2 \cdot 10 \cdot 10^5 \cdot 10 = 2 \cdot 10^7.$$

It is not a surprise that there is a big (quasi-polynomial) blow-up in the number of states – that's exactly what the algorithm promised. However, having this many nodes already in such a small example makes this approach completely impractical. We need to seek better solutions. Luckily, it seems like the number of vertices can be significantly reduced.

4.2.2 Size reduction

It's not difficult to notice that many of the vertices created are completely useless and are not reachable in the game graph. The first reduction we can apply is considering only *sorted* register contents

Proposition 4.2. *In a k-register game at all times the register contents are sorted, i.e. if $r = (r_1, \ldots, r_k)$ are the register contents during a play, then*

$$r_1 \leq r_2 \leq \ldots \leq r_k.$$

Proof. By induction on the number of actions in the play. At the beginning, the register contents are $(0, \ldots, 0)$. Assume that during the first $n-1$ actions the register contents are sorted and consider the nth action. If this is a *reset* or a *skip* action, the register contents remain sorted trivially. Let this be a *move* action, say from v to v'. By definition, the register contents will get updated to

$$(\max(r_1, \pi(v')), \ldots, \max(r_k, \pi(v'))$$

and clearly

$$\max(r_1, \pi(v')) \leq \max(r_2, \pi(v')) \leq \ldots \leq \max(r_k, \pi(v'))$$

so the register contents remain sorted. □

There is one more property of register games that allows reducing the state space. Consider a vertex $(v, r, 0) \in \mathcal{V}$. Unless v is a starting vertex, we know that all the registers in r have a value of at least $\pi(v)$, because at each move the register contents get updated. Similarly, for a vertex $(v, r, 1) \in \mathcal{V}$, the register contents are at least $(0, \pi(v), \ldots, \pi(v))$, because at most one register could have been reset.

On-the-fly generation

In fact, we can reduce the state space to its lower bound in an algorithmic fashion. Let $G = (V, V_e, V_o, E, \pi)$ be the parity game we want to convert to $R_e^k(G)$. First, find a vertex v with the highest priority, if there are multiple of those, pick any. Then 'simulate' the register game by considering all possible moves from that vertex – in essence, run a graph traversal, generating all the required nodes on the fly. This way we create all the nodes that could definitely appear in $R_e^k(G)$ and nothing else.

We are not going to provide an elaborate analysis here, but experiments show that the decrease in the number of possible states is significant – in fact, an average parity game of 8 vertices with 8 priorities resulted in a translation with around 2500 vertices – much less than the upper bound $|\mathcal{V}| + |\mathcal{V}_r| = 2|V||I^k|(k+1) \approx 300{,}000$. *Nevertheless, even then the number of vertices after the transformation is too big to assume that the algorithm could compete with the best known algorithms or even with succinct progress measures.*

4.2.3 Strategy reconstruction

In algorithm 4 we can see that the last two steps include solution and strategy reconstruction. The solution recovery works as expected – it can be shown that (v, r, t) and (v, r', t') are always winning for the same player [Leh19].

On the other hand, recovering the strategy turns out to be much more delicate. The underlying solver we make use of in line 4 doesn't know what it is solving – it sees $R_e^k(G)$ as every other parity game and returns a strategy that works for $R_e^k(G)$. However, the strategy for $R_e^k(G)$ may, in fact, give a different move from $(v, r, 1)$ than $(v, \bar{r}, 1)$: it can be the case that the strategy from $(v, r, 1)$ is to go to $(v', r', 0)$ whereas the strategy from $(v, \bar{r}, 1)$ is to go to $(v'', r'', 0)$ such that $v' \neq v''$.

Which one of v' and v'' gives the correct strategy in the underlying game G? It seems like this is not an easy question to answer. The problem in a sense boils down to recovering a positional (memoryless) strategy from a non-positional one. After discussing the problem with Karoliina Lehtinen we suggest two methods that may work – the first one by Nathanaël Fijalkow can be found at [Fij18] and the second one by Partick Totzke in appendix C.

5 Evaluation

In this section, we compare the implementation of the succinct progress measures algorithm described in section 3.3 with a few other solvers from the PGSolver library.

We start by explaining the goals of this evaluation and the methodology used and then move on to the results and the analysis thereof. At the end, based on the results, we discuss problems with succinct progress measures and mention potential improvements that one could introduce into the algorithm.

5.1 Goals and approach

The main goal of this evaluation is to see whether the quasi-polynomial time algorithm by Jurdziński and Lazić described in section 3.2 can compete with other algorithms. To this end, we have picked the following algorithms as a base for comparison – the recursive algorithm by Zielonka 2.2.1, which is known for its great actual performance, the strategy improvement algorithm 2.2.1 that is also known to perform well in practice and finally the small progress measures algorithm due to Jurdziński 3.1 which seems like an appropriate baseline for the 'progress-measure-based algorithms'.

The implementations we compare are all taken from the PGSolver library. That's because we are not interested in the actual optimal performance of the algorithm (which, by the way, will definitely be better in C++ than in OCaml), but instead in the relative performance and that's easier to assess by comparing with other algorithms that have the same foundation.

It should be mentioned that PGSolver already has an implementation of the succinct progress measure algorithm (`src/solvers/succinctsmallprogress.ml`). Nevertheless, our implementation presented in section 3.3 beats it by a few orders of magnitude[10].

The analysis focuses on three types of games that were generated using the built-in PGSolver game generator

- random games with arbitrary in- and out-degrees (`randomgame N N 1 N`),
- sparse random games with low in- and out-degrees (`randomgame N N 2 5`),
- steady games with low in- and out-degrees (`steadygame N 1 4 1 4`).

The choice of games was based on the benchmark suggestions from [Kei14]. For each type of game, 30 instances of each of the following sizes were generated:

[10] There are a few problems with `succinctsmallprogress` from PGSolver. First off, debug messages are not enclosed in 'thunks' which means that their content is evaluated even if they are not displayed. Moreover, instead of choosing whether to start by solving the game for Odd or for Even, the game is always solved for Even first, which is much slower on many instances. Fixing those problems would most likely make the two implementations competitive.

$$[50, 100, 200, 400, 600, 800, 1000],$$

which gives 630 games in total. Then for each game size and game type (random, sparse or steady), the solver was run and the time (or timeout) was collected. The time limits were adjusted to the size of the game as follows

Size	50	100	200	400	600	800	1000
Time limit (s)	30	60	120	240	360	480	600

Table 4: Time limits for different game sizes

As we will see, those bounds seem good enough – in fact, if a solver managed to solve the game below the time limit, then the game was almost always solved in less than half of the limit.

Last but not least it should be mentioned that during testing a bug was found in the PGSolver library. The bug drastically slowed down algorithms that used debug messages extensively. We have suggested a fix which was then swiftly accepted and merged to the PGSolver codebase. For the details, see https://github.com/tcsprojects/pgsolver/issues/25.

5.2 Results

The algorithms were run on all 630 games. We are going to present the results for each type of game separately because each game type gave different results. For each game type, we will present the table of the average time of solving the game of each size and the number of timeouts. For a fair assessment of the average time, we are including the timeouts in the calculation and treat each timed-out game as if it was solved in twice the time limit. Each table considers all four algorithms – recursive (`re`), strategy improvement (`si`), small progress measures (`sp`), succinct progress measures (`spm`)[11].

5.2.1 Random games

The first tests ran for the random games were using the standard configuration described in section 5.1, meaning that the PGSolver optimisations were turned on. However, as we can see in table 5 below, this turned out to be completely pointless – the pre-processor took care of solving the game, resulting in very similar times for all the algorithms and all game sizes.

[11] The algorithm identifiers come from PGSolver.

size	50		100		200		400		600		800		1000	
`re`	< 0.01	0	0.01	0	0.03	0	0.23	0	0.85	0	1.62	0	3.21	0
`si`	< 0.01	0	0.01	0	0.04	0	0.23	0	0.69	0	1.63	0	3.12	0
`sp`	< 0.01	0	0.01	0	0.04	0	0.22	0	0.68	0	1.57	0	3.70	0
`spm`	< 0.01	0	0.01	0	0.04	0	0.23	0	0.73	0	1.65	0	2.73	0

Table 5: Average time (seconds) and number of timeouts for random games

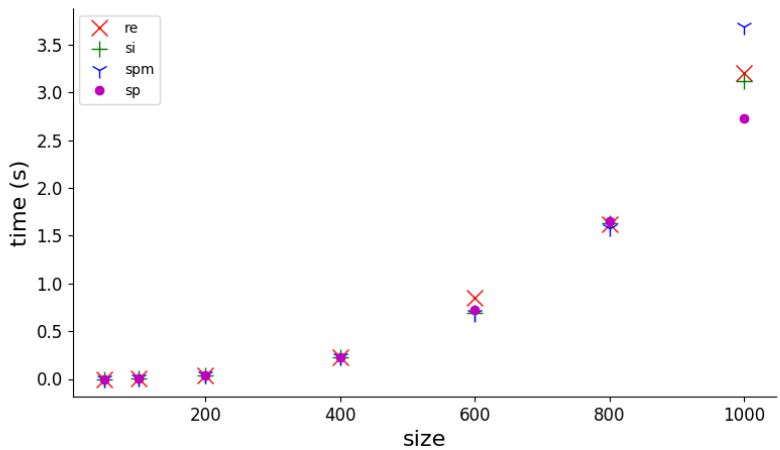

Figure 10: The plot of average runtimes for different sized random games

This observation led to a different approach for random games. All the global optimisations were turned off and different smaller games were generated instead. Below we can see the table of different sizes used and the time limits assigned

Size	30	60	100	150	200	250	300
Time limit (s)	18	36	60	90	120	150	180

Table 6: Game sizes and time limits for random games with no global optimisation

This lead to a more interesting outcome, as can be seen in the table and the plot below

size	30		60		100		150		200		250		300	
`re`	< 0.01	0	< 0.01	0	0.02	0	0.05	0	0.08	0	0.12	0	0.19	0
`si`	< 0.01	0	0.02	0	0.03	0	0.07	0	0.16	0	0.19	0	0.31	0
`sp`	< 0.01	0	2.41	1	0.06	0	6.11	1	0.23	0	0.38	0	12.63	1
`spm`	< 0.01	0	0.05	0	1.52	0	4.10	0	4.85	0	2.55	0	32.93	2

Table 7: Average time (seconds) and number of timeouts for random games without pre-processing

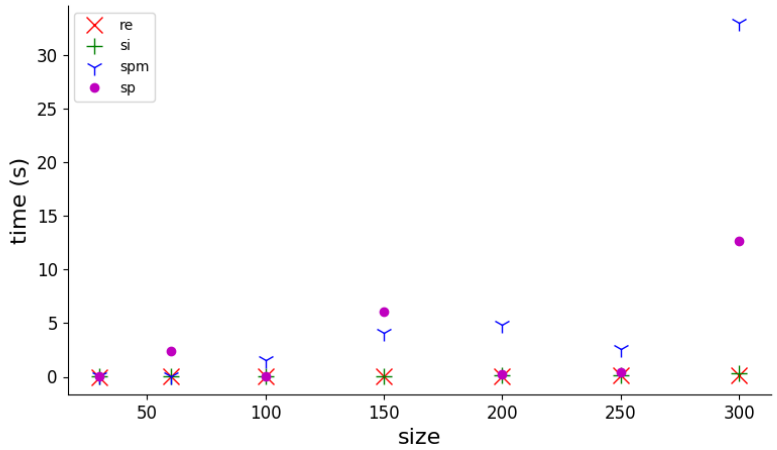

Figure 11: The plot of average runtimes for different sized random games without pre-processing

We can see that the performance of `re` and `si` is out of reach for the progress measure based algorithms. The worst runtime for the first two was below half a second, whereas `sp` and `spm` managed to time out given a 10 minute time limit.

In terms of the number of timeouts, `spm` seems to be slightly more reliable on smaller games than `sp` and has no timeouts for games below 300 nodes. However, even though `spm` doesn't time out, the average time it takes to solve a game is still worse than this of `sp`. This suggests that `sp` is much faster on the non-timeout instances, however, there are some constructions that turn out extremely difficult for it and yield the worst-case runtime. Whereas for `spm` in the current implementation, the size of the game creates more of a concern than its actual structure, as we can see by the two timeouts for the games of size 300.

5.2.2 Sparse random games

Next, the solvers were tested on sparse random games. Those constitute an interesting subset of the random games, because different solvers may behave differently on games of different edge density. Moreover, in sparse games we can usually be safe in assuming that the pre-processor from PGSolver won't take over and solve the game for us – it simply doesn't have enough information to do so.

Below we can see the results of running all four solvers on sparse games of different sizes with the PGSolver optimisation turned on.

size	50		100		200		400		600		800		1000	
`re`	< 0.01	0	< 0.01	0	< 0.01	0	< 0.01	0	< 0.01	0	0.01	0	0.02	0
`si`	< 0.01	0	< 0.01	0	0.01	0	0.02	0	0.03	0	0.04	0	0.07	0
`sp`	0.10	0	4.00	1	32.01	4	81.05	5	144.07	6	160.07	5	240.13	6
`spm`	0.03	0	0.09	0	8.15	1	50.15	3	76.89	3	65.40	2	204.73	5

Table 8: Average time (seconds) and number of timeouts for sparse random games

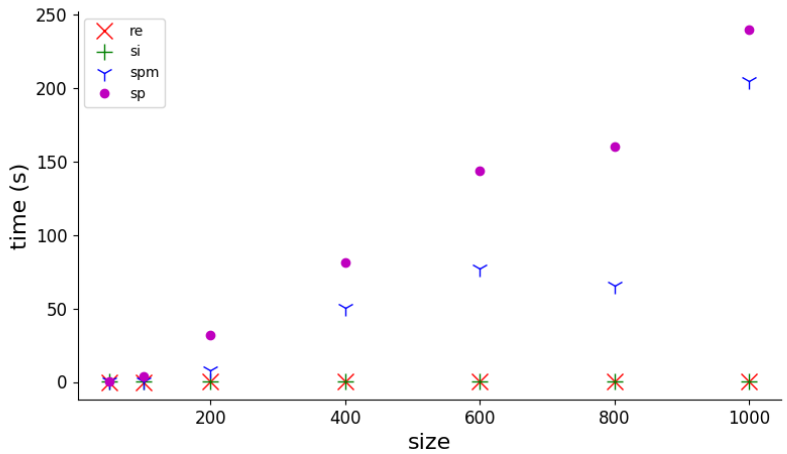

Figure 12: The plot of average runtimes for different sized sparse random games

In sparse random games, we can again see a clear advantage of the recursive and strategy improvement algorithms over both the progress measure algorithms. For all sizes, both `re` and `si` managed to solve the games in under 0.1 second, which is better than the average time for `sp` for games of size 50.

Comparing the progress measure algorithms, we can see that for this type of games there is a slight advantage for the quasi-polynomial version, both in the solving time and the number of timeouts. What's interesting is that even though the timeout threshold for the games of size 1000 was 10 minutes, in all the runs that didn't cause a timeout, the longest solving time for `sp` was only 0.55s, whereas for `spm` 92.26s (with the second longest of 12.47s). This leads to a conclusion that there must arise some instances that yield the worst-case runtime for both algorithms and the solving time for all other instances is way lower. What is also surprising is that it isn't the rule that the two algorithms time out on the same inputs – in fact, out of all 30 generated sparse games of size 1000, there were only two that resulted in a timeout for both algorithms.

5.2.3 Steady games

Last but not least, the algorithms were tested on steady games. As we can read in [FL09b, Section 4.1.3], "The Steady Random Generator tries to circumvent many universal opti-

misations in order to particularly benchmark the backend solvers", so we could expect them to be harsh to the solvers even with the global optimisation turned on.

That is exactly what happened as we can see in table 9. Steady games turned out to be the most difficult for all the algorithms.

size	50		100		200		400		600		800		1000	
`re`	< 0.01	0	< 0.01	0	< 0.01	0	0.01	0	0.015	0	0.02	0	0.03	0
`si`	< 0.01	0	0.01	0	0.02	0	0.05	0	0.08	0	0.12	0	0.18	0
`sp`	6.30	3	24.01	6	64.04	8	32.26	2	120.47	5	128.87	4	161.36	4
`spm`	0.45	0	13.15	3	18.07	2	54.71	3	255.22	10	311.90	9	498.73	12

Table 9: Average time (s) and number of timeouts for steady games

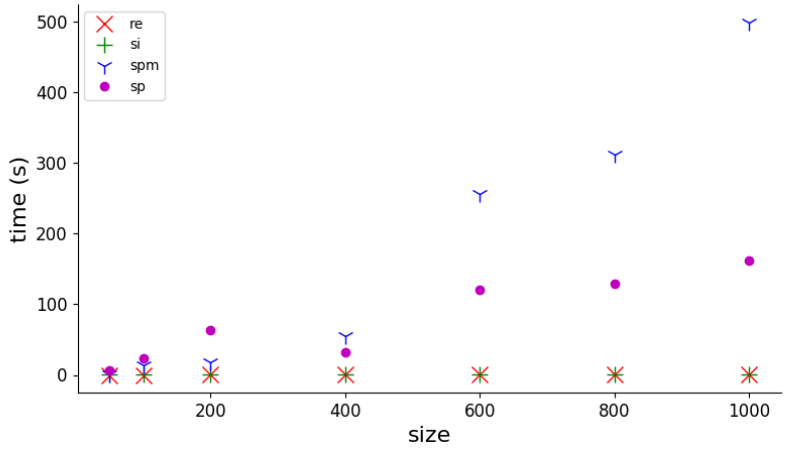

Figure 13: The plot of average runtimes for different sized steady games

Again, the times achieved by the progress measure algorithms are incomparable with the other two. For the rest of this analysis, we focus on the progress measure algorithms, for which we should differentiate between the results for smaller games of size up to 200 nodes and the larger ones.

For games up to size 200, it seems like `sp` is much less reliable with its 10% timeout rate for games with 50 nodes, a 20% timeout rate for games with 100 nodes and a 27% timeout rate for games with 200 nodes, whereas for the succinct progress measures algorithm those numbers are much smaller with 0%, 10% and 7% of timeouts respectively. This also results in a better average runtime for `spm` in those games.

The situation changes for larger games. As we can see, the number of timeouts for `sp` stays relatively constant, whereas for `spm` we see a large increase in the number of timeouts

with the increasing size of the game. This may again, as in the case of random games, suggest that `sp` is using some clever heuristics that allow for solving the game quickly regardless of its size and it's only the game structure and not its size, that can invoke the worst-case runtime.

5.3 Conclusions

The first thing that stands out immediately is that the progress measure algorithms tend to be much slower than the other two. This is not much of a surprise – both the recursive algorithm and the strategy improvement algorithm are known to behave well on a vast majority of games, except for the carefully crafted examples that yield the worst-case exponential runtime.

For `sp` and `spm`, the reality is much different. As we can see from the results, the examples that yield bad runtime are more common and even the 'easy' games take more time to solve than for `re` and `si`.

In the results, we see a bit of a competition between the progress measure algorithms – in random and steady games `sp` turned out to work better, however, in sparse games, it was `spm` that had a slight advantage. We have seen that the rule with `sp` is simple – if it fails, it fails badly, but if it succeeds, it succeeds fast, but slightly slower than `re` and `si`. For `spm` the situation is slightly different. In a sense, `spm` is more reliable than `sp`, especially on the smaller (< 500 nodes) games, because it doesn't time out as often. However, reliability comes at a cost – the average time of solving the game is much higher than for `sp`.

5.3.1 Suggestions for improvements

The results above suggest that the implementation of `sp` does not use the same straightforward approach than the one for `spm` described in Section 3.3. To verify that, we attempted to understand the code for `sp` and see what tricks are used to speed the algorithm up. There is one particular method that makes this algorithm behave much differently from our implementation of the succinct version.

However, regardless of the improvements suggested below, we point out that the results obtained in this analysis have a promising resemblance to those in [vD18] that analysed the algorithms implemented in Oink (see 2.2.2). Oink implementation was done by much more experienced parity game experts and implements a lot of improvements over the naïve methods, but the results are still similar. This may suggest that the implementation proposed in this document is not far off in terms of quality.

Double-solve and attractor computation

The small progress measures algorithm uses what we refer to as the 'double-solve', where

the game is *simultaneously solved for both players*. At each step, a node that is not progressive for one of the players is picked and made progressive by the use of Lift. What is nice about this method is that we are essentially allowing the algorithm to find the progress measure that converges quicker – whether that is the progress measure for Even or for Odd – which allows us to eliminate nodes from the game faster.

This process is accompanied by a so-called 'attractor computation' which runs occassionally in between the lifting calls and attempts to prune some parts of the game as winning for one of the players. The details of both processes are rather arcane, but curious readers are encouraged to have a look at the code for `sp` in the PGSolver library or the description of the small progress measures algorithm in [vD18, Section 5].

The methods described above could definitely be incorporated into the succinct version we have implemented. As suggested by Marcin Jurdziński [Jur19], this should speed things up considerably and seems like a standard approach to implementing the progress measures based algorithms in practice [FL09b, vD18].

Adaptive counter size iteration

Another improvement suggested by Marcin Jurdziński [Jur19] is tied to the succinct representation and the notion of the adaptive counters. Recall that in the original version proposed in section 3.3, we are using the adaptive counters of size $\lceil \lg \eta \rceil$, where η is the number of vertices with odd priority. What one could do instead is solving the game iteratively for bigger and bigger adaptive counters, starting with the size of 1 or 2 and increasing up to the size of $\lceil \lg \eta \rceil$. This approach can then be combined with the double-solve in the following way – if for any size of the adaptive counters the partitions of the game graph produced by the progress measure for Even and the progress measure for Odd are the same, this is guaranteed to be the solution.

Obviously, even with the improvements suggested one can come up with examples of games that induce the worst-case quasi-polynomial runtime. Nevertheless, everything proposed in this section should improve the average performance of the solver.

Finally, it should be mentioned that the analysis here takes into account only three types of games. To obtain a more thorough analysis one could include many other types of games – in fact, there have been attempts to standardise benchmarking of parity game algorithms with [Kei14] being one of the most recent papers on the topic. The benchmarks there include not only the games considered in this document but also encodings of problems such as model checking, equivalence checking and games difficult for specific algorithms.

6 Conclusion

We start our conclusions by reviewing the main achievements presented in this document. We also point out some questions that remain unanswered and give suggested directions of future work in the area of empirical study of quasi-polynomial parity game solvers.

6.1 Achievements

In section 3 we explained the notion of progress measures and presented an OCaml implementation of the succinct progress measures algorithm. Our algorithm performs significantly better than the existing implementation of succinct progress measures in the PGSolver library.

Section 4 presents, as far as we know, the first attempt to implement the register games algorithm. We started with the naïve approach and then discussed a possible improvement in the form of the on-the-fly node generation and mentioned that with our current understanding, the algorithm is unlikely to compete with other well-known solvers. Finally, we pointed out an important problem related to this particular approach of solving parity games – strategy synthesis – which essentially boils down to finding a positional strategy for a game from a non-positional one. We have suggested a few possible approaches for tackling this problem that may be essential in further empirical studies of register games.

We have also compared the performance of our succinct progress measures algorithm with other algorithms from PGSolver in section 5. We concluded that it could potentially compete with the small progress measures algorithm if all the suggestions discussed in 5.3.1 get implemented efficiently. Our results somewhat confirmed the results from [vD18], giving reasons to believe that the implementation in section 3 lives up to the existing standards.

6.2 Future work

Despite the amount of work put into this document, many questions and problems remain unanswered, for both succinct progress measures and register games.

In section 5 we suggested multiple improvements for the succinct progress measure algorithm. Despite the fact that this implementation already beats the other existing implementation of the algorithm in PGSolver, a fully-fledged version of the succinct progress measure algorithm should definitely take those suggestions into account. It would also be interesting to see how they affect the performance in practice and whether it can actually beat the small progress measure algorithm from the PGSolver library (with just those improvements, beating the other two algorithms – recursive and strategy improvement – seems highly unlikely, if not impossible). Moreover, even for the current version of the implementation, the evaluation presented in this document is minimal – we haven't covered many types of games which could give us even more insight into the algorithm and help

us understand its strengths and weaknesses.

We have only scratched the surface of register games. The naïve implementation certainly isn't a good enough base to draw any conclusions about the algorithm's actual performance, despite its worst case quasi-polynomial time and space requirements. As suggested, *on-the-fly generation* 4.2.2 significantly reduces the size of the resulting game. It would be interesting to analyse the problem of creating a register game from a mathematical perspective and derive actual bounds on its size. Furthermore, we state but don't investigate in depth the problem of strategy synthesis. Further work on the practical side of register games should definitely put more effort into solving this problem efficiently and the references provided in this document give a good foundation for that.

More broadly, empirical studies of quasi-polynomial approaches for solving parity games should definitely be continued, despite disheartening results. Both discussed algorithms are relatively new and it may very well be the case that with the use of clever heuristics they could compete with the fastest known algorithms.

References

[A+] Ashish Agarwal et al. What is OCaml? Official OCaml website. https://ocaml.org/learn/description.html, Last accessed on 01-02-2019.

[BDM16] Massimo Benerecetti, Daniele Dell'Erba, and Fabio Mogavero. Solving parity games via priority promotion. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification*, pages 270–290, Cham, 2016. Springer International Publishing.

[BSGV04] Henrik Björklund, Sven Sandberg, and Sergei G. Vorobyov. A combinatorial strongly subexponential strategy improvement algorithm for mean payoff games. volume 155, pages 673–685, 08 2004.

[BW18] Julian Bradfield and Igor Walukiewicz. *The mu-calculus and Model Checking*, pages 871–919. Springer International Publishing, Cham, 2018.

[CDF+18] Wojciech Czerwinski, Laure Daviaud, Nathanaël Fijalkow, Marcin Jurdzinski, Ranko Lazic, and Pawel Parys. Universal trees grow inside separating automata: Quasi-polynomial lower bounds for parity games. *CoRR*, abs/1807.10546, 2018.

[CJK+17] Cristian S Calude, Sanjay Jain, Bakhadyr Khoussainov, Wei Li, and Frank Stephan. Deciding parity games in quasi-polynomial time. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*, pages 252–263. ACM, 2017.

[EJ91] E. A. Emerson and C. S. Jutla. Tree automata, mu-calculus and determinacy. In *Proceedings of the 32Nd Annual Symposium on Foundations of Computer Science*, SFCS '91, pages 368–377, Washington, DC, USA, 1991. IEEE Computer Society.

[EJS93] E Allen Emerson, Charanjit S Jutla, and A Prasad Sistla. On model-checking for fragments of μ-calculus. In *International Conference on Computer Aided Verification*, pages 385–396. Springer, 1993.

[EM79] A Ehrenfeucht and J Mycielski. Positional strategies for mean payoff games. *International Journal of Game Theory*, 8:109–113, 06 1979.

[Fij18] Nathanaël Fijalkow. Positional determinacy for parity games, a backward approach by Muller and Schupp, 08 2018. https://games-automata-play.github.io/blog/MullerSchupp/, Last accessed on 23-03-2019.

[FL09a] Oliver Friedmann and Martin Lange. Solving parity games in practice. pages 182–196, 10 2009.

[FL09b] Oliver Friedmann and Martin Lange. The PGSolver collection of parity game solvers. *University of Munich*, 2009.

[FL10] Oliver Friedmann and Martin Lange. Local strategy improvement for parity game solving. volume 25, pages 118–131, 06 2010.

[HK66] A. J. Hoffman and R. M. Karp. On nonterminating stochastic games. *Management Science*, 12(5):359–370, jan 1966.

[JL17] Marcin Jurdziński and Ranko Lazic. Succinct progress measures for solving parity games. *CoRR*, abs/1702.05051, 2017.

[Jur98] Marcin Jurdzinski. Deciding the winner in parity games is in UP $\cap$ CO-UP. 04 1998.

[Jur00] Marcin Jurdziński. Small progress measures for solving parity games. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 290–301. Springer, 2000.

[Jur19] Marcin Jurdziński. private communication, 03 2019.

[Kei14] Jeroen J. A. Keiren. Benchmarks for parity games. *CoRR*, abs/1407.3121, 2014.

[Leh18] Karoliina Lehtinen. A modal μ perspective on solving parity games in quasi-polynomial time. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, pages 639–648, New York, NY, USA, 2018. ACM.

[Leh19] Karoliina Lehtinen. private communication, 03 2019.

[Pur14] Anuj Puri. Theory of hybrid systems and discrete event systems - chapter 5 on games. 12 2014.

[Sch08] Sven Schewe. An optimal strategy improvement algorithm for solving parity and payoff games. pages 369–384, 09 2008.

[SS98] Perdita Stevens and Colin Stirling. Practical model-checking using games. volume 1384, pages 85–101, 03 1998.

[vD18] Tom van Dijk. Oink: An implementation and evaluation of modern parity game solvers. In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 291–308, Cham, 2018. Springer International Publishing.

[Zie98] Wieslaw Zielonka. Infinite games on finitely coloured graphs with applications to automata on infinite trees. theor. comput. sci. 200(1-2), 135-183. *Theoretical Computer Science*, 200:135–183, 06 1998.

Appendices

A Ordered trees for parity games

Let $T = (V, E)$ be a tree. Following the nomenclature from [JL17], a *branching direction* from a node $v \in V$ is an edge that takes v to any of its children. A *navigation path* is a sequence of branching directions starting from the root.

As an example, if we can get from the root of T to a vertex v via edges $a, b, c \in E$, then (a, b, c) would be a navigation path and c a branching direction at (a, b).

Definition (Ordered tree, [JL17, p. 4])**.** *An **ordered tree** is a prefix-closed set of sequences of elements of a linearly ordered set.*

$S_{\eta,d}$ is an example of an ordered tree – it is certainly prefix-closed and the order on the adaptive counters 3.9 is linear. The set M_G defined in section 3.1 is not prefix-closed, however, if we consider all the truncations of $m \in M_G$, we also get an ordered tree.

As we have seen in the analysis of small progress measures 3.1.2, the number of leaves of M_G (i.e. the size of M_G) is exponential in the size of the parity game G. This issue is addressed in [JL17] with succinct progress measures, the existence of which hinges on the following remarkable result [JL17, Lemma 1].

Lemma (Succinct tree coding)**.** *All navigation paths of an ordered tree of size h with at most n leaves can be encoded using $\lceil \lg n \rceil$-bounded adaptive h-counters.*

The proof of this theorem actually provides an algorithm for creating such encoding of a given ordered tree. For the details see [JL17, Lemma 1].

It can be shown that a *trimmed* parity progress measure [JL17, Section 3.5] induces a tree of size at most $d/2$ with η leaves. Thus we can apply the succinct tree coding lemma to deduce that a succinct progress measure also necessarily always exists.

Last but not least, each $\lceil \lg n \rceil$-bounded adaptive h-counter can be represented using $\lceil \lg n \rceil (\lceil \lg h \rceil + 1)$ bits by just appending to each bit its 'position' in the tuple, which is a number from 0 to $h - 1$. For example, if $a = (0, \epsilon, 011, 1)$ is a 5-bounded adaptive 4-counter, then we can represent it as

$$\mathbf{0}00|\mathbf{0}10|\mathbf{1}10|\mathbf{1}10|\mathbf{1}11,$$

where the numbers in **bold** are the actual bits from a and the ones following them are their respective positions in a. The vertical bars are there just to denote where the representation of a new bit starts. Note that each 'bit-position block' (each sequence between two vertical bars) has the same size, namely $\lceil \lg h \rceil + 1$.

B Succinct progress measure lifting rules

The rules for lifting succinct progress measures are not very intuitive, therefore we provide a small 'rule book' for completeness. This can also be found in [JL17, Theorem 7].

Let $G = (V, V_e, V_o, E, \pi)$ be a parity game, μ a map from V to $S^{\top}_{\eta,d}$ and $(v, w) \in E$ an edge we want to perform lift on. Then if $\pi(v)$ is even or $\mu(w) = \top$, then

$$\text{lift}(\mu, v, w) = \mu(w).$$

Otherwise $\pi(v)$ is odd and $\mu(w) \neq \top$. In this case, there are 5 lifting rules. Let $\mu(w) = (s_{d-1}, s_{d-3}, \ldots, s_{k+2}, s_k)$ and let $\sigma = \text{lift}(\mu, v, w)$. Then

1. If $k > \pi(v)$ then

$$\sigma = (s_{d-1}, \ldots, s_{k+2}, s_k, 0 \ldots 0)$$

   where the padding with zeros is up to the maximum length of $\lceil \lg \eta \rceil$.

2. If $k \leq \pi(v)$ and the total length of s_i for $i \geq \pi(v)$ is less than $\lceil \lg \eta \rceil$ (i.e. there is some padding space), then

$$\sigma = (s_{d-1}, \ldots, s_{\pi(v)+2}, s_{\pi(v)}10 \ldots 0)$$

   where the padding is up to the total length of $\lceil \lg \eta \rceil$ (possibly obtained by just adding 1 and no 0s).

3. If the total length of s_i for $i \geq \pi(v)$ is equal to $\lceil \lg \eta \rceil$ (i.e. there is no padding space), s_j is the rightmost non-empty binary string of $\mu(w)$ and $s_j = s'01 \ldots 1$, then

$$\sigma = (s_{d-1}, \ldots, s_{j+4}, s_{j+2}, s').$$

4. If the total length of s_i for $i \geq \pi(v)$ is equal to $\lceil \lg \eta \rceil$ (i.e. there is no padding space), s_j is the rightmost non-empty binary string of $\mu(w)$, $s_j = 1 \ldots 1$ and $j \neq d-1$, then

$$\sigma = (s_{d-1}, \ldots, s_{j+4}, s_{j+2}10 \ldots 0).$$

   where the total length of σ is still $\lceil \lg \eta \rceil$.

5. Otherwise $\sigma = \top$.

C Register game strategy recovery

In a conversation with the author of the register games algorithm for solving parity games 4, Karoliina Lehtinen, the following idea of strategy recovery by Patrick Totzke was suggested:

Let $G = (V, V_e, V_o, E, \pi)$ be a parity game and $v \in V_e$ a vertex winning for Even that we are trying to find the strategy for. The strategy for $R_e^k(G)$ is essentially a non-positional strategy for G, so think of it as a tree on which all branches satisfy the parity condition. On any path in the tree on which v repeats, check whether the highest priority between the two occurrences of v is even or odd. If it's odd, then such a path is not of interest so it can be 'cut out' from the tree[12]. In a sense, performing such operations results in a 'more positional' strategy.

One thing left out in this reasoning is what to do if v appears on two branches with a common prefix. In this case, we want to consider the branch with 'worse outcome' as the one actually taken. We then merge those two branches at their last common ancestor into the 'worse' one and perform the operations from the paragraph above.

Applying those operations should result in a positional strategy from v. This can then be applied to every vertex in $V_e \cap W_e$ and every vertex in $V_o \cap W_o$ where W_e and W_o are the winning sets of Even and Odd respectively. Obviously, this is far from a proof and more of an intuitive explanation but seems plausible.

[12] We are guaranteed that at least one path will have even highest priority as otherwise v would not be winning for Even.

D Code structure

All the code implemented as a part of this project was an extension of the existing PG-Solver codebase which can be found at https://github.com/tcsprojects/pgsolver.

The most important additions reside in `src/solvers/`. There are four project-related files there that are my sole contribution:

- `src/solvers/sspm.ml` – implementation of the succinct progress measures algorithm described in section 3.3,
- `src/solvers/sspm.mli` – interface for the succinct progress measures algorithm,
- `src/solvers/registergame.ml` – implementation of the register games algorithm described in section 4.2,
- `src/solvers/registergame.mli` – interface for the register games algorithm.

Additionally, for testing and evaluation purposes, two other folders were added to the library. `test/` contains various examples of games used to debug and develop the algorithms. `eval/` contains scripts for evaluating algorithms' performance and plotting the results as well as the games generated for testing and data obtained.