Reproducing parity game algorithms with Oink

Tom van Dijk

Formal Methods and Tools
University of Twente, The Netherlands
t.vandijk@utwente.nl

Abstract Parity games have important practical applications in formal verification and reactive synthesis. Solving parity games is equal to solving the model-checking problem of the modal mu-calculus. They are also central to practical solutions to the reactive synthesis of linear temporal logic specifications. Furthermore, parity games are believed to admit a polynomial-time solution, but so far no such algorithm is known.
We report on the implementation of a number of algorithms in the Oink tool. These are mostly implementations of existing algorithms that have been improved for speed and compared to existing implementations.
We discuss the challenges and findings of implementing these algorithms and compare them with results in the original papers.

1 Introduction

Parity games are turn-based games played on a finite graph. Two players *Odd* and *Even* play an infinite game by moving a token along the edges of the graph. Each vertex is labeled with a natural number *priority* and the winner of the game is determined by the parity of the highest priority that is encountered infinitely often. Player Odd wins if this parity is odd; otherwise, player Even wins.

We provide a more extensive introduction to parity games in [12]. In [12], we implemented a number of modern solvers in the new **Oink** tool, which aimed to provide a high-performance implementation of parity game solvers. We used Oink to provide a modern empirical evaluation of parity game solvers based on both real world benchmarks and randomly generated games. We compared the implementations in Oink to implementations that were publicly available.

In the years before our work, various algorithms were collected in the PGSolver tool and often compared against the implementation of Zielonka's algorithm in PGSolver [17]. However, since publications like [1,28,34,35] suggested that much better performance can be obtained, we reimplemented Zielonka's algorithm with these and additional optimizations to outperform the implementation in PGSolver by several orders of magnitude. We also implemented and compared a number of different algorithms that have been proposed in the years.

We distinguish three broad categories of algorithms. The **strategy improvement** family contains many variations around the idea of fixing a strategy for one player, finding the best response of the opponent, and improving the strategy of the first player until a fixed point is reached. In Oink, we only reimplemented a parallelizable version of this algorithm due to Fearnley. The **value iteration** family contains algorithms that are all based on labeling all vertices with values from a lattice, computing the lowest fixed point of a monotonic function. These algorithms include *small progress measures*, *succinct progress measures*, *ordered progress measures*. In the category of **attractor decomposition** algorithms, we have several versions of *Zielonka's recursive algorithm* including one that runs in *quasi-polynomial time*. We also include a number of algorithms in the *priority promotion* family, as well as many of our own algorithms in the *tangle learning* family.

2 Oink

We refer to [12] for preliminaries and details on the implementation of Oink and the methodology of the empirical evaluation. The Oink tool itself is written in C++ and is publicly available under a permissive Apache-2.0 license via `https://www.github.com/trolando/oink`.

We had several aims when implementing Oink and writing [12]. First and foremost, we were interested in practical performance and understanding the state of the art, due to our practical interest in the model checking of properties LTL, CTL* and the modal μ-calculus.

Furthermore, the possibility of finding a polynomial-time algorithm is compelling for many researchers who (often briefly) study parity games. One way to acquaint oneself with existing solutions is by implementing them. In the process, we found that some papers are rather terse to read and a concise reformulation of the algorithm could be beneficial for a wider audience. Thus, [12] was also written to provide more accessible descriptions of the algorithms. Some of these algorithms were difficult to find implementations of, as the authors often did not provide any. This makes reproducing the results of a paper much harder, as implementation details can have a significant impact on the performance, sometimes resulting in a difference of several orders of magnitude.

Many algorithms were implemented in PGSolver, which is written in the OCaml language. This is a (mostly) functional programming language with which many practitioners are likely unfamiliar, so providing competitive implementations can be difficult. As the evaluation in [12] shows, a re-implementation of basic algorithms in C++ provides an improvement of several orders of magnitude.

3 Strategy Improvement

Strategy improvement is a technique where each player iteratively improves their strategies until they are optimal. Strategy improvement algorithms were first explored for parity games by Jurdziński and Vöge [36] and have been subsequently improved in [7,13,18,29,32]. Parallel implementations have been studied for the GPU [14,21,30]. Fearnley [14] also implements their parallel algorithm for multi-core CPUs.

	Model checking		Equiv checking		Random games		Total	
psi-8	**694**	0	**1078**	0	**315**	0	**2087**	0
psi	860	0	3262	0	480	0	4603	0
psi-1	1190	0	4090	0	487	0	5767	0
parsi-seq	1471	0	4199	0	1534	0	7204	0
parsi-8	2501	1	2908	0	56529	27	61938	28
parsi-1	4200	1	13867	6	71280	39	89347	46
pgsi	167596	88	95407	49	58839	27	321842	164

Table 1: Runtimes in sec. (PAR2) and number of timeouts (15 minutes) of the three solvers PGSolver (`pgsi`), the solver by Fearnley [14] with sequential (`parsi-seq`) and multi-core variants, and Oink with sequential (`psi`) and multi-core variants.

We based our implementation on the work by Fearnley [14]. Over the years, strategy improvement has been thoroughly studied to find a polynomial time solution, as for any given game there always exists a way to solve it in polynomially many iterations of the algorithm. However, no "pivot rule" that selects exactly the right improving edge has been found; for each proposed rule, a game has been found that makes that variant run in superpolynomial time.

In [12], we just implemented the basic greedy "all-switches" pivot rule and we used Fearnley's method of evaluating the paths. Furthermore, Fearnley implemented a parallel version based on list ranking, while our algorithm used a different technique based on independent execution of subtasks in a recursive backward search. We compared the performance of Oink with the sequential and parallel solvers (1 or 8 threads) by Fearnley [14] and the "optstratimprov" solver in PGSolver. We disabled optional preprocessing in all solvers. We only considered games without winner-controlled winning cycles, which are 289 model checking, 182 equivalence checking and 279 random games, in total 750 games. See Table 1 for the results of this evaluation.

While the parallel performance is somewhat interesting, the main point in the context of reproducibility is that a re-implementation of the algorithm (albeit using a different method of parallelization) resulted in a very similar performance, comparing `psi` (our non-parallelized version) to `parsi-seq` (Fearnley's non-parallelized version). Similar to Fearnley, we find that both `psi` and `parsi` are orders of magnitude faster than the implementations of essentially the same algorithm. We did however not implement the GPU-based algorithm of Fearnley and left this to future work. This GPU-based variant is again an order of magnitude faster than the CPU-based version, and is as far as we are aware the fastest method to solve these parity games. Of course, this would be an interesting opportunity to revisit (reproduce) given the advances in the hardware. The implementations by Fearnley are presently still available online[1] and are

[1] https://github.com/jfearnley/parallel-si

published under the BSD 3-clause license. PGSolver can also be found online[2] and is also published under the BSD 3-clause license.

4 Progress Measures (Value Iteration)

Progress measures is a technique that assigns to each vertex a monotonically increasing *measure*. The measure of each vertex is *lifted* based on the measures of its successors. By lifting vertices, players Even and Odd essentially play the game backwards. The measure represents a statistic of the most optimal play so far from the vertex, without storing the plays explicitly.

While progress measures have been used elsewhere, they were introduced for parity games by Jurdziński [24]. Several improvements to the original algorithm are due to Verver [35] and Gazda and Willemse [20]. A number of parallel implementations have been proposed for the Playstation 3 [6], for multi-core architectures [22,31] and for GPUs [8,21]. Different types of progress measures were introduced after the recent breakthrough of a quasi-polynomial time algorithm due to Calude et al. [9], which resulted in the succinct progress measures algorithm by Jurdziński et al. [25] and the ordered progress measures algorithm by Fearnley et al. [16].

In our implementation in Oink, we implemented several improvements to the original work that were already known in the literature.

We also implemented the two quasi-polynomial time algorithms, succinct progress measures and ordered progress measures. Regarding Fearnley's work, the original paper [16] and the journal version [15] both point to a website that no longer exists. We were able to download the source code several years ago, but were unable to compile and compare the implementation, as the implementation depended on proprietary code that was not available. It is possible to find an implementation online[3] by Ding Xiang Fei, who is also acknowledged in [15] but without referencing to this repository. In their journal article, the authors provide some empirical results on the running times of the algorithm on two categories of games: artificial hard games and random games. Their results mainly show that the ordered progress measures algorithm is quite performant compared to succinct progress measures (which times out on almost everything) and that it is quite competitive with classical strategy improvement, Zielonka's recursive algorithm, and small progress measures. However, these results compare their implementation in C++ to implementations of these algorithms in PGSolver, which as [12] demonstrates, are significantly slower than our own C++ implementations. Thus one can hardly call this comparison fair. In our study (Table 2) we find that our implementation of the ordered progress performs better than small progress measures on random games, however it is much worse than the other algorithms on practical parity games. Strictly speaking, we did not reproduce any of the results as they use a slightly different kind of random games (called "steady" games) and we also did not investigate the behavior on the artificial games,

[2] https://github.com/tcsprojects/pgsolver
[3] https://github.com/dingxiangfei2009/qpt-parity

	Model checking		Equiv checking		Random games		Total	
spm	**3637**	1	**7035**	0	168271	93	**178944**	94
qpt	122549	64	65310	31	**66303**	35	254162	130
pbesspm	38397	20	52422	27	183742	101	274561	148
pgspm	88800	45	59885	30	320666	171	469351	246

Table 2: Runtimes in sec. (PAR2) and number of timeouts (15 minutes) of PGSolver (`pgspm`), pbespgsolve (`pbesspm`) and the implementations `spm` and `qpt` in Oink.

but those results are less interesting as there are unsurprisingly also artificial hard games for ordered progress measures that can be solved quickly by other algorithms.

The succinct progress measures algorithm is in our opinion more difficult to understand and implement, although we eventually tackled this challenge. The original paper [25] did not come with an implementation and also does not provide any empirical evaluation. The Fearnley et al. article on ordered progress measures [15] references an implementation of succinct progress measures by Patrick Totzke that is available online[4].

Table 2 presents the results in [12]. We compare our implemention of small progress measures and quasi-polynomial progress measures to the small progress measures implementation of pbespgsolve that comes with the mCRL2 model checker [10,35] and the implementation of small progress measures in PGSolver [17]. The implementation of the parity game solver in mCRL2 was studied in the MSc thesis work by Maks Verver which is still available online[5].

The main takeaway is that comparing to the implementation of small progress measures in PGSolver is not a fair comparison. The online available implementation of succinct progress measures is not published with any license. Similarly, the online available implementation of ordered progress measures is also not published with any license. The mCRL2 toolset is published under the Boost license, which is an open source license that is quite similar to the MIT license.

5 Zielonka's Recursive Algorithm

The algorithm by Zielonka [37] is a recursive solver that despite its relatively bad theoretical complexity is known to outperform other algorithms in practice [17]. Furthermore, tight bounds are known for various classes of games [19].

Zielonka's recursive algorithm is based on attractor computation. By repeatedly computing the attractor (controlled predecessor, backward reachability

[4] `https://github.com/pazz/pgsolver/commits/sspm/`

[5] `https://essay.utwente.nl/64985/1/practical-improvements-to-parity-game-solving.pdf`

	Model checking		Equiv checking		Random games		Total	
zlk-8	94	0	415	0	11	0	**521**	0
zlk	88	0	472	0	**6**	0	566	0
zlk-1	97	0	512	0	7	0	616	0
uzlk	89	0	472	0	69	0	630	0
pbeszlk	64	0	513	0	338	0	915	0
spg-seq	**58**	0	**198**	0	694	0	950	0
spg-mc	389	0	1451	0	72608	37	74447	37
pgzlk	65905	33	68013	36	41629	14	175547	83

Table 3: Runtimes in sec. (PAR2) and number of timeouts (15 minutes) of the four solvers PGSolver (`pgzlk`), SPGSolver (`spg`), pbespgsolve (`pbeszlk`) and Oink (sequential `zlk`, multi-core `zlk-1` and `zlk-8`, unoptimized `uzlk`).

where one player wants to reach and the other player wants to avoid) towards the highest priority vertex, a game can be decomposed into smaller parts. Zielonka's algorithm is one of the first and most well known algorithms. Although the implementation of the recursive algorithm in PGSolver is typically used for comparisons in the literature, improved implementations have been proposed by Verver [35], Di Stasio et al. [34], Liu et al. [28], and Arcucci et al. [1]. Also variations have been proposed such as a subexponential algorithm [26] and the big steps algorithm [33] that have been reported to perform slower than ordinary Zielonka. Arcucci et al. extended the implementation in [34] with a multi-core implementation of attractor computation [1].

The implementation in Oink is based upon the ideas found in the literature, and included a number of additional ideas to improve the implementaiton. We compared our implementation with and without various optimizations to the implementation in PGSolver, to Verver's implementation pbespgsolve [10,35] and to SPGSolver [1,34]. Unfortunately, the Java version of SPGSolver (all three variations) suffered from severe performance degradation for unknown reasons. They also provide a C++ implementation in their online repository, which we used instead. See Table 3.

It is immediately obvious that the implementation in PGSolver is much worse than all other implementations by several orders of magnitude. Furthermore, we find that all other implementations are fairly close to each other in performance. Also interesting was that the optimizations did not seem to have much of an effect except for random games, where our implementation was faster still than the competition. Already in 2013, the implementation by Verver was a highly competitive parity game solver compared to PGSolver.

Unfortunately, Verver never compares their implementation directly to PG-Solver's implementation in the thesis. There is empirical evaluation in [1]. The tool is still available online[6]. Unfortunately, we did not manage to build the Java version that was presented in the paper and thus did not reproduce its performance. The C++ version did not scale very well. The paper was based on earlier work in [34]. Here, they compare their implementation in the Scala programming language to PGSolver which is written in OCaml. While their implementation is faster than PGSolver, this should not be surprising considering Table 3. The implementation of [1] and [34] is published under the MIT license.

6 Priority Promotion

Another family of algorithms closely related to Zielonka's recursive algorithm is priority promotion [4]. Priority promotion starts with a similar decomposition of the game as Zielonka's recursive algorithm, but uses a different method to refine the decomposition.

Priority promotion was proposed in [4] and improved in [2,3]. Unfortunately, these papers are rather terse for no obvious reason. We implemented all algorithms in Oink and found that they all have very similar performance. The journal article on priority promotion [5] contains an extensive evaluation of priority promotion, however they implement their algorithm in the PGSolver framework. We re-implemented the algorithms in C++ and found that the difference with Zielonka's recursive algorithm is not as much as in [5]. Their implementation is publicly available now in the PGSolver repository, although it was not available at the time of [12]. It thus has the same license as PGSolver.

7 Discussion

Apart from the algorithms mentioned above, we also implemented a number of other algorithms that were not described in the original paper. In future work, we may revisit these algorithms and provide an update to [12] with the additional algorithms, such as tangle learning and its variations, fixed point algorithms with freezing or with justifications, the succinct progress measures algorithms, the quasi-polynomial variation of Zielonka's recursive algorithm, new priority promotion variations, etc, and consider including the one GPU implementation by Fearnley.

The main barrier to reproducing results for the TACAS'18 paper was the lack of available implementations. Some of the algorithms were difficult to understand and implementations are very helpful. In a few cases, such as the succinct progress measures, the authors did not provide an empirical evaluation and presumably did not test their algorithm in practice. Most papers provided empirical results, but not in a way that is reproducible. To some extent, this is also simply due to lack of standards to provide artifacts that are reproducible. When implementations are not available, and we have to provide implementations ourselves, small differences in implementation choices can have significant impact on the outcome. Simply

[6] https://github.com/umbertomarotta/SPGSolver/

picking a different programming language already has a major influence, especially when a C++ version is compared to an OCaml version. In recent times, some authors are now using Oink to implement and compare their algorithms and *in some cases* submitting their implementations to the Oink repository.

When we implemented Oink and submitted it to TACAS'18, artifact evaluation was relatively new. The files to reproduce the results of [12] are available online[7]. Unfortunately, the artifact was rejected in the artifact evaluation procedure of TACAS'18 in a way that discouraged any potential effort to improve the artifact anyway and publish it online. We learned from this experience for our follow-up work on tangle learning (a novel parity game solving approach) at CAV'18 [11] several months later, where the artifact was well received. One improvement was to actually make the artifact self-contained, and another was to improve the instructions in the README for the evaluation committee.

Another challenge in trying to reproduce a variety of work in a single way to establish an overview of the state of the art is that every paper uses different benchmarks. Often random games are employed that are not made available. The number of random games is then relatively small, potentially resulting in outliers having a major influence on the outcome. This is somewhat alleviated by having publicly available benchmark sets such as [27] as well as competitions like SYNTCOMP [23].

While most work was licensed with a permissive license, some of the unofficial implementations on GitHub did not carry a license and can therefore not be used.

[7] `https://github.com/trolando/oink-experiments`

References

1. Arcucci, R., Marotta, U., Murano, A., Sorrentino, L.: Parallel parity games: a multicore attractor for the Zielonka recursive algorithm. In: ICCS. Procedia Computer Science, vol. 108, pp. 525–534. Elsevier (2017)
2. Benerecetti, M., Dell'Erba, D., Mogavero, F.: A Delayed Promotion Policy for Parity Games. In: GandALF 2016. EPTCS, vol. 226, pp. 30–45 (2016)
3. Benerecetti, M., Dell'Erba, D., Mogavero, F.: Improving Priority Promotion for Parity Games. In: HVC 2016. LNCS, vol. 10028, pp. 117–133 (2016)
4. Benerecetti, M., Dell'Erba, D., Mogavero, F.: Solving Parity Games via Priority Promotion. In: CAV 2016. LNCS, vol. 9780, pp. 270–290. Springer (2016)
5. Benerecetti, M., Dell'Erba, D., Mogavero, F.: Solving parity games via priority promotion. Formal Methods Syst. Des. **52**(2), 193–226 (2018)
6. van der Berg, F.: Solving parity games on the playstation 3. In: Twente Student Conference (2010)
7. Björklund, H., Vorobyov, S.G.: A combinatorial strongly subexponential strategy improvement algorithm for mean payoff games. Discrete Applied Mathematics **155**(2), 210–229 (2007)
8. Bootsma, P.: Speeding up the small progress measures algorithm for parity games using the GPU. Master's thesis, Eindhoven University of Technology (2013)
9. Calude, C.S., Jain, S., Khoussainov, B., Li, W., Stephan, F.: Deciding parity games in quasipolynomial time. In: STOC. pp. 252–263. ACM (2017)
10. Cranen, S., Groote, J.F., Keiren, J.J.A., Stappers, F.P.M., de Vink, E.P., Wesselink, W., Willemse, T.A.C.: An overview of the mcrl2 toolset and its recent advances. In: TACAS. LNCS, vol. 7795, pp. 199–213. Springer (2013)
11. van Dijk, T.: Attracting tangles to solve parity games. In: CAV (2). Lecture Notes in Computer Science, vol. 10982, pp. 198–215. Springer (2018)
12. van Dijk, T.: Oink: An implementation and evaluation of modern parity game solvers. In: TACAS (1). Lecture Notes in Computer Science, vol. 10805, pp. 291–308. Springer (2018)
13. Fearnley, J.: Non-oblivious strategy improvement. In: LPAR (Dakar). LNCS, vol. 6355, pp. 212–230. Springer (2010)
14. Fearnley, J.: Efficient parallel strategy improvement for parity games. In: CAV (2). LNCS, vol. 10427, pp. 137–154. Springer (2017)
15. Fearnley, J., Jain, S., de Keijzer, B., Schewe, S., Stephan, F., Wojtczak, D.: An ordered approach to solving parity games in quasi-polynomial time and quasi-linear space. Int. J. Softw. Tools Technol. Transf. **21**(3), 325–349 (2019)
16. Fearnley, J., Jain, S., Schewe, S., Stephan, F., Wojtczak, D.: An ordered approach to solving parity games in quasi polynomial time and quasi linear space. In: SPIN. pp. 112–121. ACM (2017)
17. Friedmann, O., Lange, M.: Solving parity games in practice. In: ATVA. LNCS, vol. 5799, pp. 182–196. Springer (2009)
18. Friedmann, O., Lange, M.: Local strategy improvement for parity game solving. In: GANDALF. EPTCS, vol. 25, pp. 118–131 (2010)
19. Gazda, M., Willemse, T.A.C.: Zielonka's recursive algorithm: dull, weak and solitaire games and tighter bounds. In: GandALF. EPTCS, vol. 119, pp. 7–20 (2013)
20. Gazda, M., Willemse, T.A.C.: Improvement in small progress measures. In: GandALF. EPTCS, vol. 193, pp. 158–171 (2015)
21. Hoffmann, P., Luttenberger, M.: Solving parity games on the GPU. In: ATVA. LNCS, vol. 8172, pp. 455–459. Springer (2013)

22. Huth, M., Kuo, J.H., Piterman, N.: Concurrent small progress measures. In: Haifa Verification Conference. LNCS, vol. 7261, pp. 130–144. Springer (2011)
23. Jacobs, S., Pérez, G.A., Abraham, R., Bruyère, V., Cadilhac, M., Colange, M., Delfosse, C., van Dijk, T., Duret-Lutz, A., Faymonville, P., Finkbeiner, B., Khalimov, A., Klein, F., Luttenberger, M., Meyer, K.J., Michaud, T., Pommellet, A., Renkin, F., Schlehuber-Caissier, P., Sakr, M., Sickert, S., Staquet, G., Tamines, C., Tentrup, L., Walker, A.: The reactive synthesis competition (SYNTCOMP): 2018-2021. CoRR **abs/2206.00251** (2022)
24. Jurdzinski, M.: Small progress measures for solving parity games. In: STACS. LNCS, vol. 1770, pp. 290–301. Springer (2000)
25. Jurdzinski, M., Lazic, R.: Succinct progress measures for solving parity games. In: LICS. pp. 1–9. IEEE Computer Society (2017)
26. Jurdzinski, M., Paterson, M., Zwick, U.: A deterministic subexponential algorithm for solving parity games. SIAM J. Comput. **38**(4), 1519–1532 (2008)
27. Keiren, J.J.A.: Benchmarks for parity games. In: FSEN. LNCS, vol. 9392, pp. 127–142. Springer (2015)
28. Liu, Y., Duan, Z., Tian, C.: An improved recursive algorithm for parity games. In: TASE. pp. 154–161. IEEE Computer Society (2014)
29. Luttenberger, M.: Strategy iteration using non-deterministic strategies for solving parity games. CoRR **abs/0806.2923** (2008)
30. Meyer, P.J., Luttenberger, M.: Solving mean-payoff games on the GPU. In: ATVA. LNCS, vol. 9938, pp. 262–267 (2016)
31. van de Pol, J., Weber, M.: A multi-core solver for parity games. Electr. Notes Theor. Comput. Sci. **220**(2), 19–34 (2008)
32. Schewe, S.: An optimal strategy improvement algorithm for solving parity and payoff games. In: CSL. LNCS, vol. 5213, pp. 369–384. Springer (2008)
33. Schewe, S.: Solving parity games in big steps. J. Comput. Syst. Sci. **84**, 243–262 (2017)
34. di Stasio, A., Murano, A., Prignano, V., Sorrentino, L.: Solving parity games in scala. In: FACS. LNCS, vol. 8997, pp. 145–161. Springer (2014)
35. Verver, M.: Practical improvements to parity game solving. Master's thesis, University of Twente (2013)
36. Vöge, J., Jurdzinski, M.: A discrete strategy improvement algorithm for solving parity games. In: CAV. LNCS, vol. 1855, pp. 202–215. Springer (2000)
37. Zielonka, W.: Infinite games on finitely coloured graphs with applications to automata on infinite trees. Theor. Comput. Sci. **200**(1-2), 135–183 (1998)