Finding Smaller Parity Game Solutions by Identifying and Solving Subgames using Oink

STIJN DIJKSTRA, University of Twente, The Netherlands

Solving parity games is an important step in some reactive synthesis tools. Reactive synthesis is the process of synthesising a controller that implements a formal definition. Finding smaller solutions to parity games can lead to smaller controller designs. Decreasing the size of controllers improves their efficiency. We propose an algorithm that identifies subgames of parity games to find smaller solutions. We implement subgame identification by building on the tool Oink and discover how it benefits performance. We compare multiple approaches to the problem and we show that pruning algorithms are a more viable approach to finding subgames than growing algorithms. We also compare the quality-based performance of various solvers to get a better understanding of the tool Oink.

Additional Key Words and Phrases: Parity Games, Reactive Synthesis, Knor, Oink

1 INTRODUCTION

A parity game is a turn-based 2-player game played on a finitely large directed graph. Solving parity games is an important step in synthesising reactive systems from formal specifications. To solve a parity game problem, we need to find a winning strategy for the even player from a specific starting node. This problem was first solved in 1998 by Zielonka [11]. In 2018, the tool Oink was developed by Van Dijk [9]. Oink is a high-performance library that implements multiple state-of-the-art solving algorithms. Oink is integrated into Knor by Van Dijk, Van Abbema and Tomov [10], a tool for synthesising AIGER circuits from linear temporal logic specifications.

In the yearly reactive synthesis competition (SYNTCOMP) [6], researchers compare the tools they developed for solving synthesis problems. In 2020 the parity game track was added, in which the task is to synthesize an AIGER circuit from a parity game. Knor ranked highest in the time-based ranking of the parity game track in 2021 and 2022, performing 10 times faster than the competitors *ltlsynt* and *Strix*. However, Strix performed much better in the quality-based ranking, which is based on the number of gates in the synthesized AIGER circuit.

The size of the AIGER circuit is highly important because it directly relates to the physical size of the hardware. The size of a parity game solution is defined by the number of reachable vertices from the initial vertex and directly relates to the number of latches in the synthesized circuit. However, there is no direct relation between the number of latches in a circuit and the total number of gates in that circuit. Nonetheless, the hypothesis is that *in general* a smaller parity game solution results in a smaller circuit design.

TScIT 40, February 2, 2024, Enschede, The Netherlands
© 2022 University of Twente, Faculty of Electrical Engineering, Mathematics and Computer Science.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

We researched a novel optimization approach that leverages the efficiency of the tangle learning implementation of the Oink solver to find smaller solutions. The approach involves identifying and solving increasingly large subgames until a solvable subgame is found. We hypothesised that this would result in finding smaller solutions.

The contributions of this paper are as follows. We implement novel pruning and growing algorithms to identify subgames in Python to find solutions to the parent game. We evaluate the implemented algorithms by comparing the sizes of their solutions to the solutions found by Oink using the benchmarks from the 2023 edition of SYNTCOMP. We find that these algorithms do **not** find smaller solutions. We discover and compare the behaviour of pruning algorithms and growing algorithms. The discoveries made give further insight into how parity games are structured and what substructures exist within them. We also propose a different method of leveraging the computational efficiency of tangle learning using subgames.

1.1 Outline

The preliminaries aim to familiarize the reader with important concepts by explaining what parity games are and why current solvers have shortcomings, it will then define the concept of a *subgame*. In the methodology section, the paper discusses how the research was conducted and why certain choices were made. This is followed by the results and findings of the research and a discussion of the results, including a recommendation for how the results can be used in future work.

2 RELATED WORK

As mentioned in the introduction, our research is based on the research done during the development of the tools Oink [2, 8, 9] and Knor [10]. Other tools that compete in the parity game category of SYNTCOMP are Strix [3, 4] and ltlsynt [5, 7]. Both configurations of Strix perform the best in the quality-based ranking.

One thing we can see in the results from SYNTCOMP 2022 [6] is that the different configurations of Knor (*synt_sym*, *synt_sym_abc* and *synt_tl*) perform very differently. The *synt_sym* and *synt_sym_abc* configurations implement symbolic algorithms based on fixpoint algorithms developed by Lijzenga and Van Dijk [2]. While the *synt_tl* configuration implements tangle learning developed by Van Dijk [8]. The tangle learning approach performs the best in the time-based rankings by far. However, in the quality-based ranking, it is not competitive.

As far as we are aware, the approach of using subgames to find smaller solutions has not been explored before.

3 PRELIMINARIES

3.1 Parity Games

A parity game is a type of directional graph in which each vertex has a priority given by a positive integer. In a parity game each vertex must have at least one outgoing edge. A parity game can be seen as an infinite game played between two players who both control the same pawn that moves between vertices. Each vertex is owned by one and only one of the two players. The two players are referred to as 'even' and 'odd'. Players do not take turns in the traditional sense, instead, the acting player is determined by who owns the vertex that the pawn is currently on. The acting player chooses which outgoing edge the pawn traverses next. In other words, the player who owns the vertex decides the next move. Since each vertex has at least one outgoing edge and players *must* make a move, a parity game never ends.

To determine the winner of a parity game we assume that each player has a defined strategy, meaning that from each vertex a player owns, that player will always make the same move, other information is irrelevant. Positional strategies are applicable to parity games. In other words, to determine the optimal next move, the player only needs to know the current position of the pawn, no information about any previous moves is necessary. Since the graph is finite and the number of vertices visited in a game is infinite, the game always eventually visits a subset of the vertices an infinite number of times. To determine the winner of a game given the two positional strategies, we determine the subset of vertices that are infinitely visited. The winner is given by the parity of the highest priority in that subset.

When both players have a strategy, we can determine for each vertex which player would win if the pawn starts from that vertex. Additionally, algorithms that compute the optimal strategy for one player, also compute the optimal strategy for the other player as part of that process.

3.2 Solutions and solution domains

A solution to a parity game is given as the optimal strategy for the even player and includes for each vertex whether that vertex is won using this strategy. Within the field of reactive synthesis, we are only interested in a single *initial vertex* and its *solution domain*. The initial vertex represents the initial state of the controller that we are synthesising. If the initial vertex is not won by the even player, we consider the parity game *unsolvable* and a controller can not be synthesised.

The solution domain is defined as the subset of vertices that can be reached from the initial vertex given the strategy of the even player. Since we have no control over the strategy of the odd player, we look at all possible paths that can be taken. The *solution size* is the size of the solution domain. All the vertices that are *not* in the solution domain represent non-reachable states, which can be ignored in the following synthesis steps.

Figures 1, 2, 3 and 4 show how the solution domain is determined. For this to be a solution domain and not just a subgame, the highest priority in each of the 3 possible cycles must be *even*. Otherwise, it means that an odd strategy exists for which we do not win.

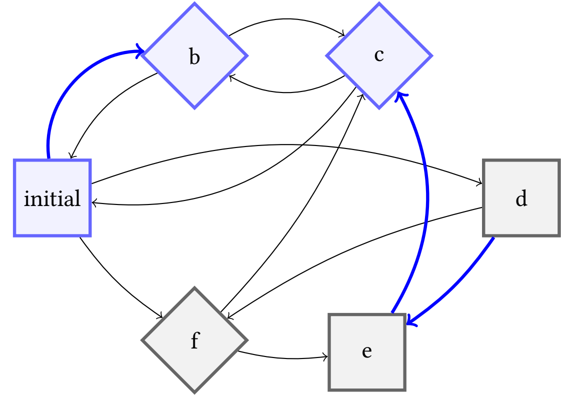

Fig. 1. Assuming the even strategy $((init \rightarrow b), (d \rightarrow e), (e \rightarrow c))$ (given in blue edges) is a winning strategy, the blue vertices would be the solution domain. Figures 2, 3 and 4 illustrate all the possible cycles that can result from different odd strategies. Squares are even vertices, and diamonds are odd vertices. Priorities are not shown here.

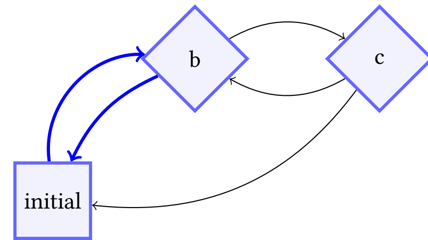

Fig. 2. Given the even strategy illustrated in Figure 1. This figure illustrates the infinite cycle given the odd strategy $((b \rightarrow init), (c \rightarrow ?), (f \rightarrow ?))$.

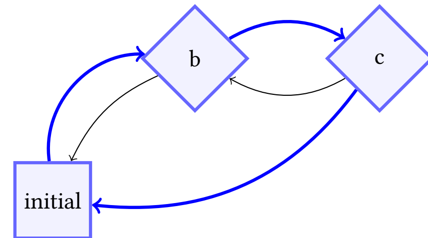

Fig. 3. Given the even strategy illustrated in Figure 1. This figure illustrates the infinite cycle given the odd strategy $((b \rightarrow c), (c \rightarrow init), (f \rightarrow ?))$.

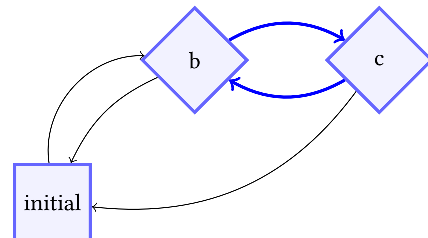

Fig. 4. Given the even strategy illustrated in Figure 1. This figure illustrates the infinite cycle given the odd strategy $((b \rightarrow c), (c \rightarrow b), (f \rightarrow ?))$. Notice that the initial vertex is not always included in the infinite cycle.

3.3 Subgames

A subgame is a region of the original parity game (from now called the *parent game*) that contains the initial vertex. For a region to be considered a subgame there needs to exist a strategy for the even player such that the pawn always stays within the region, regardless of the strategy of the odd player. In other words, all vertices in the region that the even player owns should have *at least one* successor inside the region, and all vertices in the region that the odd player owns should *not* have *any* successor outside the region. Any vertices for which these requirements do not hold are deemed *problematic*.

Note that the concept of subgames in this case relies on the existence of an initial vertex. This is an extension of the concept of a parity game, which does not normally have a defined initial vertex. The initial vertex can be added because in the context of reactive synthesis, there is also an initial state. This means that the proposed improvements do not apply to parity games in general.

3.4 Tangle Learning

The reason why the synt_tl configuration of Knor performs faster is that it uses tangle learning, developed by Van Dijk [8]. Tangle learning revolves around finding tangles and dominions. The issue with tangle learning is that it works by gradually increasing the size of tangles, which may make it less likely to find small solutions.

3.5 Visual Representation

To better explain our findings we use multiple graphs in this paper. In these graphs, we use squares to represent even vertices and diamonds to represent odd vertices. When discussing a set of vertices, we use green vertices to indicate vertices that are within the set and are non-problematic. We use red vertices to indicate vertices that are within the set and are problematic. Grey vertices are outside of the set.

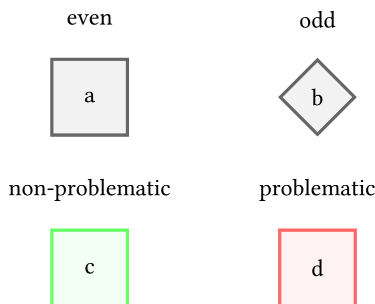

Fig. 5. A visual example of the different possible vertices.

4 METHODOLOGY

4.1 Technologies and Tools

While Knor and Oink are both written primarily in C++ and C, we use Python for the implementation of our algorithms. Using Python does limit the possibilities of using execution time metrics for comparing performance. However, our research is primarily concerned with finding higher-quality solutions, rather than finding solutions quicker, meaning that execution time metrics are less important for evaluating results. Additionally, we run the larger benchmarks, which Oink can solve in seconds, for up to 10 minutes. This way we ensure that no interesting results are lost due to poor optimization. We use the parity game format that was previously introduced by the tool PGSolver [1] and is now being widely used in related research.

4.2 Identifying Subgames

The methods for subgame identification that we use all consist of 2 components. The first component generates a set of vertices based on the distance of each vertex from the initial vertex. The algorithm starts with a small set of vertices and gradually increases the maximum distance from the initial vertex.

The distance can be calculated in various ways. We evaluate three options; SDSI (Simple Distance Subgame Identification), SDSI-BI (SDSI-Bidirectional) and SDSI-REV (SDSI-Reverse). SDSI calculates the distance based on the number of edges that need to be traversed to get from the initial vertex to the other vertex, taking into account the direction of the edges. SDSI-BI and SDSI-REV do the same, except SDSI-BI ignores the direction of edges, and SDSI-REV traverses edges in the reverse direction.

The vertexset that is generated by the first component is in most cases not a subgame. It is more likely that it contains one or more problematic vertices. The second component is responsible for removing those problematic vertices from the vertexset. We evaluated two approaches; pruning and growing. Pruning finds a subgame by repeatedly removing vertices from the set. Growing finds a subgame by repeatedly adding vertices to the set. These approaches are explained in more detail in the *findings and results* section.

4.3 Evaluating Results

To evaluate the results we time the execution of different algorithms for each benchmark. We use the benchmarks of SYNTCOMP 2023 and use Knor to convert them to parity games in the PGSolver format.[1] This means that we evaluate the algorithms using the exact same experiments as were used to evaluate the performance of Oink. A 60-second timeout was used, this should be more than enough, even on a relatively slow system.

5 FINDINGS AND RESULTS

In this section, we discuss the performance of various algorithmic approaches to the problem of subgame identification. We first discuss the differences in performance between SDSI, SDSI-Bidirectional and SDSI-Reverse, all using pruning as the second component. Then we discuss the performance of pruning methods and why the growing methods did not offer valuable results. Finally, we discuss our findings regarding subgames and how they relate to each other and other structures. For reference, the system used for performance evaluation is a desktop system running Windows and a Debian image using WSL. The system uses an Intel i5-4670k at 4GHz and 16Gb of memory. Oink is not GPU-accelerated.

5.1 SDSI, SDSI-BI and SDSI-REV

During testing we found that the algorithm SDSI-Reverse has an issue. Since the algorithm only traverses edges backwards, and some vertices are only reachable from the initial vertex by traversing the

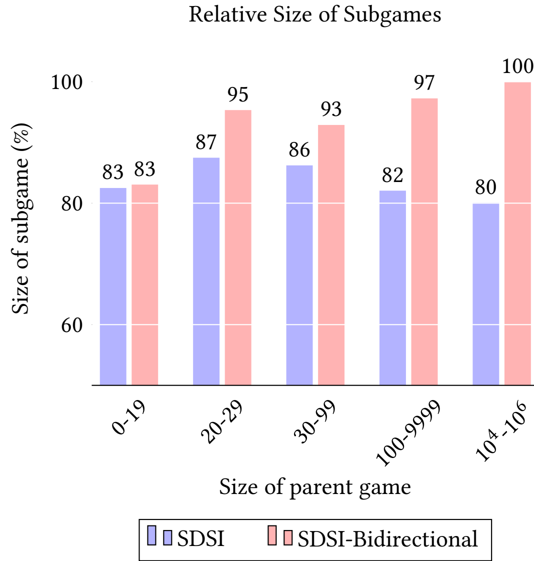

Fig. 6. The average size of subgames found by SDSI and SDSI-BI, relative to the parent game. The games are grouped by the number of vertices in the parent game.

edges in the normal direction, it is not guaranteed that as we increase the maximum distance the whole graph is eventually included in the vertexset. This means that it will not contain all possible subgames. Therefore we op to exclude it from our results.

For SDSI it is also not guaranteed that it will eventually contain all vertices. However, SDSI will always eventually include all vertices that are reachable from the initial vertex. Since any vertices in a solution domain *must* be reachable from the initial vertex, we know that SDSI never excludes any vertices that may be part of a solution domain.

SDSI-Bidirectional will always eventually include the whole game, meaning that it also never misses any subgames. However, this is also why SDSI works better than SDSI-Bidirectional in many cases. SDSI will never include any vertices that are not reachable from the initial vertex, while SDSI-Bidirectional will sometimes include vertices that can not be part of a solution domain. One situation where SDSI-Bidirectional offers better results is when the initial vertex is part of a cycle, then it requires fewer iterations because each iteration travels that cycle in both directions simultaneously. Fewer iterations can lead to a smaller subgame.

As we can see in Table 1 SDSI finds subgames in 135 out of 208 cases, whereas SDSI-Bidirectional only finds subgames in 55 cases. However, in some cases, SDSI-Bidirectional finds even smaller subgames than SDSI. In Figure 6 we show the size of subgames found by SDSI and SDSI-BI relative to the size of the parent game. We grouped the 208 different benchmarks by their size and took the average relative size. The 0-19 group contains 44 benchmarks, the 20-29 group contains 40 benchmarks, the 30-99 group contains 54 benchmarks, the 100-9999 group contains 61 benchmarks and the 10^4-10^6 group contains 9 benchmarks. This graph clearly shows that in the benchmarks that we tested, the benefit of SDSI-BI diminishes when games get larger, while the relative size of SDSI remains around 84% for each group of games.

	<	=	>
SDSI v Parent	135	73	0
SDSI-BI v Parent	55	153	0
SDSI v SDSI-BI	102	97	9

Table 1. This table compares the size of the subgames found by SDSI and SDSI-BI to the size of the Parent game and compares them to each other. For example, there are 135 benchmarks where SDSI finds a subgame that is smaller than the parent game, and there are 73 benchmarks where the best subgame it finds is as big as the parent game, meaning that it is the parent game itself.

5.2 Pruning

When a set of vertices does not represent a subgame because there exist problematic vertices in the set, it is possible to prune it. The process of pruning repeatedly removes any problematic vertices. Note that removing a vertex can cause neighbouring vertices to become problematic, that is why we should repeat the process over multiple iterations. If at any point the initial vertex is removed, we should abort the process and the pruning is unsuccessful. In figures 7, 8, 9 and 10 we illustrate how the algorithm works.

An interesting characteristic of pruning is that a problematic vertex can never become non-problematic by removing another vertex, thus pruning always results in the same subgame independent of in what order the problematic vertices are removed. This means that there is never an intermediate state of the set that is a subgame. This also means that pruning always finds the *largest* possible subgame in a set of vertices. Using this information we can conclude that if the pruning process is unsuccessful, it is proven that there exists no subgame within the given set of vertices.

We improved the pruning algorithm further by only considering the neighbouring vertices of vertices that were deleted in the previous iteration. When vertex f is removed from the set as illustrated

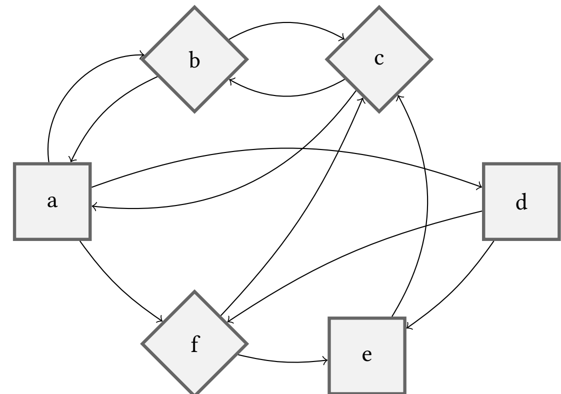

Fig. 7. The complete parity game with initial vertex a. In this example, we are interested in finding the largest subgame that does not contain the vertex e.

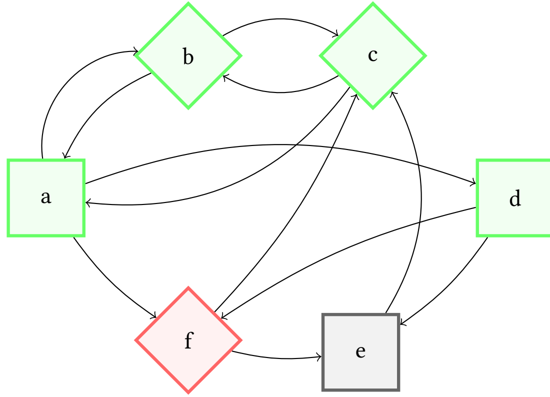

Fig. 8. Vertex f is problematic because it is odd and it has an outgoing edge to vertex e which is outside the set. Vertex d also has an outgoing edge to vertex e but is not problematic because it is even *and* it also has an outgoing edge to vertex f (which at this point is still in the set).

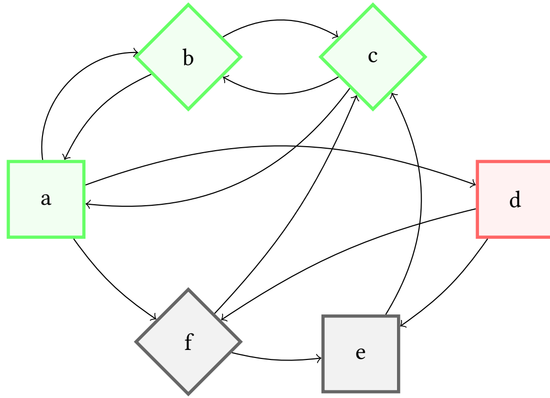

Fig. 9. Vertex d has now become problematic because it has no successors within the set.

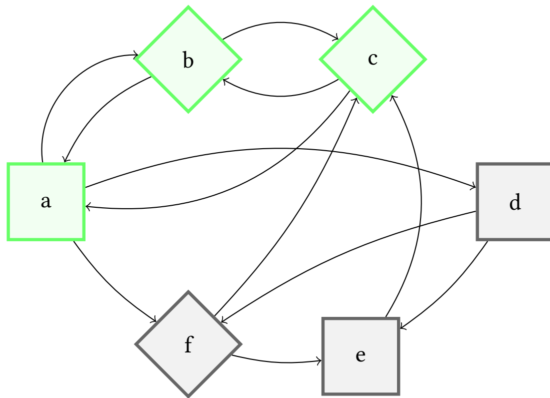

Fig. 10. There are no longer any problematic vertices. Both odd vertices b and c have no successors outside the set and the even vertex a has one successor within the set. The set of vertices (a, b, c) represents a subgame. Subgame (a, b, c) is the largest subgame that does not contain the vertex e.

in figure 8 only the vertices a, c and d would be rechecked. The efficiency improvement that comes from this method depends heavily on the structure of the game. Structures in which only one vertex is removed per iteration benefit most from this optimization.

In Algorithm 1 we show the optimized algorithm. In this algorithm, G indicates the parent game, V indicates the vertexset that is being pruned, P is a temporary value that stores the problematic vertices of each iteration and N is a temporary value that stores the neighbours of the vertices that were in P in the previous iteration.

Algorithm 1 Pruning algorithm

```
G = parent game
V = vertexset to prune
P ← ∅
for v ∈ V do
    if owner(v) = even then
        if successors(v) ∩ V = ∅ then
            ▹ Vertex is even and has no successors in the set.
            P ← P + v
        end if
    else if owner(v) = odd then
        if successors(v) − V ≠ ∅ then
            ▹ Vertex is odd and has successors outside the set.
            P ← P + v
        end if
    end if
end for
while P ≠ ∅ ∧ initial(G) ∉ P do
    V ← V − P
    N ← successors(P) ∪ predecessors(P)
    P ← ∅
    for n ∈ N do
        if owner(n) = even then
            if successors(n) ∩ V = ∅ then
                ▹ Vertex is even and has no successors in the set.
                P ← P + n
            end if
        else if owner(n) = odd then
            if successors(n) − V ≠ ∅ then
                ▹ Vertex is odd and has successors outside the set.
                P ← P + n
            end if
        end if
    end for
end while
if P ≠ ∅ then
    V ← ∅
end if
```

5.3 Growing

The alternative to pruning is growing. We start with a set of vertices that is not a subgame and try to resolve any problems by adding vertices. In contrast to pruning, growing can lead to multiple different subgames, as it is dependent on the order of adding vertices. One problem can have multiple solutions.

We tested an algorithm which creates a new branch for each possible next vertexset. We tested this method with both a breadth-first approach and a depth-first approach by using queue and stack data structures respectively. We applied additional optimization by storing a hash of each previously evaluated vertexset, so that other branches can be evaluated more quickly. Both the breadth-first approach and the depth-first approach were unsuccessful because the number of branches increases too rapidly with each iteration. Results show that the computational complexity of these algorithms is too high for larger parity games.

We opt to not include the performance metrics of the growing algorithms because there is no value in them. The algorithms are only capable of finding solutions to the smallest parity games in the benchmark. The existing solvers in Oink are capable of solving them in milliseconds and these implementations are unable to solve them within 60 seconds. Furthermore, we tested the algorithms with a 10-minute timeout on a select number of parity games from the benchmark and found that they were also unable to find smaller solutions.

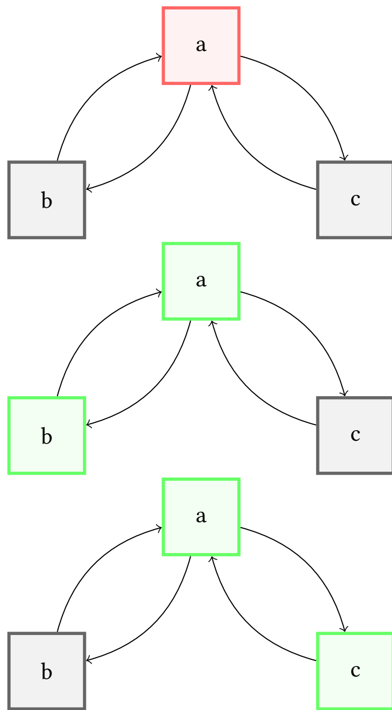

Fig. 11. An example where the growing method has multiple possible solutions. Vertex a is problematic because it does not have a successor inside the set. Adding either vertex b or vertex c to the set could resolve this problem and create a subgame.

5.4 Subgames

We discovered that the vertices in a solution domain always represent a subgame. For any given solution domain, it must be true that there is no way for the odd player to escape the domain. It must also be true that all even vertices in the domain have at least one successor going back into the domain. It must also be true that the solution domain contains the initial vertex. This means that all the requirements hold for the set of vertices of the solution domain to be a subgame. Therefore all solution domains are subgames.

More formally, let S be the set of sets that represents all possible subgames in parent game G and let D be the set of sets that represents all possible solution domains in the parent game G. We know $\forall d \in D \exists s \in S(s = d)$. Then we also know that $D \subseteq S$. In some parity games $D = S$, for example, a trivial game with only one vertex with even priority with only one outgoing edge back to itself. This does not hold for all parity games because there are games that are unsolvable and any game can be expressed as a subgame of a larger game. Any unsolvable game logically does not contain a solution domain because there is no solution.

We experimented using this property to quickly find small subgames. We first solve the parent game using Oink and get a solution domain d. Then for each vertex $v \in d$, we create a new set d' where $d' = d - v$. For each set d' we prune the set. If the pruning is successful, the resulting subgame is again solved using Oink. For each subgame that is solvable, we calculate the size of the solution domain. We use the solution domain with the smallest size to repeat the process until we no longer find smaller solution domains. This greedy approach does work for quickly finding small subgames, but it does not lead to smaller solution sizes.

5.5 Evaluation of Oink

After finding the negative results of the subgame-solving approach, we decided to further analyse the performance of the various solvers in Oink. We tested Tangle Learning (TL), Parallel Distraction Fixpoint Iteration (FPI), Fixpoint Iteration using Justifications (FPJ), Parallel Zielonka's Recursive Algorithm (ZLK), Parallel Strategy Improvement (PSI), and Priority Promotion (PP). Contrary to our expectations, all the algorithms found solutions with the exact same sizes for every single problem, as seen in Table 2. This suggests that those solutions may be the best possible solutions to the parity game problems. However, as noted by Van Dijk in [8] many solving algorithms implicitly use the concept of tangles. This indicates that the various solving algorithms may be more similar to each other than previously thought and may mean that all those algorithms have similar blind spots.

6 DISCUSSION

The performance of our testing tool is limited because our subgame identification algorithms are not integrated directly with the solving algorithms in Oink. Currently, the implementation stores data in files before using Oink to solve them. This adds considerable overhead to solving games.

	finds smallest known solutions	runtime (full benchmark)
TL	✓	3.445s
FPI	✓	7.191s
FPJ	✓	4.840s
ZLK	✓	7.147s
PSI	✓	7.539s
PP	✓	3.512s

Table 2. The performance of 6 solvers in Oink with parallelization enabled. We see that all solvers are capable of finding the smallest known solutions to the problems.

We would suggest further research on the performance of the SDSI and pruning combination. Currently, it is only tested with the 208 benchmarks from SYNTCOMP.

As we discussed in our findings, for any possible game the set of solution domains is a subset of the set of subgames. Therefore it may seem more interesting to look deeper into solution domains. However, at that point, we are straying so far from subgame identification that it becomes an addition to existing solving techniques. We deliberately tried not to create a quasi-solving algorithm.

We pivoted our approach multiple times during our research. In multiple instances, the preliminary results of an implemented algorithm showed that our initial approach was not viable. This is unavoidable in exploratory research. During implementation, we gained better insight into how computationally expensive some initially proposed algorithms are, a good example of this is the growing algorithm.

7 CONCLUSION

In this work, we explored the problem of subgame identification as a separate problem from parity game solving. We created algorithms that are capable of identifying subgames in parity games and determined that using SDSI with pruning is the best approach for identifying subgames among the tested algorithms. We hypothesised that finding small subgames would also lead to small solutions, but our data shows no evidence of this.

As we noted in our findings, the mediocre performance in the quality-based rankings of SYNTCOMP is not a result of the tangle learning algorithm. All the existing algorithms in Oink, including tangle learning, are capable of finding small solutions to parity games. Therefore subgame identification may not be the solution to the issues with quality-based performance in Oink.

However, there are potential efficiency benefits to using subgame identification as a preprocessing step. We propose a system that uses a combination of the available solvers. This allows us to verify that subgames have a solution first using the quick evaluation that tangle learning in Oink offers, and then find a small solution to that subgame by using a different solving algorithm. In the large benchmarks, SDSI with pruning identifies subgames that are on average 80% of the size of the parent game. Given the suspected non-polynomial complexity of parity game problems, the 25% decrease in size can even result in more than 25% decrease in runtime.

ACKNOWLEDGMENTS

I want to express my appreciation for the help I got from my supervisors dr. Tom van Dijk and Matthew Maat MSc. They aided me in my research by guiding me towards interesting papers and engaging in useful discussions about the subjects at hand. They were genuinely invested in my research and this gave me a lot of motivation.

REFERENCES

[1] Oliver Friedmann and Martin Lange. 2010. The PGSolver Collection of Parity Game Solvers Version 3. https://api.semanticscholar.org/CorpusID:17467844
[2] Oebele Lijzenga and Tom van Dijk. 2020. Symbolic Parity Game Solvers that Yield Winning Strategies. In *GandALF (EPTCS, Vol. 326)*. 18–32.
[3] Philipp J. Meyer and Salomon Sickert. 2022. Modernising Strix. https://api.semanticscholar.org/CorpusID:247080664
[4] Philipp J. Meyer, Salomon Sickert, and Michael Luttenberger. 2018. Strix: Explicit Reactive Synthesis Strikes Back!. In *CAV (1) (Lecture Notes in Computer Science, Vol. 10981)*. Springer, 578–586.
[5] Thibaud Michaud and Maximilien Colange. 2018. Reactive Synthesis from LTL Specification with Spot. In *Proceedings Seventh Workshop on Synthesis, SYNT@CAV 2018 (Electronic Proceedings in Theoretical Computer Science, Vol. xx)*. xx.
[6] Guillermo Perez. 2022. SYNTCOMP 2022 Results | The Reactive Synthesis Competition. http://www.syntcomp.org/syntcomp-2022-results/
[7] Florian Renkin, Philipp Schlehuber, Alexandre Duret-Lutz, and Adrien Pommellet. 2021. Improvements to `ltlsynt`. Presented at the SYNT'21 workshop, without proceedings..
[8] Tom van Dijk. 2018. Attracting Tangles to Solve Parity Games. In *CAV (2) (Lecture Notes in Computer Science, Vol. 10982)*. Springer, 198–215.
[9] Tom van Dijk. 2018. Oink: An Implementation and Evaluation of Modern Parity Game Solvers. In *TACAS (1) (Lecture Notes in Computer Science, Vol. 10805)*. Springer, 291–308.
[10] Tom van Dijk, Feije van Abbema, and Naum Tomov. 2024. Knor: reactive synthesis using Oink. *(In submission)* (2024).
[11] Wieslaw Zielonka. 1998. Infinite Games on Finitely Coloured Graphs with Applications to Automata on Infinite Trees. *Theor. Comput. Sci.* 200, 1-2 (1998), 135–183.