Modifying tangle learning to find smaller solutions

ROB KOOIKER, University of Twente, The Netherlands

LTL-synthesis is the process of converting a linear temporal logic (LTL) specification to a circuit. One way to synthesise a circuit is to convert the specification to a parity game, and to convert the solution of the parity game to a circuit. Because a parity game can admit many different solutions the resulting circuit can also be different. Ideally this circuit is as small as possible because smaller circuits are more efficient than bigger circuits. One algorithm that solves parity games is tangle learning. A downside of this algorithm is that it often finds worse solutions than other solvers. We implement two algorithms that aim to increase the quality of solutions from tangle learning and we benchmark the algorithms to verify whether or not they actually find higher quality solutions. We also perform an analysis comparing different parity games, and parity game solvers

Additional Key Words and Phrases: Parity games, Tangle learning

1 INTRODUCTION

During the yearly reactive synthesis competition (SYNTCOMP) many researchers compete for the best reactive synthesis tool [5]. Reactive synthesis is the process of automatically generating a state machine such that it satisfies a high-level specification [10]. One type of specification is linear temporal logic (LTL). LTL is a branch of logic that reasons about how a system changes over time. Interesting applications of LTL include the verification of certain properties, like whether something bad will never happen in a system (safety), or whether something good will eventually happen in a system (liveness) [1].

An LTL-specification can be translated to a controller. One approach to synthesise a controller from an LTL-specification is to convert the specification to a parity game, and to convert the solution of the parity game to a controller. This method was deemed infeasible until recently because there was no efficient method to carry out the conversion [10].

In 2020 the parity game track was added to SYNTCOMP [5]. The goal of this track is to synthesise an AIGER circuit from a parity game specification. In this track, there were three tools that competed against each other. These tools are: ltlsynt, Strix, and Knor. The fastest solver for this track was Knor, configured to use the tangle learning algorithm for solving the parity game. Despite its exceptional performance on the speed based metrics, the solver and algorithm developed by Van Dijk did not convince in the quality based metrics. It scored much lower than its competitors. This is further supported by other benchmarks [10], which also show the poor performance of tangle learning on providing good solutions.

An AIGER circuit is an and-inverter graph (AIG) presented in the AIGER format [2]. An AIG is simply a circuit that consist of only AND gates, and inverters. AIGER circuits have many practical applications. These applications include model checking and equivalence checking [4]. An AIGER circuit can also be converted into a physical circuit. Being able to find a better solution is important because a better solution leads to a smaller AIGER circuit. Smaller AIGER circuits are more efficient than bigger circuits, as the physical circuit is faster, and consumes less power. Model checking and equivalence checking also benefit from smaller circuits, as any algorithm can run faster on a smaller circuit.

In this study we explore why tangle learning is unable to find smaller strategies, and we propose possible modifications to the algorithm. We also implement the proposed modifications and benchmark the effect they have on the size of resulting solutions. The main research question we answer in this paper is: *How can tangle learning be modified such that it finds smaller solutions?*

In section 2 (*related work*) we shortly discuss the previous research that was conducted with regards to this topic. In section 3 (*preliminaries*) we provide some information on parity games and tangle learning to familiarise the reader with these topics. We also define some important terms that we will use in the rest of the paper. In section 4 (*methodology*) we explain how we approached the problem. This section also provides the proposed modifications to tangle learning, and some details on how to implement them. Section 5 (*findings*) is where we will show the results of our research. In this section we also interpret the results, and explain their significance. Section 6 (*conclusion*) is where we summarise our findings. This is also the section in which we provide a concise answer to our research question. Lastly, in section 7 (*discussion and future work*) we provide some topics that we think are interesting to look into further. We also discuss some shortcomings of this research, and whether or not these will have impacted our results.

2 RELATED WORK

In 2018 Van Dijk published a paper in which he initially proposes the tangle learning algorithm [7]. This paper details the implementation of the algorithm, how it works, and why it works. Van Dijk later made some improvements to the algorithm. Some details on the new version are found in the paper by Van der Veen [6]. In the introduction the tool Knor was briefly mentioned. Knor is a symbolic synthesis tool for HOA parity automata [10]. Knor uses tangle learning to solve the parity game it generates from the HOA automata. The paper by Van Dijk [10] provides more context on Knor. Knor does not directly implement tangle learning, but instead relies on Oink. Oink is a tool that provides multiple implementations of modern parity game solvers, including multiple variants of tangle learning. More information on Oink can be found in the papers by Van Dijk from 2018 and 2024 [8, 9].

This paper is not the first to cover the topic of reducing the size of strategies from tangle learning. A previous paper by Dijkstra attempted to reduce the size of the strategy by solving subgames [3]. Dijkstra mentions that this approach was not fruitful, and that more research was needed on the topic.

TScIT 41, July 5th, 2024, Enschede, The Netherlands

3 PRELIMINARIES

3.1 Parity game

A parity game is a game that is played on a finite directed graph. The game is played by two players, Odd and Even. A node on this graph is associated with a player, and a value (priority). The players move a shared token along the edges of the graph. The next move is determined by the associated player of the current node of the token. The players both take an infinite number of turns. The node with the highest priority that is passed infinitely often determines the winner. If the priority on that node is even, player Even wins. If the priority is odd, player Odd wins [3, 6–9, 11]. More formally, a parity game G consists of a set of Nodes V. The nodes belonging to a certain player are denoted by V_{even} or V_{odd}. The edges which connect the nodes is given by the set $E \subseteq V \times V$. The successors of a given node v are given by $E(v)$. After solving the game, the strategy for a given node is given by the function $strategy : V \rightarrow V$. The set of all strategies is denoted by S. Because we focus on the applications of parity games in LTL-synthesis, and not just parity games in general, this definition can be extended with an initial node V_0. This node follows from the original specification, and represents the initial state of the system.

3.2 Tangle learning

Many algorithms for solving parity games use the concept of attractors to find winning regions [7]. An attractor is a set of nodes, for which a certain player can force a visit to some other set of nodes, where the other set of nodes is a subset of the attractor. Tangle learning is an attractor based algorithm that extends the regular attractor with something known as a tangle attractor [6, 7]. A tangle is a strongly connected component within the graph, where one of the players has a strategy to win all cycles within the tangle. A tangle from which the opponent cannot escape is known as a dominion. The tangle attractor attracts the same nodes as a regular attractor, with the addition of all nodes in a tangle, where the player that does not win the tangle can only escape to the attractor. There are multiple variants of tangle learning. This paper is based on a variation called recursive tangle learning, which unlike normal tangle learning is implemented to use recursion. Because the different variations of tangle learning are still very similar to each other it is expected that the results of this paper will generalise to also be relevant to the other variations.

3.3 Solution quality

The solution to a parity game is not inherently better than another solution. Some applications do require a solution to have certain properties. Since this paper focuses on parity games in the process of LTL-synthesis we define a better solution to be one that minimises the size of the resulting controller. As mentioned in the introduction a smaller circuit consumes less power, and is faster than a bigger circuit. A downside to this metric is that it also depends on the method of converting the solution to a circuit, which makes it hard to compare between synthesis tools. Instead, we will use the amount of nodes that can be reached from the initial node as a metric. In general being able to reach fewer nodes from the initial node also leads to a smaller circuit. This paper uses the metric of reachable nodes as provided by the default configuration of Knor. We will also provide the size of the generated AIGER circuit sometimes, because two different strategies with the same size can still have a different performance, and in the end this is the metric that is the most important. For this metric we provide three different values, namely the one that is provided by the default configuration of Knor, and also the ones that we get when running Knor with parameters –onehot and –sop [10]. In the default configuration Knor uses a binary encoding to generate the AIG. With the parameter –onehot Knor will use one-hot encoding, and with parameter –sop Knor will use the sum of products format to generate the AIG.

4 METHODOLOGY

This research can be divided into two parts. The first part consists of an analysis of parity games, and parity game solvers, and the second part consists of implementations of multiple modifications to tangle learning.

4.1 Analysis

The goal of the analysis was to get a broader understanding of parity games, and parity game solvers. A secondary goal was to create a small set of games that are interesting to use as a benchmark. The analysis started out with all 288 games from the benchmark from SYNTCOMP 2023 [5], and four of the solvers available in oink, namely: *Distraction Fixpoint Iteration (fpi)*, *Priority promotion (npp)*, *Recursive Tangle Learning (rtl)*, and *Zielonka's recursive algorithm (zlk)*. All parity games that were not realisable (the initial node was won by player Odd) were discarded, as these games are not interesting for reactive synthesis. We used the remaining 208 games during the rest of the analysis.

For the first component of the analysis we ranked all parity games based on the variance in solution quality between the different solvers. Games with a high variance are likely to admit various different solutions, and are therefore the most interesting when benchmarking a new solver, or a variant of a solver. A game with low variance is likely to have the same solution regardless of the solving method, and is therefore not interesting when benchmarking. After this, we selected the twenty games with the highest variance to use in the rest of the research as a benchmark, because these are likely to be the most interesting.

For the second component of the analysis we ranked all selected solvers based on their robustness. We define a robust solver as one that is able to consistently provide similar quality solutions, regardless of the order of the input. Oink uses an input file to generate the parity game. Changing the order in which the nodes appear in the file does not change the game. It can still impact the order in which the game is solved. Because the game is still the same after shuffling the order of the nodes a solver should be able to provide the same solution. In practice, changing the order can affect certain parts of a solver, such as the order in which nodes are attracted. This can impact the quality of the solution. It is important to know whether a solver is robust or not, since a solver that is not robust is very susceptible to the order of the input, and will therefore not always generate a good solution, even if it is able to find one for that particular game. The robustness of the solvers was measured by

making each solver solve each one of the 208 realisable games ten times with the order of the nodes shuffled each time, and summing the variance of the results of each game. Next to that we summed up the variance for the games that were previously selected as interesting, because we expected that these have a relatively high stake in the overall variance per solver. Important to note here is that after randomly shuffling the order of the nodes in the input Oink sorts them based on priority. This is done because many of the solvers rely on the nodes to be in this order. Because of that it is not possible to truly randomise the order.

4.2 Implementations

We measured the performance in terms of solution size for two new algorithms, with the aim that they would find smaller solutions than tangle learning. The implementation of the first algorithm, recursive tangle attraction, was provided by our supervisor, Tom van Dijk. The second algorithm, strategy rewriting, was implemented by us.

4.2.1 *Recursive tangle attraction.*

Recursive tangle attraction works by removing the head (highest priority node) of a region and recursively entering the region and solving this part of the game. The idea is that we will find new, smaller tangles and attractors by doing this, which might also reduce the total size of the solution. The implementation of this algorithm can be found on GitHub[1].

4.2.2 *Strategy rewriting.*

Strategy rewriting is based on the research by Dijkstra [3]. In his research he attempts to find subgames, and solve these to find smaller solutions. He proposes two types of algorithms to find subgames, and states that solving subgames does not lead to smaller solutions. He proposes a pruning algorithm, and a growing algorithm. Strategy rewriting modifies his growing algorithm to reduce the size of an existing subgame. Dijkstra's growing algorithm starts out with a set of nodes that is not a subgame. The algorithm then repeatedly looks for nodes that can be added to the set, until a subgame is found. Our algorithm has one key distinction. When our algorithm adds a node to the set that will form the subgame, and that node belongs to the player that wins the subgame, we update that nodes strategy to a node that is also in the set, if possible. The node that was the strategy might now not be reachable from the initial node anymore. If that is the case, the size of the solution is reduced. Important to note is that the algorithm does not guarantee that the new solution is valid. To counteract this the new solution has to be verified, and if it is not valid the old solution has to be used.

The main rewrite algorithm can be found in Algorithm 1. Before this algorithm can be used the original game needs to be solved. This algorithm is not ran on the full game, but instead the subgame that is created by removing all nodes that are not reachable from the initial node in the solution. The parameter *nodes* should be equal to the initial node when the function is called. In line 13 of the algorithm we update the strategy to a successor of the current node, where the successor must have even parity. We only do this if such a successor exists, and we have not previously changed the strategy. This step is arbitrary, but in practice drastically reduces the amount of invalid solutions given by this algorithm. This step might also cause the resulting solution to be bigger than the one that was initially found. In this case it is possible to use the initial solution. We did not implement this, so that we can analyse how often this occurs, and how much it impacts the size of the solution.

Because strategy rewriting is a post processing step, to be executed after the solver finds a strategy, it is not limited to tangle learning, and can also be used with other solvers. For solvers that already provide good solutions it will likely not have much effect, or maybe even worsen the solution. When used with solvers that provide relatively bad solutions the benefits will probably be comparable to those we find on tangle learning.

Algorithm 1 Strategy rewriting

```
1:  function Rewrite(nodes)
2:      hasNew ← false
3:      for node ∈ nodes do
4:          if node ∈ V_odd then
5:              newNodes ← E(node) - nodes
6:              if newNodes ≠ ∅ then
7:                  hasNew ← true
8:              end if
9:              nodes ← nodes ∪ newNodes
10:         end if
11:         if node ∈ V_even then
12:             if strategy(node) ∈ nodes then
13:                 continue
14:             end if
15:             strat ← nodes ∩ E(node)
16:             if strat = ∅ then
17:                 strat ← { v ∈ E(node) | priority(v) % 2 = 0 }
18:             end if
19:             if strat = ∅ then
20:                 continue
21:             end if
22:             S_node ← strat_0
23:             nodes ← nodes ∪ strat
24:             hasNew ← true
25:         end if
26:     end for
27:     if hasNew then
28:         Rewrite(nodes)
29:     end if
30: end function
```

[1] https://github.com/trolando/oink/blob/01e86856842ec02aff897a00a34519502c9fd497/src/rtl.cpp

Table 1. The variance, minimum and maximum of the size of the solution per parity game

	Game	Variance	Min	Max	Best solver(s)
1	full_arbiter_8.tlsf.ehoa	100856	1688	2451	fpi
2	generalized_buffer.tlsf.ehoa	28436	608	1048	fpi
3	genbuf2.tlsf.ehoa	28273	600	987	fpi
4	full_arbiter_7.tlsf.ehoa	24765	718	1153	fpi
5	full_arbiter_unreal3.tlsf.ehoa	13736	249	540	fpi
6	ltl2dba08.tlsf.ehoa	3613	239	405	rtl
7	full_arbiter_6.tlsf.ehoa	3018	319	465	fpi
8	lilydemo21.tlsf.ehoa	867	6	74	fpi, npp, zlk
9	full_arbiter_5.tlsf.ehoa	442	143	170	fpi
10	ltl2dpa03.tlsf.ehoa	220	56	95	rtl
11	lilydemo17.tlsf.ehoa	193	19	53	fpi, npp
12	lilydemo20.tlsf.ehoa	67	3	23	fpi, zlk
13	round_robin_arbiter_unreal3.tlsf.ehoa	67	58	78	fpi, zlk
14	full_arbiter.tlsf.ehoa	27	16	29	fpi
15	full_arbiter_unreal1.tlsf.ehoa	27	16	29	fpi
16	full_arbiter_unreal2.tlsf.ehoa	27	16	29	fpi
17	full_arbiter_4.tlsf.ehoa	21	66	78	fpi
18	ltl2dpa23.tlsf.ehoa	19	5	16	rtl
19	ltl2dpa24.tlsf.ehoa	19	5	16	rtl
20	SPIPureNext.tlsf.ehoa	18	38	49	rtl

Table 2. The variance of solution size per solver per game

	Game	Variance fpi	Variance npp	Variance rtl	Variance zlk
1	full_arbiter_8.tlsf.ehoa	0	70	732	1049
2	generalized_buffer.tlsf.ehoa	0	3624	495	1411
3	genbuf2.tlsf.ehoa	0	2515	727	981
4	full_arbiter_7.tlsf.ehoa	0	41	986	1271
5	full_arbiter_unreal3.tlsf.ehoa	0	2575	7042	7361
6	ltl2dba08.tlsf.ehoa	0	120	1831	33
7	full_arbiter_6.tlsf.ehoa	0	20	61	211
8	lilydemo21.tlsf.ehoa	0	416	0	416
9	full_arbiter_5.tlsf.ehoa	0	2	68	37
10	ltl2dpa03.tlsf.ehoa	0	33	65	13
11	lilydemo17.tlsf.ehoa	0	1	203	107
12	lilydemo20.tlsf.ehoa	0	2	32	0
13	round_robin_arbiter_unreal3.tlsf.ehoa	0	28	21	13
14	full_arbiter.tlsf.ehoa	0	3	10	4
15	full_arbiter_unreal1.tlsf.ehoa	0	3	12	10
16	full_arbiter_unreal2.tlsf.ehoa	0	3	12	10
17	full_arbiter_4.tlsf.ehoa	0	0	16	5
18	ltl2dpa23.tlsf.ehoa	0	22	8	0
19	ltl2dpa24.tlsf.ehoa	0	29	9	0
20	SPIPureNext.tlsf.ehoa	0	7	5	8

Table 3. The sum of variance of the size of solutions per game, where the order of the game was randomly sorted and solved ten times, per solver

Solver	All games	Selected games
fpi	0	0
npp	9745	9523
rtl	13150	12344
zlk	13062	12946

5 FINDINGS

5.1 Analysis

5.1.1 *Variance.*

Table 1 shows the twenty games with the most variance in solution quality. The table also shows the minimum and maximum size of the strategies given by the solvers, and the solver which had the best (smallest) strategy. In case multiple solvers found the smallest strategy all of those are listed. Interesting to note is that the variance appears to be heavily dependent on the size of the game. Because the first few games are much larger than the last few this implies that even in this list the first few games are a lot more interesting than the last few games. Another interesting finding is that out of these twenty games there are five games on which rtl actually has the best performance. Out of these five games three are from the ltl2dpa family, and one is from ltl2dba family. We think that these games probably have a certain property that makes rtl perform well compared to other solvers. Because these games are closely related we think that they might even share the same property. We were not able to find what this property is.

Not visible in Table 1, but still interesting is that 119 out of 208 games showed no variance at all. Upon inspection most of these games are relatively small in size, and therefore we think that they do not admit many different solutions. The SYNTCOMP benchmark was initially composed to measure the performance of the whole reactive synthesis process, and not just to measure the performance of different parity game solvers, so we could have expected that some of the games it includes are not interesting for this purpose.

5.1.2 *Randomness.*

Table 3 shows how randomising the order of the nodes in the parity game specification file impacts the results of the different solvers. From the comparison between all realisable games and the twenty selected games we can see that most of the variance comes from the twenty selected games. Upon further inspection, using Table 2 we can see that there is one game that causes most of the variance. This game is full_arbiter_unreal3.tlsf.ehoa. We were not able to find a direct cause for this.

Interesting about this data is that better solvers appear to have a lower variance in their results. This makes sense, since a solver that is able to generally find good solutions cannot have much variance, as that implies that it will also regularly find a bad solution.

5.2 Implementations

5.2.1 *Recursive tangle attraction.*

In Table 6 the results of recursive tangle attraction on the twenty selected games are shown. Both the size of strategies, and the size of the circuit after synthesis are shown. The initial size refers to the size given by the original rtl implementation, while the new size refers to the size with recursive tangle attraction. What can be seen here is that there are no major differences between the initial size and the new size. The only changes visible here are in ltl2dba08.tlsf.ehoa, which has a slightly smaller circuit, and ltl2dpa03.tlsf.ehoa, which is quite a bit bigger than before. Furthermore, Table 4 shows that indeed the new algorithm often leads to the same strategy, and more often leads to a worse strategy than a better strategy. We think that this algorithm likely does not work because the previously removed nodes still need to be solved, and are very likely to still end up in the same region as they would have been in otherwise.

Table 4. The amount of games on which recursive tangle attraction performs worse, better and equal, compared to rtl

Comparison type	Worse	Better	Equal
strategy size	12	6	190
circuit size binary	11	3	194
circuit size sop	12	6	190
circuit size one-hot	12	6	190

Table 5. The amount of games on which strategy rewriting performs worse, better and equal, compared to rtl

Comparison type	Worse	Better	Equal
strategy size	11	24	173
circuit size binary	24	33	151
circuit size sop	24	32	152
circuit size one-hot	26	31	151

5.2.2 *Strategy rewriting.*

Table 7 shows the initial and new size of strategies and circuits, generated using rtl, with and without strategy rewriting. What we see here is that there are many cases in which strategy rewriting performs better than plain rtl. The difference between results is sometimes relatively small, such as with lilydemo20.tlsf.ehoa, but it can also be relatively big, such as with lilydemo21.tlsf.ehoa. There is also a case where strategy rewriting leads to a bigger solution and circuit. This is ltl2dpa03.tlsf.ehoa. From the more general overview provided by Table 5 we can see that especially the strategy size is better with strategy rewriting. This is of course the main goal of the algorithm, so it is not a surprise. What is interesting is that the circuit size is different in more games than the games where the strategy size is different. This implies that that there is also a subset of games in which the solution is not improved, but merely changed without reducing the strategy size. This table also shows that there are still quite some cases where the new solution is worse than the old one. As mentioned in section 4 it is possible to use the original solution in this case. Something to note is that the verifier found all new solutions found by our algorithm to be valid. This means that out of 208 solutions none were invalid. Since the algorithm does not guarantee that the solution is valid this is somewhat surprising.

Table 6. The size of a solution and circuit with and without recursive tangle attraction

		Solution size		circuit size binary		circuit size sop		circuit size one-hot	
	Game	**Initial**	**New**	**Initial**	**New**	**Initial**	**New**	**Initial**	**New**
1	full_arbiter_8.tlsf.ehoa	2451	2451	238515	238515	71815	71815	84292	84292
2	generalized_buffer.tlsf.ehoa	1048	1048	56682	56682	6993	6993	10951	10951
3	genbuf2.tlsf.ehoa	987	987	47388	47388	6400	6400	9499	9499
4	full_arbiter_7.tlsf.ehoa	1153	1153	87741	87741	25843	25843	30877	30877
5	full_arbiter_unreal3.tlsf.ehoa	540	540	12450	12450	1705	1705	2702	2702
6	**ltl2dba08.tlsf.ehoa**	239	239	**11411**	**11355**	**2509**	**2458**	**4297**	**4262**
7	full_arbiter_6.tlsf.ehoa	465	465	25309	25309	7405	7405	8835	8835
8	lilydemo21.tlsf.ehoa	74	74	864	864	233	233	307	307
9	full_arbiter_5.tlsf.ehoa	170	170	4995	4995	1555	1555	1809	1809
10	**ltl2dpa03.tlsf.ehoa**	**56**	**70**	**1424**	**1820**	**296**	**338**	**514**	**587**
11	lilydemo17.tlsf.ehoa	53	53	740	740	185	185	241	241
12	lilydemo20.tlsf.ehoa	23	23	298	298	78	78	94	94
13	round_robin_arbiter_unreal3.tlsf.ehoa	78	78	1113	1113	289	289	357	357
14	full_arbiter.tlsf.ehoa	29	29	324	324	122	122	132	132
15	full_arbiter_unreal1.tlsf.ehoa	29	29	324	324	122	122	132	132
16	full_arbiter_unreal2.tlsf.ehoa	29	29	324	324	122	122	132	132
17	full_arbiter_4.tlsf.ehoa	78	78	2003	2003	614	614	731	731
18	ltl2dpa23.tlsf.ehoa	5	5	75	75	22	22	23	23
19	ltl2dpa24.tlsf.ehoa	5	5	75	75	22	22	23	23
20	SPIPureNext.tlsf.ehoa	38	38	386	386	85	85	135	135

Table 7. The size of a solution and circuit with and without strategy rewriting

		Solution size		circuit size binary		circuit size sop		circuit size one-hot	
	Game	**Initial**	**New**	**Initial**	**New**	**Initial**	**New**	**Initial**	**New**
1	**full_arbiter_8.tlsf.ehoa**	**2451**	**1815**	**238515**	**130816**	**71815**	**49470**	**84292**	**53701**
2	generalized_buffer.tlsf.ehoa	1048	1048	56682	56682	6993	6993	10951	10951
3	genbuf2.tlsf.ehoa	987	987	47388	47388	6400	6400	9499	9499
4	**full_arbiter_7.tlsf.ehoa**	**1153**	**854**	**87741**	**48783**	**25843**	**15933**	**30877**	**17726**
5	full_arbiter_unreal3.tlsf.ehoa	540	540	12450	12450	1705	1705	2702	2702
6	**ltl2dba08.tlsf.ehoa**	**239**	**212**	**11411**	**10484**	**2509**	**2232**	**4297**	**3647**
7	**full_arbiter_6.tlsf.ehoa**	**465**	**343**	**25309**	**14259**	**7405**	**4772**	**8835**	**5325**
8	**lilydemo21.tlsf.ehoa**	**74**	**6**	**864**	**109**	**233**	**40**	**307**	**36**
9	**full_arbiter_5.tlsf.ehoa**	**170**	**158**	**4995**	**4702**	**1555**	**1530**	**1809**	**1766**
10	**ltl2dpa03.tlsf.ehoa**	**56**	**63**	**1424**	**1655**	**296**	**329**	**514**	**532**
11	lilydemo17.tlsf.ehoa	53	53	740	740	185	185	241	241
12	**lilydemo20.tlsf.ehoa**	**23**	**22**	**298**	**288**	**78**	**77**	**94**	**95**
13	round_robin_arbiter_unreal3.tlsf.ehoa	78	78	1113	1113	289	289	357	357
14	full_arbiter.tlsf.ehoa	29	29	324	324	122	122	132	132
15	full_arbiter_unreal1.tlsf.ehoa	29	29	324	324	122	122	132	132
16	full_arbiter_unreal2.tlsf.ehoa	29	29	324	324	122	122	132	132
17	**full_arbiter_4.tlsf.ehoa**	**78**	**66**	**2003**	**1586**	**614**	**497**	**731**	**586**
18	ltl2dpa23.tlsf.ehoa	5	5	75	75	22	22	23	23
19	ltl2dpa24.tlsf.ehoa	5	5	75	75	22	22	23	23
20	SPIPureNext.tlsf.ehoa	38	38	386	386	85	85	135	135

6 CONCLUSION

We explored different approaches to modify tangle learning to find smaller solutions. First we performed an analysis of parity games, and parity game solvers, and then we implemented two new algorithms and benchmarked their effectiveness. During the analysis we found that many of the games in the SYNTCOMP benchmark do not seem to be interesting, as all solvers we tested with found a solution of similar quality. Often the solvers even found a solution of exactly the same quality. Based on our findings we selected a set of twenty games, which we found the most likely to be interesting for our research, and which might also be useful for other research which benchmarks different parity game solvers. These games are shown in Table 1. Next to that we found that most solvers find a different solution on some parity games if the order in which the nodes appear in the input is shuffled. We reasoned that a good solver needs to be resilient against reordering the nodes, while a bad solver can still be resilient, but does not have to be. Based on our implementations we found that recursively entering closed regions and attempting to attract tangles there does not lead to smaller solutions. We think this is because the nodes that were left out in the recursive search will likely later still be attracted to the same region.

Lastly, we found that rewriting the strategies found by tangle learning to find smaller subgames often decreases the size of the solution. Because of the way we implemented this it is also possible to find bigger or invalid solutions, but we found that this did not happen often.

7 DISCUSSION AND FUTURE WORK

For the analysis part of this research we chose four solvers to make a comparison. There are many more solvers available, and also many more families of solvers. For a more complete overview of the topics new research using more solvers would be required.

There are only a few relatively large games in the benchmark we used. Most games consist of only a few hundred nodes, and some are even as small as seven nodes. Because we show that games with more nodes tend to have a higher amount of possible solutions these are much more interesting for benchmarking the different solvers. The current benchmark was merely chosen for convenience, and future research could benefit from including more large games, with at least a few thousand or more nodes. However, as previously mentioned, the lack in variance for most games could also be because we only used four solvers for our comparison.

We also argue that a good solver needs to be robust, so that a change in the order of the nodes in the input does not impact the quality of the solution. We focused mainly on ways to directly improve the solutions that are found by tangle learning, but perhaps another approach can be to improve the robustness of the solver.

Twice in this paper we mention that we found potentially interesting occurrences, without being able to find out why that happens. These could also be interesting for future research. Firstly, we mention in section 5 that we were not able to find out why tangle learning appears to perform well on ltl2dba and ltl2dpa games. Secondly, we found that the solutions to the full_arbiter_unreal3 game appear to vary a lot depending on the order of the nodes in the input specification.

Based on the success of our strategy rewriting algorithm we suggest a modified version for future research, that might work better. This version works by employing a greedy algorithm, which goes over all nodes that are won by player Even, and attempting all possible strategies per node, and keeping the strategy that immediately improves the current solution, without having to change another strategy. This new method might decrease the size of the solutions even more, since it goes over all nodes, while our current implementation often only looks at a small part of the nodes.

8 ACKNOWLEDGMENTS

I thank my supervisor, Tom Van Dijk, for his useful feedback throughout this research. I would also like to thank him for providing the implementation of the recursive tangle attraction algorithm, which ensured that I could direct my efforts to other parts of the research.

REFERENCES

[1] Saman Arzaghi. 2021. An Introduction To Linear Temporal Logic (LTL). (02 2021).
[2] Armin Biere. 2007. The AIGER And-Inverter Graph (AIG) Format Version 20071012. https://fmv.jku.at/aiger/FORMAT.aiger
[3] Stijn Dijkstra. 2024. Finding Smaller Parity Game Solutions by Identifying and Solving Subgames using Oink. https://essay.utwente.nl/98288/1/dijkstra_BA_eemcs.pdf
[4] A. Kuehlmann, V. Paruthi, F. Krohm, and M.K. Ganai. 2002. Robust Boolean reasoning for equivalence checking and functional property verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 21, 12 (2002), 1377–1394. https://doi.org/10.1109/TCAD.2002.804386
[5] Guillermo Perez. 2023. SYNTCOMP 2023 Results. http://www.syntcomp.org/syntcomp-2023-results/
[6] Suzanne Ellen van der Veen. 2024. A Formal Proof for the Correctness of Tangle Learning. https://essay.utwente.nl/98665/1/Veen_van_der_MA_EEMCS.pdf
[7] Tom van Dijk. 2018. Attracting Tangles to Solve Parity Games. In *CAV (2) (LNCS, Vol. 10982)*. Springer, 198–215. https://doi.org/10.1007/978-3-319-96142-2_14
[8] Tom van Dijk. 2018. Oink: An Implementation and Evaluation of Modern Parity Game Solvers. In *TACAS (1) (LNCS, Vol. 10805)*. Springer, 291–308. https://doi.org/10.1007/978-3-319-89960-2_16
[9] Tom van Dijk. 2024. Reproducing parity game algorithms in Oink. https://www.tvandijk.nl/pdf/2024rrrr_oink.pdf
[10] Tom van Dijk, Feije van Abbema, and Naum Tomov. 2024. Knor: reactive synthesis using Oink. In *TACAS (1) (Lecture Notes in Computer Science, Vol. 14570)*. Springer, 103–122.
[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.

A USE OF ARTIFICIAL INTELLIGENCE

During the preparation of this work the author(s) used Scribbr in order to generate and format references. After using this tool/service, the author(s) reviewed and edited the content as needed and take(s) full responsibility for the content of the work.