Performing Bisimulation Minimisation To Parity Game Strategies To Improve Controller Quality

FEIJE VAN ABBEMA, University of Twente, The Netherlands

Reactive synthesis is the process of creating a controller out of a high-level specification. Recently, new research created a way of converting a linear temporal logic specification to an and-inverter graph. In this process, a parity game is created and solved to obtain a strategy which is directly translated into an and-inverter graph. However, the strategy of the parity game could have some redundant states. Reducing the number of states will result in a smaller graph and therefore a smaller and more efficient controller. This paper investigates how much parity game strategies can be reduced in the reactive synthesis process. Around half of the strategies can be reduced in size and when reduction is possible on average 28% of the strategy is reduced.

Additional Key Words and Phrases: automata, parity games, bisimulation, bisimulation minimisation, partition refinement, binary decision diagrams

1 INTRODUCTION

Using Linear Temporal Logic (LTL) you can construct a specification that captures the behaviour of a system. It is desirable to automatically create a controller only by specifying a LTL specification since this controller is a physical object that adheres to the LTL specification. This process can be done using reactive synthesis. Abraham [1] created a new way to synthesise a controller out of a LTL specification in his master thesis.

The process of reactive synthesis is complex. The scope of this paper is to improve a specific part of this process. But before that will be discussed, a few concepts need to be explained. Firstly, ω-automata and the parity game and secondly bisimulation will be explained.

1.1 Parity games and bisimulation

ω-automata are automata that run on an infinite input of symbols instead of a finite input of symbols. A run on an ω-automaton is an infinite sequence of states such that for each symbol in the word there is a state transition that can be taken to the next state e.g. the word can run through the automaton. Furthermore, instead of a set of accepting states, an acceptance condition is used to determine if a run is accepted. This acceptance condition depends on the type of ω-automata. [3]

A game is a type of ω-automata. In a game, an arena and a winning condition are used. An arena is defined as follows: $A = (V_0, V_1, E)$ where V_0 is the 0-vertices, V_1 is the 1-vertices disjoint from V_0 and $E \subseteq (V_0 \cup V_1) \times (V_0 \cup V_1)$ is the edge relation. This automaton is controlled by two players. Player 0 controls V_0 and player 1 controls V_1. A play on a game is then a run through the automaton where the players make the transitions on the vertices that they control. [3]

TScIT 37, July 8, 2022, Enschede, The Netherlands

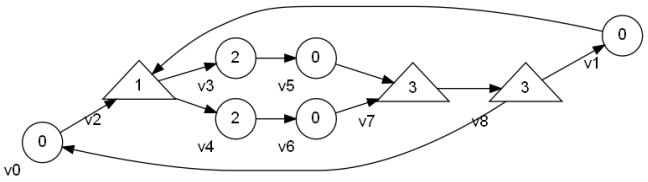

Fig. 1. An example parity game

A parity game is a game with a specific winning condition. For the winning condition of a parity game, each state has to receive a colour i. A play is then winning for player 0 if and only if the lowest colour i in a play that is visited infinitely often is even. [3]

An example parity game can be found in fig. 1 where one player controls the vertices that are circles and the other player controls the vertices that are triangles.

Automata can be reduced in size by performing bisimulation minimisation. This is done by reducing a set of bisimilar states to only one state. There are multiple types of bisimulation [2] but for this paper, we will apply strong bisimulation as described by van Dijk et al. [6]. This is because the parity games do not have any internal state transitions (also called τ-transitions) and the other types of bisimulation are illogical to apply because they deal with internal state transitions. Strong bisimulation only deals with states and their observable state transitions and therefore this type of bisimulation will be used.

In this paper, bisimulation will refer to strong bisimulation (or a conjugation of it). A set of states are bisimilar if the states have the same behaviour, in other words, the transitions one state can make can also be made by the other state.

In fig. 1 there are some bisimilar states. For example vertices V_0 and V_1 are bisimilar. They both have only one outgoing transition to V_2. The same goes for V_5 and V_6 which both only have one outgoing transition to V_7. Consider that V_5 and V_6 are now replaced by a single state. It can then be seen that V_3 and V_4 both only have one transition to the same state and are also bisimilar. More formally, V_3 and V_4 both have a transition to a state that belongs to the bisimilar set of states which contains V_5 and V_6.

1.2 Related work

In the reactive synthesis process of Abraham [1], some optimisations and suggestions are made. One of these optimisations is to improve the quality of the controller. Abraham suggests that the quality of the controller can be improved by investigating if the number of states in the parity game can be reduced. He indicates applying bisimulation minimisation is the way to go.

Though Abraham indicates that bisimulation minimisation can be applied at various stages of the reactive synthesis, this research will only focus on applying it after the winning strategy for the

parity game is found. This strategy is directly translated in the and-inverter graph so optimisations will directly impact the controller quality.

Cranen et al. [2] already showed that parity games can be checked for bisimulation relations. Multiple bisimulations (including strong bisimulation) can be applied to parity games to find out if states are bisimilar. According to this research, we can conclude that the parity games in the reactive synthesis process can also be checked for bisimilarity. It can therefore be a way to reduce the size of the parity game.

Van Dijk [6] created a tool to apply bisimulation minimisation to labelled transition systems and continuous-time Markov chains. Though this tool is not usable for minimising parity games, the ideas for minimising through signature-based partition refinement are similar and will also be used in this paper.

1.3 Contribution

This paper will investigate how much parity games can be reduced in the reactive synthesis process. It is already known that states can be bisimilar but it is not known how much reduction can be performed if bisimulation minimisation is performed on a parity game created through Abraham's novel way of synthesis [1]. Therefore, this paper will answer the following question:

(1) How many redundant states can be removed if you apply bisimulation minimisation to the strategy of the parity game in the reactive synthesis process from a LTL specification to a controller?

1.4 Approach and structure

In order to answer this research question, the tool Knor will be used which will be explained in subsection 2.1. In Knor an algorithm will be created to transform the states into a list of blocks. Each block will then represent a set of bisimilar states. How this algorithm works is described in the rest of section 2. Finally, the number of states and amount of blocks are presented in the results. These results will be presented and discussed in section 3. The paper will be concluded in section 4.

2 MEASUREMENT ENVIRONMENT

2.1 Knor

Knor[1] is a tool to synthesise a controller out of Extended Hanoi Omega Automata (ehoa) files. These ehoa files are used as input for Knor. In an ehoa file, an automaton is described. This includes the kind of automaton, the atomic propositions (AP), which APs are controllable and the states with the transitions and APs that are taken. These files are parsed by Knor and put into a binary decision diagram.

A binary decision diagram (BDD) is a data structure that is used to represent a boolean function. At each level of the diagram, there will be a node and a variable belonging to that level. If the variable is false then the low edge of the BDD is followed to the lower level which usually is the edge that goes to the left. If the variable is true, the high edge is followed to the lower level which usually is the edge that goes to the right. In Fig. 2 the low edge is indicated with a dashed line while the high edge is indicated with a solid line.

BDDs can also be used for representing an automaton. Fig. 2 shows an example of how that works. It shows how the 9 states of fig. 1 are modelled in a BDD with their state transitions. In the software, the v0 to v8 labels are not added but for this figure, it is added for convenience. The label *true* means it is a true terminal and the relation exists in the parity game. Likewise, the label *false* means it is a false terminal and the relation does not exist. The BDD also simplifies the tree if all paths lead to the same result. For example, if one follows the low edge and then the high on V_0 it will result in a false terminal meaning that V_0 has no transition to V_4, V_5, V_6 and V_7. Another example is when the high edge is followed at the top of the BDD. This results directly in V_8 since all other vertices belong to the low edge.

Knor itself also uses the tools Sylvan and Oink. Sylvan[4] is used to represent the parity automata and parity games as BDDs and Oink[5] is used to solve parity games.

Knor parses an ehoa file and creates a symbolical parity game out of it. This parity game is encoded in a BDD. The structure of this BDD is shown on the left of fig. 3. It solves the game and gives a strategy for the parity game. This strategy tells for each reachable state in the game and each uncontrollable atomic proposition (UAP) possible in that state which controllable atomic proposition (CAP) has to be performed. This can also be seen in the top-right of fig. 3. From this strategy, for each state the BDD is calculated and used to create an And-Inverter Graph (AIG). Together with the BDDs for the CAPs, the AIG is made complete.

Reducing the size of the strategy BDD will result that less has to be modelled in the AIG. In the next subsection, the algorithm for reducing the size of the strategy is explained.

2.2 Algorithm: bisimulation

This algorithm[2] is adapted from van Dijk et al. [6]. In the adaption, the signature computation is different because the tree structures are not the same. Furthermore, this algorithm does not run in parallel while the algorithm of Dijk et al. does execute in parallel. Therefore, we will not be concerned with actions that ensure thread safety. The ideas of refining each state and creating new blocks when a new signature is found are still the same.

The goal of this algorithm is to divide all states among some blocks. Each block is then a set of bisimilar states. The algorithm can be found at alg. 1

[1]The tool can be found on https://github.com/trolando/knor

[2]The code can be found on https://github.com/FeijevanAbbema/knor_bisim

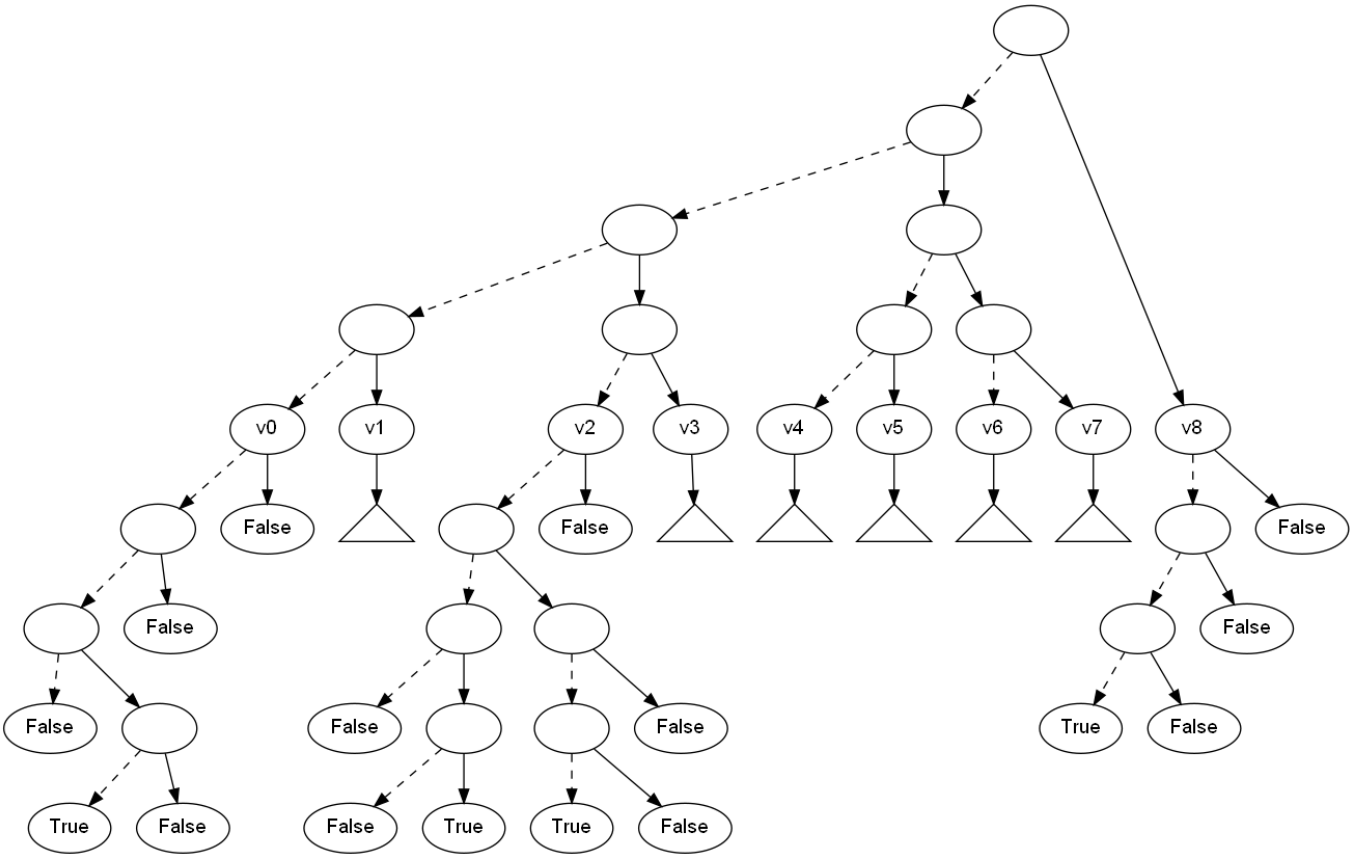

Fig. 2. The BDD representation of fig. 1. A dashed line indicates the low edge while a solid edge indicates the high edge. Furthermore, a triangle indicates that there is a BDD below it but for simplicity it is left out.

Algorithm 1. Compute bisimulation

```
1  def calculate_bisimulation:
2    blocks[0] = states;
3    while no_block != no_old_blocks:
4      new_blocks = blocks
5      for each block in blocks:
6        first_sig = true
7        known_signatures = []
8        for each state in block:
9          signature = compute_signature(state, [])
10         if first_sig:
11           known_signatures.append(signature)
12           first_sig = false
13         else if signature not in known_signatures:
14           new_blocks.append([state])
15           block.remove(state)
16           known_signature.append(signature)
17         else if known_signatures[0] != signature:
18           block.remove(state)
19           block_no = find_block(signature)
20           blocks[block_no].append(state)
21     blocks = new_blocks
22     no_blocks = new_blocks.size
```

The algorithm initialises by putting every state in one block (line 2). Then the algorithm will iteratively try to refine each block into the coarser partition (lines 5-20). This is done by calculating the signature of each state (line 9). A different signature within a block means that two states are not bisimilar. How the computation of the signature works will be described in subsection 2.3. The algorithm keeps track of every new signature that is found in a block (lines 7, 11, 16). The signature that is computed first, belongs to the block number that already exists (lines 10-12). If that is not the case but if a new signature is found, a new block is created and the state is added to the newly created block and removed from its former block (lines 13-16). If a state has a known signature but the signature does not belong to the first block, it belongs to a block recently created in this iteration. Therefore, it has to move to the new block (lines 17-20). Finally, after some iterations, the algorithm converges to the coarsest partition which means that the blocks cannot be refined any further (line 3). This means that each block is a set of bisimilar states.

2.3 Algorithm: signature computation

A signature of a state is used in the algorithm of bisimulation. This subsection will explain how the signature is computed.

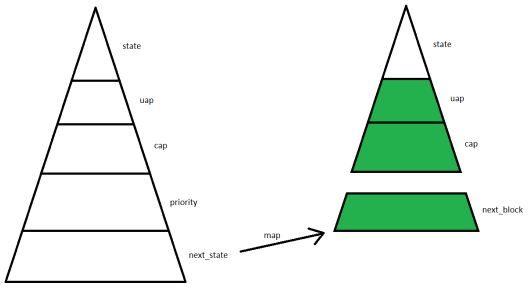

Fig. 3. The left BDD shows the structure of transition relation BDD. The right-top BDD shows the structure of strategy BDD. The right-bottom part is the block numbering. The green parts indicate what the signature consists of

The signature defines the behaviour of a state. It contains what uncontrollable atomic propositions (UAP) can be done and which controllable atomic propositions (CAP) are used in response. Furthermore, it matters to which block it will transition given a UAP and CAP. In fig. 3 a visual representation is shown of what the signature looks like in the BDD. Knor's strategy BDD (the one which is used to decide what the response of the controller will be) consists of a state layer, a UAP layer and a CAP layer. Therefore, for each state, the UAP and CAP in the BDD can be used for the signature. However, the next block number is not available in this BDD. This has to be determined by tracking down the same path in the transition relation BDD. This path will result in the next state (after falling through the priority part). This state can be mapped to the known state to block mapping to get the block number. The UAPs, CAPs and the next block numbers will be used to compute the signature. The pseudocode of the algorithm can be found at alg. 2

In order to compute the signature, the signature of the low edge and the high edge need to be known. Therefore, the signature of both the low edge and high edge will be computed recursively (lines 12 and 20) until a terminal is reached. A *false* terminal means the relation does not exist and a -1 will be recorded at location low_var or high_var (lines 7-8 and 15-16). The low_var and the high_var are variables that tell how deep the node in the tree is. A true terminal means the relation does exist and the corresponding block number has to be found (lines 9-10 and 17-18). As described above the strategy BDD does not contain which state will be next so this is found by tracking down the same path in the BDD containing all the relations. Therefore, the path is also kept track of to retrace the path in the transition relations BDD (lines 6, 13, 14, 21). Keep in mind that when *compute_signature* is called in calculate_bisimulation, the taken_path variable is initialised as an empty list.

Two signatures are equal if all transitions that are possible correspond to the correct block number. This means that the list of var and block number combinations correspond. Keep in mind that also the false terminals should match. If they do not match it means that one BDD does have a transition that the other BDD does not have and they are not bisimilar.

Algorithm 2. Compute signature

```
1  def compute_signature(bdd, taken_path):
2    low = bdd_getlow(bdd)
3    low_var = bdd_getvar(low)
4    high = bdd_gethigh(bdd)
5    high_var = bdd_getvar(high)
6    low_res, high_res = []
7    taken_path.append(false)
8    if (low == terminal_false):
9      low_res.append(low_var, -1)
10   else if (low == terminal_true):
11     low_res.append(low_var, get_block_no(
           taken_path))
12   else:
13     low_res = computeSignature(bdd, taken_path)
14   taken_path.pop_last()
15   taken_path.append(true)
16   if (high == terminal_false):
17     high_res.append(high_var, -1)
18   else if (high == terminal_true):
19     high_res.append(high_var, get_block_no(
           taken_path))
20   else:
21     high_res = computeSignature(bdd, taken_path)
22   taken_path.pop_last()
23   return low_res ++ high_res
```

3 RESULTS

In this section, the results will be presented and discussed. The results will be divided into sets depending on the size of the parity game strategy which will be further described in subsection 3.1. For the discussion of the results, the results of all the games will be discussed first, then each set will be discussed individually and finally, the relation between the sets will be discussed.

3.1 Gathering data

The algorithm that is described in section 2.2 returns a mapping from a block number to a set of states. This mapping is transformed into some data. First, the number of states is recorded. Then the number of blocks is recorded. Finally, for each block that exists the number of states is recorded. This is appended as a line to a CSV file.

This CSV file is then read in a python script[3]. This small script calculates the data that is represented in the table 1. The script divides each minimised game over 3 sets. The first set contains all the games that started with 1-10 states, the second set contains all games that started with 11-100 states and the last set contains all the games that contain more than 100 states. The reason that these games are divided is to investigate if the size of the game influences the calculated data.

For each set the script will calculate the following statistics:

- The maximal number of states that could be reduced
- The average number of states that could be reduced

[3]The code and CSV file can be found on https://github.com/FeijevanAbbema/knor_bisim

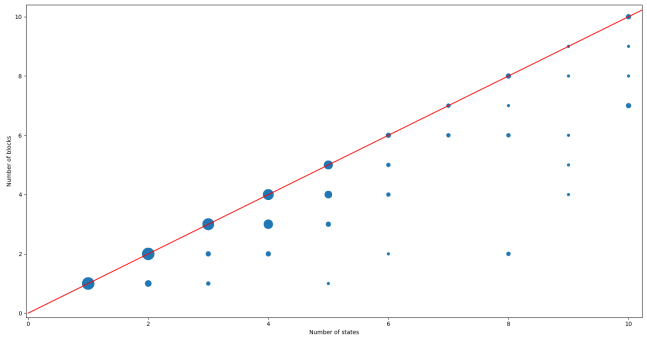

Fig. 4. Scatter plot of the set of small games. The x-axis has the number of states and the y-axis has the number of blocks. If a point is larger, there are more games with the same amount of states and blocks

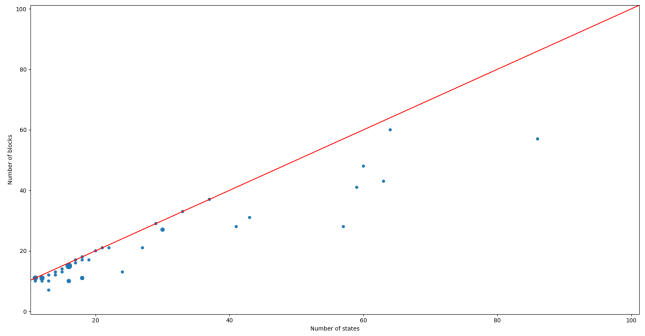

Fig. 5. Scatter plot of the set of medium games. The x-axis has the number of states and the y-axis has the number of blocks. If a point is larger, there are more games with the same amount of states and blocks

- The average number of states that could be reduced over the games where there was at least one block with more than one state
- The highest percentage of reduction
- The average percentage of reduction
- The average percentage of reduction over the games where there was at least one block with more than one state
- The size of the block with the most states in it
- The average size of the blocks
- The number of reduced games
- The percentage of reduced games
- The number of games

The average values are calculated as follows:

$$avg = \frac{\sum_{i=1}^{n} s_i}{n}$$

where s_i is the amount for game i and n is the number of games. Furthermore, if the average is calculated for games that have at least one block that has more than one state in it (e.g. where reduction is possible), s_i is the amount for game i where something can be reduced and n is the number of games where something can be reduced.

All these values will also be calculated for all the games (independent of which set they belong to).

The data that has been gathered can be found in table 1. The value in parentheses in the rows of maximal reduction and largest block indicate how large the parity game strategy originally was.

Furthermore, there are scatter plots of all the data that can be found in figures 4, 5 and 6. The set of the small games (with 1-10 states) is shown in fig. 4, the set of medium games (10-100 states) is shown in fig. 5 and the set of large games is shown in fig. 6. The red line in all plots indicates the line of no reduction (number of states=number of blocks).

3.2 All games

In total 209 games were processed. Of these games, only half of them were reduced in size. On one hand, this indicates that there is a reduction possible but on the other hand, it means that half of the games are not reducible and no improvement on them can be

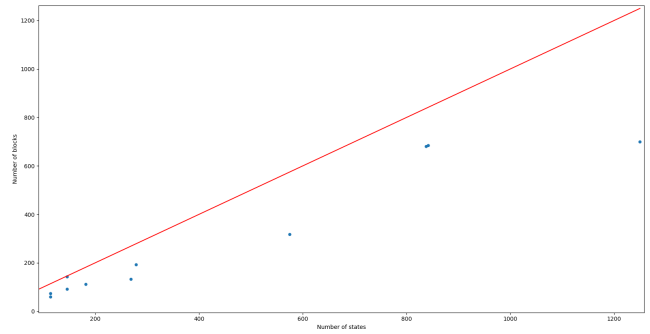

Fig. 6. Scatter plot of the set of large games. The x-axis has the number of states and the y-axis has the number of blocks

achieved. If however there is a possibility of reduction, the results are more promising, namely, on average 28.8% of the states are reduced. This greatly reduces the states and therefore greatly decreases the number of gates that are required for the controller.

Another interesting result is the size of the blocks. The largest block is 12 states in a game of 1250 states. This means that among all the games there are mostly small groups of states belonging to a bisimilar block. There are no large blocks that have a lot of states in them since the largest block of all games is 12 states. Adding the fact that 28.8% of the states are reduced (if there is at least some reduction) we can reason that to gain such a high percentage there are a lot of blocks containing a few states.

3.3 Small games

The set of the small games with 1-10 states is also the set with the most games. It concerns 72% of all games processed. The first noticeable thing is the low percentage of reduced games. Only 37.1% of the games could be reduced. One of the reasons that this happens is because there are games that are already reduced to only 1 state. And logically a game with 1 state cannot be reduced. There are 20 games recorded where this happens which is already 13% of the games. A speculation of why a part of the other 50% cannot be reduced could be that since the games are already small, the chance

	1-10 states	11-100 states	100+ states	All games
Maximal reduction of states	6 (8)	29 (86)	551 (1250)	551 (1250)
Average states reduced	0.66	4.64	143.18	9.06
Average states reduced if reduction is possible	1.79	5.89	143.18	18.2
Maximal percentage reduced	80%	50.8%	50.9%	80%
Average percentage reduced	12.7%	14.9%	33.8%	14.3%
Average percentage reduced if reduction is possible	34.3%	18.9%	33.8%	28.8%
Largest block	7 (8)	7 (86)	12 (1250)	12 (1250)
Average block size	1.26	1.22	1.57	1.27
Number reduced games	56	37	11	104
Percentage reduced games	37.1%	78.7%	100%	49.8%
Total games	151	47	11	209

Table 1. Data gathered from minimising the games

that there is actually a state that has the same behaviour is quite low since there are not many states to choose from.

There are also interesting results in the block size and maximal reduction. The largest block is 7 states (for a game with 8 states) which also was the maximal number of reduced states. The maximal percentage reduced is 80%. These are both incredibly high numbers. Therefore, it is possible to drastically reduce games to small sizes even if the games are small.

Also, the average percentage of reduced states when reduction is possible is quite high. If we combine this with the average states that were reduced if reduction was possible we can also see the reason why the percentage is so high. The average size of games where reduction was possible was 5.21 states. Already reducing one state of a game with 5 states means 20% is reduced. A percentage of 34.3% is therefore more easily achieved. Still, a reduction of 34.3% is desirable to have.

3.4 Medium games

There are a total of 47 medium games containing 11-100 states. 78.7% of those games are reduced which is very good. This means that a majority of the medium-sized games can be optimised by reducing the size. This makes up for an average reduction of 14.9% over all games. The reduction varies from 0-50.8%.

The reduction of states is not substantial but at least is some reduction available. The rest of the results for this set is not so interesting.

3.5 Large games

The set of large games contains 11 games which is not much. In order to make statements about this set, a bit more care has to be taken into account.

The first result that stands out is the percentage of reduced games which is 100%. We can conclude that large games can mostly be reduced in size.

Next, the average percentage of reduction is also very large namely 33.8%. Furthermore, the average number of reduced states is 143, which is also high. We can conclude that large games can be reduced highly in size and a lot of optimisation can be achieved if bisimulation minimisation is applied here.

The maximal percentage of reduced states is 50.9% and the maximal reduction of states is 551 (of 1250 states). This means that there are games that can be greatly reduced in size. Remember that there are only 11 games in this set which means that if more games were used there could be even more promising results. It could be that 60% or maybe even 70% can be reduced for some games.

3.6 Relations

Finally, the relations between the size of the games and the results obtained will be discussed.

The first relation is the number of states that can be reduced. It is logical to see that if the game increases in size there will be more states that can be reduced. Looking at the fig 5 and 6, a non-linear trend could be identified. With some imagination, a square root line can be identified as average reduction. However, this would require more data to confirm that the relation is non-linear.

If this relation is non-linear, it means that once games start to get large, the reduction starts to increase faster than linear which is promising. This means that how bigger the controller is, the higher the percentage of reduction will be when applying bisimulation minimisation first. This is an interesting direction for further research.

The maximal percentage that was reduced is also interesting. For small states, the highest percentage is 80% while the others sets do not go above 51%. This indicates that small games can be greatly reduced while larger games can only be largely reduced. It might be that if there are more large games this might change because that percentage might become higher with more data.

The percentage of reduced games also increases with game size. This is a reason why the average percentage that is reduced over all games increases with the game size. This indicates that the larger the game, the more (possible) redundancy it has. The relation changes when only the games where reduction is possible are considered. Then the small games have the highest average reduction and the medium games have the lowest reduction for which no logical relation can be found in the currently acquired data.

The block size remains constant over all the games which is also interesting. No matter how big the games get, they will not have a large block of bisimilar states. It will only have a lot of blocks with a few bisimilar states.

4 CONCLUSION

We have seen how to calculate bisimulation on parity games using a signature-based partition refinement algorithm. The results of the algorithm showed that it is possible to reduce the size of parity game strategies. Generally speaking, half of the games can be reduced in size. The probability that a game can be reduced in size, increases as the size of the game increases.

When reduction is possible, on average, more than a quarter of the game will be reduced in size which is a substantial amount of reduction. The reduction can go up to 80% reduction and maybe even further which allows some games to be greatly decreased in size.

From this, the conclusion can be made that bisimulation minimisation is a very good optimization technique to perform on parity game strategies. If optimisation is possible the parity game strategy size is greatly decreased.

There are multiple directions for future work that can be performed.

Firstly, as we have seen in subsection 3.6 there is a possible non-linearity in the relation between game size and the number of blocks. We have seen that if the number of states increased the average percentage of reduction also increased. Though this could not be confirmed due to the lack of large games. It is interesting to investigate if this relation holds if it is tested against more and bigger larger games.

Secondly, the algorithm discussed in this paper is sequential. There is room for parallelisation in the algorithm. Ideas for parallelisation are refining each block in parallel, computing the signatures of each state in parallel and computing the signature of the low edge and the high edge in parallel. Furthermore, adding the states to new blocks depending on their signature could also be done in parallel but this requires the use of concurrency features like locks since all the blocks are edited at the same time and it could be that the overhead induced by concurrency slows the process down instead of increasing its speed.

Lastly, this research focused on how many states can be reduced. How the controller is affected is not yet researched. It could be that the controller decreases with the same percentage as the BDD is decreased in size but it could also be that the controller does not improve or the quality even deteriorates. This depends on how the new BDD is formed. A naive approach might give not so much improvement as a smart approach to forming the new BDD.

REFERENCES

[1] R. Abraham. 2021. Symbolic LTL Reactive Synthesis. http://essay.utwente.nl/87386/
[2] S. Cranen, J. J. A. Keiren, and T. A. C. Willemse. 2018. Parity game reductions. *Acta Informatica* 55, 5 (2018), 401–444. https://doi.org/10.1007/s00236-017-0301-x 401.
[3] E. Grädel, W. Thomas, and T. Wilke. 2002. *Automata, logics, and infinite games.* Springer, Berlin ;. https://doi.org/10.1007/3-540-36387-4
[4] T. van Dijk. 2016. *Sylvan: Multi-core Decision Diagrams.* Ph.D. Dissertation. University of Twente, Enschede, Netherlands. https://doi.org/10.3990/1.9789036541602
[5] T. van Dijk. 2018. Oink: An Implementation and Evaluation of Modern Parity Game Solvers. In *Tools and Algorithms for the Construction and Analysis of Systems*, D. Beyer and M. Huisman (Eds.). Springer International Publishing, Cham, 291–308.
[6] T. van Dijk and J. van de Pol. 2016. Multi-core symbolic bisimulation minimisation. In *TACAS (LNCS, Vol. 9636)*. Springer, 332–348. https://doi.org/10.1007/978-3-662-49674-9_19