



Leiden University

Computer Science

Checkerboard Patterns and Binary Decision Diagrams
for the Game 2048

Name: Luc Edixhoven
Date: 18 July 2019
1st supervisor: Dr. W.A. Kusters
2nd supervisor: Dr. A.W. Laarman

MASTER'S THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

This two-part thesis is about the game 2048, in which the player slides and merges tiles to create a tile with a specific value, unless the board is full before they manage to do so.

In the first part, we show on which board sizes a specific checkerboard-like pattern can be reached, setting a lower bound on the length of games of 2048.

In the second part, we encode 2048 as a Boolean function and implement it using binary decision diagrams in an attempt to solve the typical 4×4 setup, falling short of completing this computation but nonetheless gaining some insights.

Contents

1	Introduction	1
2	On 2048	2
2.1	Game rules and modifications	2
2.2	Research goals	3
3	Properties of 2048	5
3.1	General properties	5
3.2	Properties of the bounded variant	6
4	Checkerboard patterns	8
4.1	Two rows or columns	8
4.2	Three rows or columns	10
4.3	General case	11
4.4	Filling boards	18
5	On solving 2048 with binary decision diagrams	20
5.1	Solving 2048	20
5.2	Binary decision diagrams	21
5.3	Retrograde analysis	23
6	Implementation and results	25
6.1	Encodings	25
6.2	Sweep line approach	27
6.3	Results	29
7	Conclusions and further research	32
7.1	Conclusions	32
7.2	Further research	32
	References	34

1 Introduction

Since its release in 2014, the game 2048 has received much attention, not only from enthusiastic players but also from the scientific community. Notably, in their 2018 paper [1] (although available on arXiv since 2015), Stefan Langerman and Yushi Uno prove the problem of deciding whether a given starting position can be played to reach a specific (constant) tile value, which is the typical objective of the game, to be NP-hard, using an ingenious reduction from 3-SAT. Even more recently, in their 2019 paper [2], Naoki Kondo and Kiminori Matsuzaki present new computer players using deep convolutional neural networks, achieving an impressive average score. These and other results on (specific) games often provide insights or methods that are more broadly applicable in the field of artificial intelligence.

For a more complete overview of work on 2048, which so far has mostly been focused on its computational complexity and the development of good artificial players, we recommend the extensive bibliographies of Mathé Zeegers' 2016 master thesis [3] and David Eppstein's 2018 paper [4].

In particular, we mention the results of Zeegers' work, in which he explored various theoretical properties of the game — notably reachability and short games — and the possibilities for solving its game-theoretical value. We will further study and expand upon theoretical properties of short games, focusing on the reachability of a specific, checkerboard-like pattern in a bounded variant of the game using only low-value tiles. Furthermore, we will attempt to compute the game-theoretical value of the typical 2048 game, hoping to use binary decision diagrams to bridge the gaps in computation time and memory use.



In Section 2, we describe the game 2048, its rules and our modifications, and present the goals of our research. In Section 3, we give an overview of properties of 2048 that are relevant to our questions. We analyse the possibilities for filling game boards of various sizes with a specific pattern and briefly speculate about an upper bound on shortest games in Section 4. We use Section 5 to discuss the solvability of 2048, to introduce binary decision diagrams and to describe our reason for using them and our approach. In Section 6, we give our encoding and implementation of 2048 with binary decision diagrams and analyse its results on three board sizes. Finally, we summarise our findings in Section 7, drawing our conclusions and giving some suggestions for further research we deem to be interesting.

This research was done as a master thesis at the Leiden Institute of Advanced Computer Science (LIACS), Leiden University, under the supervision of Walter Kosters and Alfons Laarman, with special thanks to Mathé Zeegers.

2 On 2048

The game 2048 was developed in 2014 by Gabriele Cirulli [5], based on 1024 by Veewo Studio, which itself is a clone of Threes by Asher Vollmer [6]. It is played on a 4×4 board, its cells either being empty or containing powers of two, which you can tilt in one of the four cardinal directions, causing all tiles to slide and align in that direction and merging same-valued tiles into the next power of two. A new tile is dropped between tilts on a random empty cell. The objective of the game is to create a tile with a value of 2048.

2.1 Game rules and modifications

New tiles typically have value 2 or value 4; all our new tiles will have value 2. Each tile may not merge more than once per move, the order of the merging depending on the direction of the move: sliding a row  to the right will result in . Tilting the board is only allowed if it causes the configuration to change in some way; the player loses if they can not make a valid move.

We adopt some of the modifications and conventions used by Zeegers [3, page 9]:

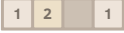
- We consider 2048 as a two-player game, calling the original player ‘Slider’ and introducing a new player ‘Dropper’, who drops the new tiles instead of this being done randomly. The game generally starts with an empty board and it is Dropper’s turn first, with the two players taking alternating turns from then on.
- We will typically consider a general $m \times n$ board instead of a fixed 4×4 one, with $m \geq 1$ the height and $n \geq 1$ the width of the board.
- For the sake of conciseness, we will use logarithmic values for tiles instead of powers of two, and 0 for value zero, i.e., empty tiles. Furthermore, since we will rarely use multiple-digit values in this notation, we will generally use shorthands such as 1201 instead of drawing . Later on, we will also resort to using regular expressions such as $1(21)^+0^*$ to represent patterns of arbitrary length.
- We say a configuration is *reachable* by either player if it is their turn and there exists some sequence of moves to it from the starting configuration. We say a configuration is *reachable in general* if it is reachable by at least one of the players.

Figure 1 shows an instance of 2048, both in normal and in logarithmic notation.

Furthermore, we also use the distinction made by Langerman and Uno [1, page 3] to describe *deterministic* 2048, which is identical to regular 2048 except that a new tile is always dropped in the topmost, then leftmost empty cell¹.

¹This is subtly different from the leftmost, then topmost convention used by Langerman and Uno. This is not for any particular reason, except that it maybe felt more natural. One

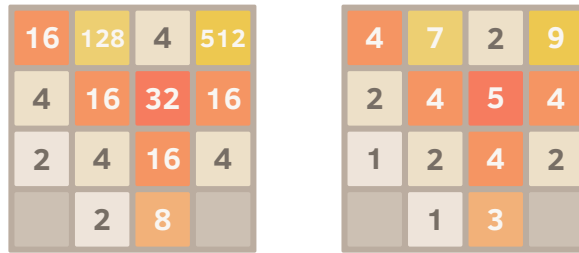


Figure 1: An example configuration of 2048, in normal and in logarithmic notation.

To denote sequences of moves, we will use a concatenation of U, D, L and R for Slider’s moves up, down, left and right. Dropper’s moves, if relevant, consist of coordinates (x, y) , where the upper left corner cell has coordinates $(1, 1)$.

2.2 Research goals

Our research consists of two separate questions, respectively concerning the reachability of a specific pattern on boards of arbitrary size and ‘solving’ the 4×4 game using binary decision diagrams.

In the first part, consisting of Section 4, we aim to answer the question whether a so-called ‘checkerboard’ pattern, consisting of alternating value 1 and 2 tiles, is reachable on an arbitrarily sized rectangular $m \times n$ board. Figure 2 shows an example of a checkerboard pattern on a 3×4 board, which we will later show to be unreachable. This can also be seen as an attempt to give a lower bound on the number of moves needed to finish a game of 2048. Since we are only interested in reachability, we will assume Slider and Dropper to cooperate in attempting to reach this configuration. Furthermore, since a tile can never decrease in value, we add a rule that it is not allowed to create a tile with a value of 3 or higher.

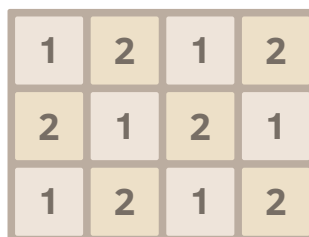


Figure 2: A checkerboard pattern on a 3×4 board.

In the second part, consisting of Sections 5 and 6, we translate 2048 to a series of Boolean formulas and use binary decision diagrams, a data structure designed specifically to handle these formulas efficiently, to try to compute the game’s

can obtain the other convention by simply mirroring the board (and any subsequent moves) in the main diagonal. They are thus symmetrical.

game-theoretical value, i.e., whether Slider can always perform moves in such a way that they can at some point create a value 2048 tile, or that Dropper can always drop new tiles in such a way that the game terminates before such a configuration is reached.

3 Properties of 2048

In this section, we summarise several properties of 2048, some from earlier research, for use in later sections.

3.1 General properties

Lemma 1. *Following a slide, there is at least one empty cell at the border of the board.*

Proof. This is a rewording of Theorem 4 by Zeegers [3, pages 13–14], which is proven by induction. \square

Corollary 2. *The last move in any game which fills the board is always a drop in a border cell.*

Lemma 3. *Following a slide in a given direction, there can be no empty cell with a non-empty cell in the direction opposite to that of the slide.*

Proof. In other words, all tiles are aligned against the border in the given direction, giving a typical skyscraper horizon such as in Figure 3. This follows from the game rules. \square

	1		1
	2	1	2
2	1	2	1

Figure 3: Example configuration of a 3×4 board after a downwards move.

Corollary 4. *Following a slide, there can be no adjacent pair of an empty cell and a non-empty cell in all four directions at the same time.*

Corollary 4 is also the first part of Theorem 5 by Zeegers [3, page 14], whose wording we borrow.

Lemma 5. *Following a slide, there can not be both a pair of horizontally and a pair of vertically adjacent value 1 tiles, possibly separated by empty cells.*

Proof. As a slide must be either horizontal or vertical, one of the pairs would have merged. \square

Lemma 6. *Using k cells, one can not obtain a tile with a value higher than k .*

Proof. We use the proof by induction for Theorem 6 by Zeegers [3, pages 14–15], substituting the mn cells of an $m \times n$ board for k cells in some arbitrary shape. \square

Corollary 7. *It must hold for any valid configuration that, for each value k tile present on the board, there must be at least $k - 1$ cells empty or with a value lower than k .*

So, for instance, it can not happen on a 4×4 board that there are 7 value 11 (or higher) tiles at the same time.

Lemma 8. *The values of the tiles in a row (respectively column) obtained solely by horizontal (respectively vertical) moves must be bitonic, i.e., first only increasing and from some point onwards only decreasing.*

Proof. This is a part of Theorem 10 by Zeegers [3, pages 19–21], which is proven by induction. \square

This means that, for instance, any sequence generating a configuration with a row such as 212 must contain at least one vertical move.

3.2 Properties of the bounded variant

We now describe some properties of 2048 when it is not allowed to create a tile with a value of 3 or higher.

Lemma 9. *Consider a game where it is not allowed to create a tile with value 3 or higher. Immediately after a downward slide, the board has the following properties:*

- a) *A 0 can not have a 1 or a 2 above it.*
- b) *There are no vertically adjacent 1's.*
- c) *If there are two vertically adjacent 2's, there must be a 0 in the first row of the same column.*

Furthermore, if there is a subsequent slide which is horizontal, the board also has the following properties:

- d) *There can be no horizontally adjacent 2's.*
- e) *There can be no horizontally adjacent 1's in any row but the bottom row.*

Proof. Property a) follows from the game rules as everything must be aligned in the direction of the slide, as does b) as two vertically adjacent value 1 tiles should have merged during a vertical slide; d) follows from the next slide being horizontal, as this would result in a value 3 tile; e) follows from a), b) and d).

Finally, observe that two vertically adjacent value 2 tiles are only possible after a vertical slide if at least one of them was just created from two value 1 tiles. There must thus be at least one empty cell in the same column and everything is aligned to the bottom, so at least the cell in the first row must be empty, which gives us c) — in fact, there must be at least one empty cell at the top for *any* disjoint pair of vertically adjacent 2's, but this is not needed for our proof. \square

Naturally, proofs for Lemma 9 for other directions are completely symmetrical, so the lemma applies to them as well.

Lemma 10. *Consider a game where it is not allowed to create a tile with value 3 or higher. If there are two full rows or columns with pattern $2(12)^+$ after a horizontal respectively vertical slide, the rows or columns between them can not contain any empty cells and either all of them follow a checkerboard pattern or at least one of them contains two adjacent value 2 tiles.*

Proof. These properties are trivial if the rows are adjacent, so we assume there to be at least one row between them. As a row with pattern $2(12)^+$ can not have been constructed solely horizontally by Lemma 8, these two must have been obtained either by a vertical move or by a drop. Let us first examine the former. We consider the configuration immediately after this last vertical move. Note that there can not have been any empty cells between the two rows because of Lemma 3. If these rows contain horizontally adjacent 2's, there can not have been any subsequent horizontal moves as they would have merged into a 3, so there still are horizontally adjacent 2's. If the rows contain horizontally adjacent 1's, because of Lemma 5 and there being no empty cells, there must also be horizontally adjacent 2's which brings us back to our earlier case. If the rows contain neither of these, they are complete and will not change with any subsequent horizontal moves, meaning they are still complete afterwards.

The alternative would be that one of the rows was completed by a drop. In this case, we have somehow obtained the following subconfiguration:

$$\begin{bmatrix} 2 & \cdots & 2 & 0 & 2 & \cdots & 2 \\ \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 2 & \cdots & 2 & 1 & 2 & \cdots & 2 \end{bmatrix}$$

By Zeegers' analysis [3, page 19], the first and last columns can not have been obtained solely by vertical slides so there must have been some horizontal slide involved. However, this is not possible by the empty cell in the top row of this subconfiguration. We can thus conclude that this situation can not be reached.

The proof for columns is completely symmetrical. \square

The arguments used in the proof of Lemma 10 can possibly also be applied more generally to restrict the difference in length between three rows in certain cases, without them necessarily being full. However, since this is not needed for our proofs we will not delve further into this.

4 Checkerboard patterns

In this section, we analyse whether an $m \times n$ board can be filled with a checkerboard pattern and, if so, whether it is possible to do so deterministically. Note that, if $m \neq n$, because of the spawning rules, solutions for deterministic cases can not always simply be mirrored. We will thus have to consider the cases $m \times n$ and $n \times m$ separately in some instances.

A $1 \times n$ board can be filled with a checkerboard pattern for $n \leq 3$, also deterministically. While it is possible to reach 121 with $n = 3$, it is not possible to avoid creating a value 3 tile for $n \geq 4$. Note that $m \times 1$ is completely symmetrical to $1 \times n$.

4.1 Two rows or columns

Both $m \times 2$ and $2 \times n$ boards can be filled with a checkerboard pattern deterministically.

For $m \times 2$, a strategy is to create alternate rows 12 and 21 and then move them down to stack them on top of each other. This is achieved by the sequence $(RRDRLD)^{\lfloor m/2 \rfloor} (RRD)^{(m-1) \bmod 2} RL$, where the last RRD takes care of the second topmost row if the number of rows is even, and the RL at the end fills the topmost row. We give an example for $m = 6$ in Figure 4.

To obtain the other checkerboard pattern, with a 1 in the upper left corner, simply swap RR's and RL's: $(RLDRRD)^{\lfloor m/2 \rfloor} (RLD)^{(m-1) \bmod 2} RR$.

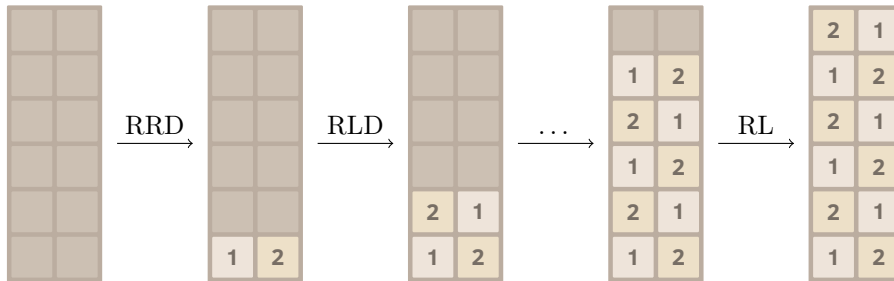


Figure 4: Filling a 6×2 board deterministically with a checkerboard pattern.

For $2 \times n$, a strategy is to first create a pattern of alternating 1's and 2's in the top row, and then completing it by correctly filling the bottom row. Some special care is needed for the leftmost columns depending on whether n is odd or even. This is achieved by the sequence $(DUR)^{\lfloor n/2 \rfloor - 1} DUD (UUD)^{\lfloor n/2 \rfloor - 1} UU$, which handles both the even and odd numbers of columns. We give examples for $n = 4$ and $n = 5$ in Figure 5.

To obtain the other checkerboard pattern for n even, with a 2 in the upper left corner, use the sequence $(DUR)^{\lfloor n/2 \rfloor - 1} RDU (DDU)^{\lfloor n/2 \rfloor - 1} DD$; swapping DUD for RDU lets you fill the upper row instead of the lower one, after which you need to invert up and down to correctly fill the second row. For n odd, use the

sequence $(\text{DUR})^{\lceil n/2 \rceil - 1}(\text{DDU})^{\lfloor n/2 \rfloor} \text{DD}$; after almost filling the upper row, use DDU instead of DUD to get a 1 in the upper left corner and invert up and down to correctly fill the rest of the second row. In the formula, the lone DDU substituting the DUD has been merged with the sequence of DDU's following it.

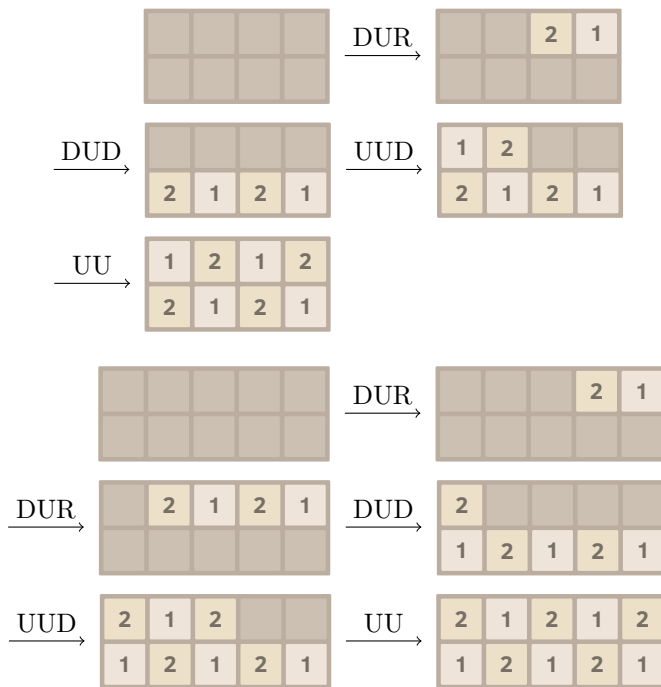


Figure 5: Filling 2×4 and 2×5 boards deterministically with a checkerboard pattern.

We have given a way to deterministically fill $m \times 2$ and $2 \times n$ boards with both variants of the checkerboard pattern. It follows that it is also possible nondeterministically by mimicking the deterministic spawns.

Methodology

The sequence for $m \times 2$ boards was intuitive enough to come up with by ourselves. For the sequence for $2 \times n$ boards, we wrote a program which generated all deterministic sequences to generate a checkerboard pattern for $3 \leq n \leq 13$. We quickly noticed that many of them only make use of three directions, after which we focused our attention on those containing only D, U and R, admittedly in part because of the easier pronunciation. We then searched these sequences, both manually and then procedurally with regular expressions, for one containing recurring patterns, which we eventually found.

Table 1 gives an overview of the number of sequences leading to a checkerboard pattern on a $2 \times n$ board. Interestingly enough, the sequence count for $n = 10$

is lower than that for $n = 9$ and the same is true for $n = 12$ and $n = 11$. We do not know why.

n	<i>Sequences</i>
3	197
4	761
5	1 860
6	3 320
7	7 385
8	8 470
9	18 351
10	16 988
11	42 655
12	35 869
13	98 859

Table 1: The number of deterministic sequences leading to a checkerboard pattern on a $2 \times n$ board.

4.2 Three rows or columns

As shown by Zeegers [3, page 47], although it is possible to fill $3 \times n$ boards with a checkerboard pattern for $n = 3$ and $n = 5$, it is not possible for $n = 4$. Our computations show that it is also possible for $n \in \{7, 9, 11\}$ and that it is not possible for $n \in \{6, 8\}$, which leads us to suspect that it is, in general, only possible for odd values of n — and of course for $n = 2$. Furthermore, although there are 750 ways to obtain a checkerboard pattern deterministically for 3×3 and 304 ways for 5×3 , it does not seem to be possible to do this for other board sizes (including 3×5) and these two cases seem to be the exception rather than the rule.

We prove that it is indeed not possible to create $3 \times n$ or $m \times 3$ checkerboard patterns for even values of m and n later on in Theorem 14, and give a way to generate these patterns for odd values of m and n in a nondeterministic game. Note that since we only seek to show existence, this is basically us saying we may drop new tiles wherever we want.

Since we already covered the case $n = 1$, we now assume $n \geq 3$ and odd. We then generate a checkerboard pattern on a $3 \times n$ board in three steps, as follows:

- First, we generate $\begin{bmatrix} 0 & 0 & 0 & \cdots \\ 1 & 1 & 0 & \cdots \\ 2 & 2 & 0 & \cdots \end{bmatrix}$ with Slider to move. This can be done, for instance, with the sequence of moves $(1, 1)D(1, 1)D(1, 1)D(1, 2)D(1, 2)D(2, 2)$.
- Next, for a configuration $\begin{bmatrix} \cdots & 2 & 1 & 0 & 0 & 0 & \cdots \\ \cdots & 1 & 2 & 1 & 1 & 0 & \cdots \\ \cdots & 2 & 1 & 2 & 2 & 0 & \cdots \end{bmatrix}$ with column k being the last checkerboard column (the second one depicted here, although

possibly 0 if one has just finished the first step), we perform the sequence of moves: $U(3, k + 2)D(1, k + 1)R(3, 1)L(1, n)R(1, 1)U(1, 2)L(1, n)D(2, n)L(2, k + 4)$. This generates exactly the same pattern, but with two additional checkerboard columns to the left and two fewer empty columns to the right. Repeat this until there is exactly one empty column to the right, which is the case after $(n - 3)/2$ times. Note that this means that this step is skipped for $n = 3$.

- Finally, once we have $\begin{bmatrix} \dots & 2 & 1 & 0 & 0 & 0 \\ \dots & 1 & 2 & 1 & 1 & 0 \\ \dots & 2 & 1 & 2 & 2 & 0 \end{bmatrix}$, we perform the following sequence of moves: $U(3, n - 1)D(1, n - 2)R(1, 1)L(1, n)R(2, 1)U(2, 1)D(1, 1)$, which completes our checkerboard pattern.

Methodology

Our approach for finding patterns by enumerating sequences does not work for nondeterministic sequences as there are simply too many of them. Recall from Table 1 that 2×3 had 197 deterministic sequences and 2×13 had 98 859 deterministic sequences. There are, with all symmetry unfiltered, 2 859 468 nondeterministic sequences to generate a checkerboard pattern on a 3×2 board. We thus opted for generating random sequences and searching these for patterns.

We noticed after generating 100 of these sequences on 3×7 that they all passed

the configuration $\begin{bmatrix} 2 & 1 & 2 & 1 & 0 & 0 & 0 \\ 1 & 2 & 1 & 2 & 1 & 1 & 0 \\ 2 & 1 & 2 & 1 & 2 & 2 & 0 \end{bmatrix}$ or one very similar to it. While generating random sequences to reach this configuration, we stumbled upon a sequence that passed the configuration

$\begin{bmatrix} 2 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 2 & 1 & 1 & 0 & 0 & 0 \\ 2 & 1 & 2 & 2 & 0 & 0 & 0 \end{bmatrix}$. The sequence of moves leading from this configuration to the former turned out to also work to

reach the latter from $\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 2 & 2 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$ and in general, granting us the missing link in our final sequence.

4.3 General case

Finally, we prove that it is not possible to reach a checkerboard pattern on all remaining board sizes, for which we first prove some helpful lemmas.

Figure 6 gives an overview of all patterns that can be transformed into a checkerboard pattern for rows with an even width, using only horizontal moves.

Lemma 11. *Any row (or column) that must become an even checkerboard row (or column) with length $n \geq 4$ using only horizontal moves, has the following properties:*

- The row (or column) must begin and/or end with a 2.*

b) The row (or column) contains at most two empty cells.

Proof. This can be observed in Figure 6. □

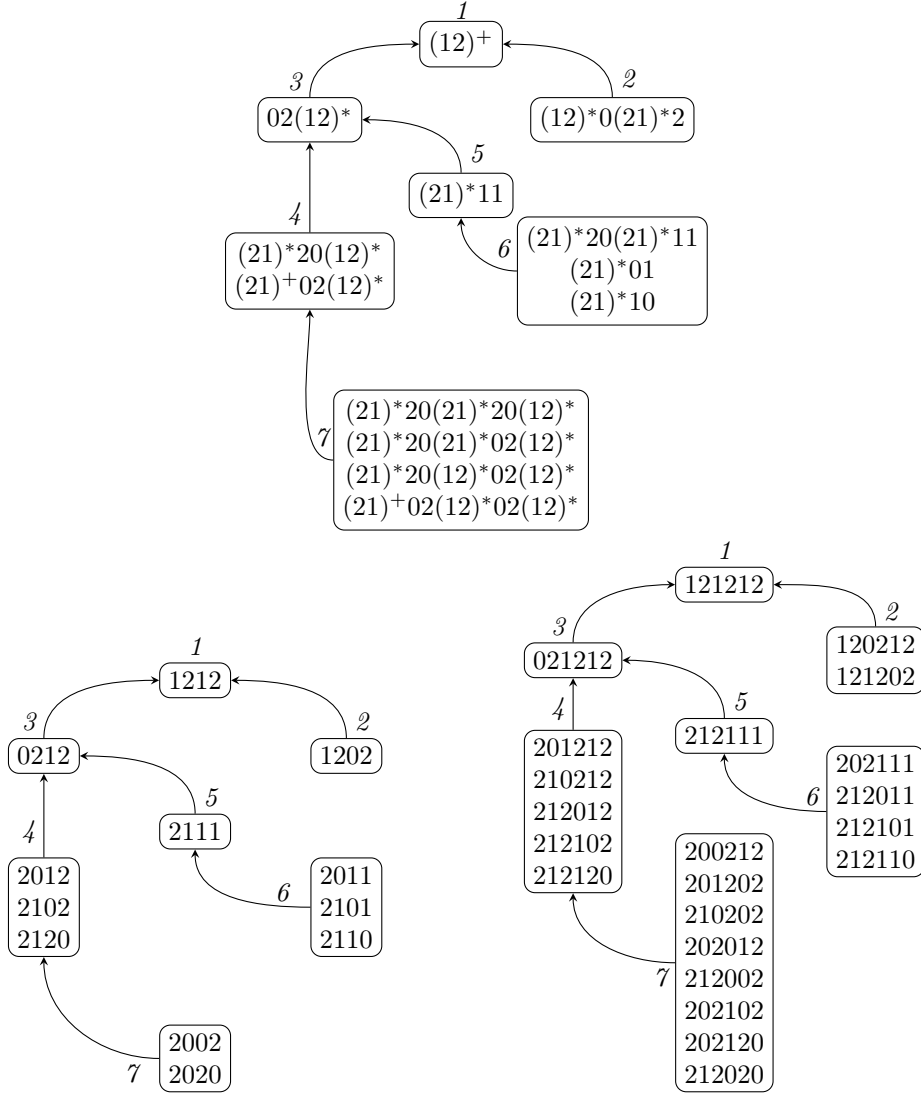


Figure 6: Diagrams showing the patterns from which a checkerboard row can be reached with an even board width in general (top) and specifically for board widths of 4 (bottom left) and 6 (bottom right). Note that the pattern from node 3 is mirrored at the bottom of node 4. As we could possibly slide a row multiple times while dropping tiles in other rows, the mirrored versions of nodes 4 and 7 can also be considered in the case of multiple rows.

Lemma 12. Any sequence of moves creating a checkerboard pattern on an $m \times n$ board with m and n both odd ends with a drop in a corner cell.

Proof. This is trivial if m or n equals 1, so let us assume that $m, n \geq 3$. Now suppose that there *does* exist some sequence of moves creating a checkerboard pattern on this board which does not end in a corner cell. We may assume that the last sliding move was a horizontal one. This must not be in the first or the last row since that would mean that the last move *is* in a corner. Our final row thus has two neighbouring rows, which must be identical. If the con-

figuration is $\begin{vmatrix} 1 & \cdots & 1 \\ x_1 & \cdots & x_n \\ 1 & \cdots & 1 \end{vmatrix}$, by Corollary 2 the final drop must be on x_1 or x_n ,

which is not possible since they must both be 2. We thus have the configuration

$\begin{vmatrix} 2 & \cdots & 2 \\ x_1 & \cdots & x_n \\ 2 & \cdots & 2 \end{vmatrix}$. Since $m, n \geq 3$, there must have been at least one vertical move

during the game. The $2 \dots 2$ rows are not affected by horizontal moves, so they must have been the same after the last vertical move; we may assume that it was a downward slide. It follows from Lemma 9bc that all even columns from the row $x_{1\dots n}$ contain a 2 and that all odd columns contain a 1. The row is thus already complete, contradicting our assumption that there exists some sequence not ending in a corner. \square

Corollary 13. *There are no reachable checkerboard patterns with 2's in the corners on an $m \times n$ board with m and n both odd.*

We are now well-prepared to prove our main theorem about checkerboard patterns.

Theorem 14. *It is not possible to fill an $m \times n$ board with $m, n \geq 3$ with a checkerboard pattern if either side is of even length or if both are odd and greater than or equal to 5.*

Proof. Given an $m \times n$ board with $m, n \geq 3$, we may assume that the last sliding move is a horizontal one, along the side with length n . If not, we can rotate the board. It is clear that, due to the board size, a checkerboard pattern can not be achieved using only horizontal moves Lemma 8, so there must have been at least one previous vertical move. We may assume that the last vertical move was a downward slide — otherwise we can just flip the board. We consider the configuration directly after this downward slide, before the subsequent dropping of a new tile, and with the knowledge that there must follow at least one horizontal move.

This gives us two cases, which we will handle separately. Let us first consider the case in which the width of the board is even. It follows from Lemma 9abc that any 2 in the top row must be in a complete checkerboard column. Using Lemma 11a we know that the top row begins and/or ends with a 2, so we may assume that the first (leftmost) column is a checkerboard column with a 2 in the first row. As there is a 1 in the second row of this column, Lemma 11a gives us that there must be a 2 in the second row of the last column. Schematically, we now have the following configuration:

$$\begin{bmatrix} 2 & \\ 1 & 2 \\ 2 & \\ \vdots & \end{bmatrix}$$

We now turn our attention to the third row of the last column. It can not be empty due to Lemma 9a, so let us assume that it contains a 2. It follows from Lemma 9c that the upper right corner must be empty and that the second and third rows of the second-to-last column may not contain 2's. As $\begin{smallmatrix} 1 \\ 0 \end{smallmatrix}$ and $\begin{smallmatrix} 1 \\ 1 \end{smallmatrix}$ are both not allowed, the second row must then contain a 0 and the first row as $\begin{smallmatrix} 0 & 0 \\ 0 & 0 \end{smallmatrix}$ well. This gives us $\begin{smallmatrix} 0 & 2 \\ ? & 2 \end{smallmatrix}$ for the top three rows of the last two columns. As can

be seen in the diagram in Figure 6 there is no pattern satisfying the first row, so the third row of the last column must be a 1. Due to the same argument and Lemma 9b, the rest of the column is then filled with a checkerboard pattern. We can thus update our configuration:

$$\begin{bmatrix} 2 & \\ 1 & 2 \\ 2 & 1 \\ \vdots & \vdots \end{bmatrix}$$

The upper right corner cell may not contain a 2 due to Lemma 9c. Suppose it contains a 0. It follows from Lemma 9de that the top row, apart from empty cells, consists of alternating 1's and 2's. If it does not contain another empty cell, the top row would be $(21)^+20$. A new tile is dropped in the corner and no further moves are possible, contradicting our statement that the last move was a horizontal one. There is thus at least one other empty cell in the first row — and, because of Lemma 11b, exactly one. The only pattern matching this scenario, as seen in the diagram, is $(21)^+20$ with an additional 1 changed into a 0. This 0 is flanked by two 2's, so the next tile must be dropped there to prevent the creation of a 3. If the second row is full, this forces the first row to create $(12)^+$, which is exactly the mirror of what is needed. There must thus be an empty cell in the second row, specifically below the aforementioned empty cell in the first row. However, this would give us $\begin{smallmatrix} 2 & 0 & 2 & ? \\ 1 & 0 & 1 & 2 \end{smallmatrix}$ in the first two rows where we still have to drop a 1 in the first row, leading to a 3 in the second row. We can thus conclude that the upper right corner cell contains a 1:

$$\begin{bmatrix} 2 & 1 \\ 1 & 2 \\ 2 & 1 \\ \vdots & \vdots \end{bmatrix}$$

We see in the diagram that there is no pattern $2 \dots 1$ with two empty cells, so the first row contains exactly one empty cell. It follows from Lemma 9abc that all

columns except for the one with the empty cell contain alternating checkerboard patterns. Because we know the value of the corners, we know that the values flanking the empty cell are the same — and likewise for any additional empty cells below it. Using the same argument as before, any configuration with 202 in the first row either gives the wrong pattern or creates a 3. The only remaining possibility is 101, for which the same argument can be made.

We therefore conclude that the width of the board can not be even: it must be odd.

Since we already assumed that the last move is horizontal, we can use an argument similar to that of Lemma 12 to show that the last move must be a drop in a corner; we may assume this to be in the first row, which then has 1's in the corners. It follows that every even row begins and ends with a 2 and is thus complete. Because of Lemma 9abd, all rows below them must also be complete, so the first row is the only one that is not yet complete. Furthermore, we know that the first row does not contain a 1 above a 1 in the second row or a 2 above a 2, and that it contains at least one empty cell.

After we drop a new tile, which must be in the first row, and perform a slide, which we may assume to be rightwards, we are in one of six configurations for the first row:

1. 0^+1
2. 0^+2
3. $0^+(12)^+1$
4. $0^+(12)^+$
5. $0^+(21)^+$
6. $0^+(21)^+2$

Configurations 1 and 2 are invalidated for $n > 3$ by Lemma 8. Configurations 3 and 4 both lead to the creation of a 3. Configuration 5 leads to configuration 3 unless it contains only a single 0, and configuration 6 leads to configuration 4 or 5, where it must contain exactly two 0's for configuration 5 to contain only a single one. It follows that the only configuration for the top row leading to a checkerboard pattern for $n \geq 5$, complying to the earlier restrictions and before the drop of a new tile, is $0(21)^*202(12)^*0$, which requires $n \geq 5$.

Let us first consider the case where the width $n \geq 5$ and the height m is odd. All odd columns are now of the pattern $2(12)^+$ which, combined with the fact that 202 must occur in the top row, is in contradiction with Lemma 10, so m can not be odd and must thus be even.

If m is even, we know that all columns are either $(12)^+$, $02(12)^+$ or $(21)^+$. Our configuration for $n = 5, m = 4$ is as follows:

$$\begin{bmatrix} 0 & 2 & 0 & 2 & 0 \\ 2 & 1 & 2 & 1 & 2 \\ 1 & 2 & 1 & 2 & 1 \\ 2 & 1 & 2 & 1 & 2 \end{bmatrix}$$

Since the last move was a downward slide, columns $(12)^+$ or $(21)^+$ will not have changed. The change must thus have come from one of the three columns $02(12)^+$, whose only predecessors are obtained by moving the empty cell along the column, or are $2(12)^*111$. Note that $2(12)^*111$ can not have occurred next to $(21)^+$ without contradicting Corollary 4 or Lemma 5, so thus far it can not have occurred at all. To comply with Corollary 4, the empty cells from the predecessors must all be in the same row to preserve a horizon and the only valid row for this is the bottom row. So our only valid predecessor is, for $n = 5, m = 4$:

$$\begin{bmatrix} 2 & 2 & 2 & 2 & 2 \\ 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 \\ 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

The only valid previous drop complying with Corollary 4 would have been in the bottom row. Undoing this and flipping the board gives us:

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 2 & 2 & 2 & 2 & 2 \\ 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 \end{bmatrix}$$

This resembles our original configuration. Now note that $2(12)^*111$ can not have occurred next to $(12)^+$ as there would be another column on the other side and this would either contradict Corollary 4 or Lemma 5, or Lemma 9c. The other option is a predecessor of $02(12)^+$, which gives us five options for the bottom three rows of those two columns:

$$\begin{array}{cc} 1 & 2 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 1 & 2 & 1 & 1 \\ 1 & 2 & 1 & 2 & 1 & 2 & 1 & 0 & 1 & 1 \end{array}$$

If we use the fact that there must be another column in at least one direction, it follows that none of these options are possible without contradicting at least one of Corollary 4, Lemma 5 and Lemma 9c. Once again, $2(12)^*111$ can not have occurred and, following the same argument as before, the only possible predecessor is, for $n = 5, m = 4$:

$$\begin{bmatrix} 2 & 2 & 2 & 1 & 2 \\ 1 & 1 & 1 & 2 & 1 \\ 2 & 2 & 2 & 1 & 2 \\ 0 & 0 & 0 & 2 & 0 \end{bmatrix}$$

Note that, in this particular configuration, there is no drop we can undo without contradicting Corollary 4. This fact is not changed when using a larger value of m as this does not change the bottom two rows. If we use a larger value of n and add more columns, there might have been an odd column $(21)^+$ where we could undo a drop by removing a 1 in the bottom row. However, this just brings us back in the same loop. This does not change the fact that $2(12)^*111$ can not have occurred in any previous configuration, so all we can do is keep

moving back and forth and removing 1's in the top or bottom row. We can do this $n - 3$ times before running out of possible moves to undo and contradicting Corollary 4. We can thus conclude that it is not possible for n to be odd and greater than or equal to 5.

This leaves us with only $n = 3$, for which the only two (or three) configurations for the top row that can lead to 121 are 100 (or 001) and 000. Since we have previously given a way to create a checkerboard pattern if m is odd, we will now assume m to be even. As before, we know everything to be complete except for the top row. The overlap between the two options for the top row give us the following situation:

$$\begin{bmatrix} 0/1 & 0 & 0 \\ 2 & 1 & 2 \\ 1 & 2 & 1 \\ \vdots & \vdots & \vdots \\ 2 & 1 & 2 \end{bmatrix}$$

The main difference with the earlier case $n \geq 5$ is the second column, which does not have a 2 at the top. Note that, due to its structure, no tiles can have merged in the last vertical move in $01(21)^+$ — it should then either have ended with a 2, or have had adjacent 2's. Once again, it is not possible to have had $2(12)^*111$ in the first or third column in the previous move, as all four options

$$\begin{array}{cccccc} 1 & 1 & 0 & 1 & 2 & 1 & 2 & 1 \\ 2 & 1 & 2 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \end{array}$$

for the bottom three rows of the last two columns, combined with options for the remaining column, contradict at least one of Corollary 4, Lemma 5 or Lemma 9c. Our only option for a predecessor is thus to shuffle the empty cells around and, by a combination of earlier arguments for $n \geq 5$ and Lemma 10, the only possibility is, after undoing the preceding drop as well and flipping the board if needed, assuming that the upper left corner contained a 0:

$$\begin{bmatrix} 0 & 0 & 0 \\ 2 & 0 & 2 \\ 1 & 2 & 1 \\ \vdots & \vdots & \vdots \\ 2 & 1 & 2 \end{bmatrix}$$

Note that, for the top two rows, $\begin{bmatrix} 0 & 0 & 0 \\ 2 & 1 & 2 \end{bmatrix}$ and $\begin{bmatrix} 1 & 0 & 0 \\ 2 & 0 & 2 \end{bmatrix}$ are also possible if the upper left corner contained a 1, but the first one is exactly our other starting configuration and the second one is not significantly different as it does not invalidate the following argument.

As the first and third columns now contain less zeroes than the second one and can be seen as 'taller', it follows from Lemma 3 that, unless we can reduce the number of nonzeros in the first or third column, or increase the number of nonzeros in the second column, there can be no previous horizontal moves — of which there must be at least one, from Lemma 8 and the first or third

column. Decreasing the number of nonzeros in the first or third column can not be done with only vertical moves, as splitting a 2 into two 1's gives $2(12)^*111$ which can not be the result of a vertical move, even after undoing a drop. We must therefore increase the number of nonzeros in the second column by splitting a 2 there. This can not be done immediately due to alignment, so we must first take another step back to obtain:

$$\begin{bmatrix} 0 & 0 & 0 \\ 2 & 0 & 2 \\ 1 & 0 & 1 \\ 2 & 2 & 2 \\ \vdots & \vdots & \vdots \\ 2 & 2 & 2 \end{bmatrix}$$

If we were to split the bottom 2 in the second column, we would obtain $00(21)^*11$ or $(21)^*1100$ as our options not contradicting Corollary 4, the former one only for $n = 4$ in which case it is symmetrical to the latter, so we will only consider the second case. The options for the bottom four rows of the first and third columns are:

$$\begin{array}{cccccc} 2 & 1 & 0 & 2 & 2 & 2 \\ 1 & 2 & 2 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 2 \\ 1 & 2 & 2 & 2 & 2 & 0 \end{array}$$

Putting these next to 1100 either gives a direct contradiction with Corollary 4 or requires undoing a drop in the second column, which decreases its number of nonzeros instead of increasing it. As it is not possible for previous horizontal moves to exist and they must exist to obtain a checkerboard pattern, we conclude that it is not possible for n to equal 3 if m is even.

Summing everything up, we can conclude that the width of the board can also not be odd if $n \geq 5$ or if m is even, with the last move being horizontal. Since the width could also not be even, taking symmetry into account, it is not possible at all to reach a checkerboard pattern on an $m \times n$ board with $m, n \geq 3$ if the length of either side of the board is even, or if both are greater than or equal to 5. \square

4.4 Filling boards

Although this is not part of our stated research goals, it is interesting to note that, while it is not possible to fill every board with a checkerboard pattern, it is generally possible to fill every board with only value 1 and 2 tiles at some point — naturally with the exception of $1 \times n$ boards. This might give some insights and possibly new upper bounds on the lengths of shortest games. On a consumer laptop, our program consistently takes about 10 seconds to find a sequence of moves leading (nondeterministically) to the configuration in Figure 7. From this configuration, we can reach the endgame configuration in Figure 8 in a total of 157 drops, which is significantly less than the 228 given by Zeegers for this board size. In fact, our brief experiments show that similar configurations can be reached on 4×4 , 6×6 and 8×8 boards, leading us to suggest a suspected

upper bound of $\frac{3}{2}m \cdot n + 7$ on any board with $m, n > 2$ and even, instead of the upper bound of $\frac{3}{2}m(n-1) + 3 \cdot 2^{\frac{m}{2}} - 3$ given by Zeegers. The same experiments suggest that a very similar configuration can be obtained on boards where only one of m, n is odd. However, we are not able to give any concrete strategy to actually reach any of these configurations.

1	1	2	1	2	1	2	1	2	1
1	2	1	2	1	2	1	2	1	2
2	1	2	1	2	1	2	1	2	1
1	2	1	2	1	2	1	2	1	2
2	1	2	1	2	1	2	1	2	1
1	2	1	2	1	2	1	2	1	2
2	1	2	1	2	1	2	1	2	1
1	2	1	2	1	2	1	2	1	2
2	1	2	1	2	1	2	1	2	1
1	2	1	2	1	2	1	2	1	2

Figure 7: A reachable configuration on a 10×10 board, filling the board with only value 1 and 2 tiles. The only anomaly with regard to a checkerboard pattern is the upper left corner cell.

1	2	3	1	2	1	2	1	2	1
2	3	1	2	1	2	1	2	1	2
3	1	2	1	2	1	2	1	2	1
1	2	1	2	1	2	1	2	1	2
2	1	2	1	2	1	2	1	2	1
1	2	1	2	1	2	1	2	1	2
2	1	2	1	2	1	2	1	2	1
1	2	1	2	1	2	1	2	1	2
2	1	2	1	2	1	2	1	2	1
1	2	1	2	1	2	1	2	1	2

Figure 8: A reachable endgame configuration on a 10×10 board in 157 moves by Dropper.

5 On solving 2048 with binary decision diagrams

In this section, we describe our interpretation of ‘solving’ 2048, why we believe that binary decision diagrams might help with this and how we apply them.

5.1 Solving 2048

As described by Aaron Siegel [7], in terms of combinatorial game theory, *solving* a game is to give a polynomial-time algorithm to compute its outcome. As we view 2048 as a two-player game, which has no hidden information or chance elements, has a finite number of distinct subpositions and does not allow for infinite runs, with a known starting player, it follows from the theory that exactly one of the two players must have a winning strategy, regardless of their opponent’s moves. As any bounded game is by this definition a priori solved because a trivial brute-force approach is only bounded by some (potentially impossibly large) constant and is therefore polynomial, the definition needs a little more nuance. We thus follow the terminology proposed by Allis [8] to distinguish three levels of game solving; we consider a game to be solved *ultra-weakly* if its game-theoretical value is known, i.e., from the starting position, *weakly* if there is a known strategy for obtaining the game-theoretical value regardless of the opponent, and *strongly* if such a strategy is known for every legal position. While solving 2048 by the first definition is trivial in principle, solving it ultra-weakly is already a much more challenging task even for the typical 4×4 board. This is, amongst other things, suggested by the fact that the generalised version of 2048 is NP-hard [1]. To our knowledge, the game-theoretical value of the typical 2048 setup has not yet been successfully computed. However, very good results are achieved by AI players using heuristics, such as described on Stack Overflow [9].

To illustrate this, consider the 4×4 board. It contains 16 cells with (logarithmic) tile values ranging from 0 to 16 — or up to 17 if we were to allow value 2 tiles to be dropped directly instead of only value 1 tiles. This gives us an upper bound on the number of states of $17^{16} \approx 5 \cdot 10^{19}$, $12^{16} \approx 2 \cdot 10^{17}$ of which are interesting if we only wish to know whether it is always possible to reach a value 11 tile. A significant number of these states are illegal as, for instance, it is not possible to have sixteen value 11 tiles at the same time by Corollary 7. However, the vast majority of those 12^{16} states can not be discounted easily and the resulting state space is gigantic, as is the hidden constant in the trivial polynomial algorithm.

Furthermore, these are only the unique configurations — which most certainly is not a bijection of the paths in the game tree. Even at the very top of the game tree, there are 16 possible cells to drop the first tile and 48 nodes at the second level (after discarding the slides that do not change the configuration), but only 12 distinct configurations — or even as few as 2 after considering symmetry. Because of this duplicity, which occurs all over the game tree, the number of paths from the root to leaves is far more than the number of distinct states in the game. Iterating over all these paths would take an astronomical amount of time, so programs tackling this problem instead save the outcome of (some of) the states to speed up the computation.

Zeegers [3] presents an algorithm using a hash table to save configurations and describes several optimisations to reduce the table’s size, such as packing multiple configurations in a single byte, eliminating symmetry and the use of constraints to exclude certain impossible configurations from the hash table. Zeegers also suggests possible further improvements to reduce the program’s memory footprint. However, in the end this approach still requires a table containing a huge number of individual configurations — which is where binary decision diagrams come in. Binary decision diagrams, or BDDs for short, are popular data structures for representing and manipulating Boolean functions. Although they had been around for slightly longer, they were formalised by Sheldon Akers [10] in 1978. Randal Bryant showed their practical importance in his 1986 paper [11] due to their ability to allow for various function manipulations in polynomial time. Bryant showed that, when ordered and reduced, BDDs give a unique representation of Boolean functions, and presented efficient algorithms for various operations on Boolean functions. While BDDs thus also sacrifice memory for computation time, they do not save single states but instead use a compressed, symbolic representation of sets of states, thus greatly expanding the range of possible applications. In particular, they might be efficient enough to overcome our 2048 memory hurdle. Stefan Edelkamp, Peter Kissmann and Martha Rohte show in their 2014 paper [12] that a BDD-based approach to solving games is feasible by applying a hybrid approach to Connect Four.

5.2 Binary decision diagrams

A *binary decision diagram* (BDD) is in essence a compressed binary decision tree representing a Boolean function, where the variables are evaluated in order. An example binary decision tree is showed in Figure 9 with its corresponding BDD.

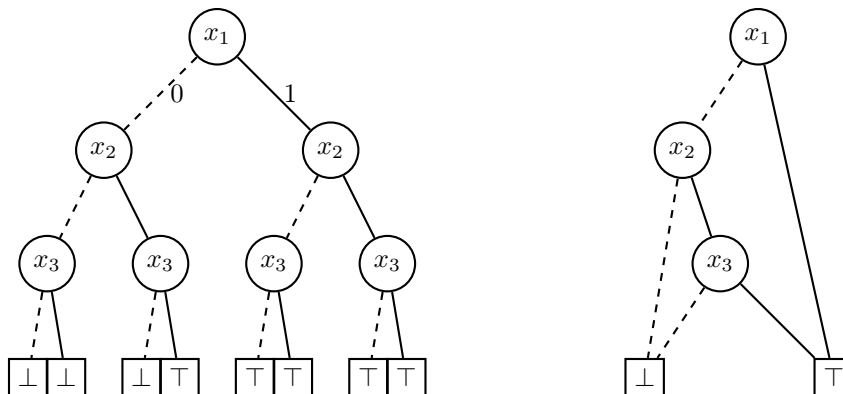


Figure 9: The binary decision tree (left) for the Boolean function $x_1 \vee (x_2 \wedge x_3)$ and its corresponding BDD (right).

Formally, a BDD is a directed acyclic graph. Each node is labelled with a variable and has both a low and a high branch, pointing to the next node in the decision tree when the current node evaluates to false respectively true, or to one of the

sinks \top (true) or \perp (false). Typically, the low branch is depicted as a dashed line and the high branch as a solid one. In practice, a BDD can be seen as a collection of triples (variable, low, high), where *variable* is typically an integer corresponding to its position in the ordering and *low* and *high* are pointers to other triples.

We will assume all our BDDs to be both ordered and reduced. In an *ordered* BDD, it holds for each branch from a node i to a node j that $i < j$, where i and j are the indices of the variables of those nodes. In a *reduced* BDD, there are no nodes whose high and low branches point to the same node or sink, and there are no two nodes with the same combination of variable and high and low branches. The former are skipped, the latter are merged. Figure 10 illustrates why the exact order of variables is important: depending on the ordering, the number of nodes in a BDD can grow possibly exponentially in the number of variables. Finding optimal orderings is however notoriously difficult. Furthermore, there exist inherently complex functions for which each ordering leads to an exponentially sized BDD, as shown by Knuth [13].

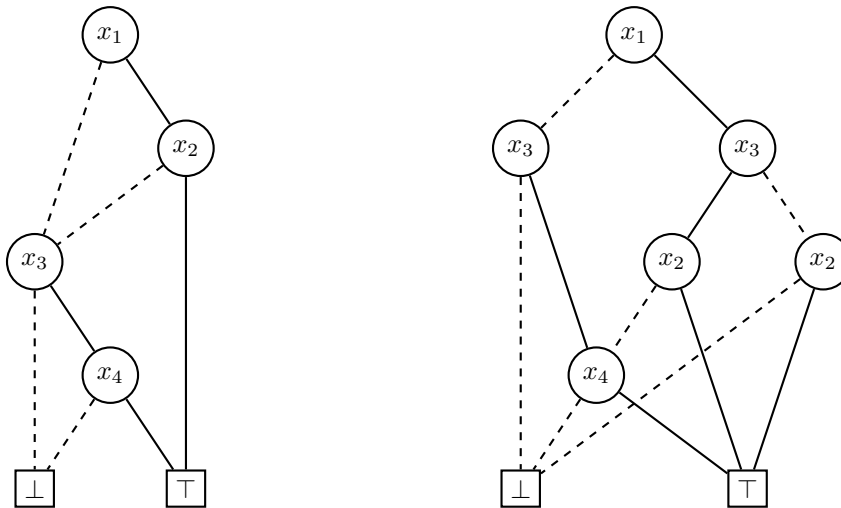


Figure 10: The BDD representing the Boolean function $(x_1 \wedge x_2) \vee (x_3 \wedge x_4)$, using two different orderings of the variables: 1, 2, 3, 4 and 1, 3, 2, 4.

Once a Boolean function is compressed into a BDD, there exist efficient algorithms to perform various operations on the function, such as, naturally, evaluating it for a given valuation, but also finding the lexicographically smallest satisfying assignment, counting solutions and efficient manipulations for applying conjunctions, disjunctions, quantification, factoring and variable renaming. The interesting part here is that all these operations, including the synthesis of new BDDs, can be performed directly on the BDDs — the compressed form — without the need to uncompress.

For more reading on BDDs, we recommend Donald Knuth’s 2011 book on combinatorial algorithms [13].

5.3 Retrograde analysis

Our aim is to apply retrograde analysis to solve 2048 using BDDs, starting with the set of winning states for Slider and iteratively adding winning predecessors for Slider by alternately adding the states from which Slider can move to a winning state and the states from which Dropper is forced to move to a winning state. If at any point we add the initial state, the initial state is winning for Slider. Conversely, if the set of winning states converges without adding the initial state, the initial state is winning for Dropper.

Formally, let us take B to be the set of all board configurations, $T_1, T_2 \subseteq B \times B$ the move relations for player P_1 respectively P_2 , $W_1, W_2 \subseteq B$ the subsets of boards won by P_1 respectively P_2 , and $I \subseteq B$ the set of initial configurations. Retrograde analysis now comes down to fixpoint computation. Roughly speaking, in our current context a *prefixpoint* is a set of configurations from where the starting player can always move and force moves to other configurations within the prefixpoint.

Let $p = 1$ (player) and $o = 2$ (other), or vice-versa. We find a prefixpoint $F \subseteq B$ such that the following holds:

- $W_p \subseteq F$ — all winning boards for P_p are contained within the fixpoint;
- $(\exists y \in B : xT_p y \wedge y \in F) \rightarrow x \in F$ — any configuration from which P_p can reach a configuration in the fixpoint, must also be in the fixpoint;
- $(\exists y \in B : xT_o y \wedge y \notin F) \rightarrow x \notin F$ — any configuration from which P_o can reach a configuration that is not in the fixpoint, must also not be in the fixpoint.

Let F_1 be the *least* such prefixpoint F with $p = 1, o = 2$ and F_2 the least prefixpoint with $p = 2, o = 1$. The sets F_1 , F_2 and $B - F_1 - F_2$ are now three disjoint sets solving the game by dividing all configurations into those winning for P_1 , those winning for P_2 and those leading to a tie. Note that a tie is not possible for 2048. A winning strategy, thus strongly solving the game, is then simply to perform arbitrary moves whose result stay within the current set, as at least one of these must exist. By construction, the other player is then forced to do so as well. As the game is finite, an endgame position is reached at some point in time within the same set. If it is known which of the above sets contains I , the distinction between the other two can be omitted, in which case the game is solved weakly. Finally, to only solve the game ultra-weakly, it actually suffices to prove the existence or nonexistence of any prefixpoint F explicitly *not* containing I — note that, by the Knaster-Tarski theorem [14], this prefixpoint F does not have to be minimal.

We can compute these prefixpoints by using preimaging: starting with the set $F = W_p$, we can use a preimage operator ($\text{preimage}(F, R) = \{x : \exists y \in F \text{ such that } xRy\}$) to add all configurations from which P_p (using R) has a move to a configuration in the fixpoint, and the \forall preimage operator described by Richard Huybers and Alfons Laarman [15] ($\forall\text{preimage}(F, R) = \{x : xRy \rightarrow y \in F\}$) to add all configurations from which all of P_o 's moves lead to configurations in the fixpoint. Note that any configuration without valid successors

will automatically fulfill the condition for \forall preimage, resulting in so-called deadlocks. These deadlocks can be simply found by computing \forall preimage(\emptyset, R) (\emptyset representing the empty set) and can be excluded by intersecting with its negation. This approach for deciding whether a game is winning for a player P_p can be summarised in Algorithm 1. The algorithm solves a game (ultra-)weakly, assuming no ties are possible. If ties are possible, one can still solve the game by running it twice — once for each player; the game results in a tie if both runs return false. Note that Algorithm 1 stops as soon as it finds the initial state and thus does not construct the entire prefixpoint F_p , but it can be adapted to do so by moving the check on line 8 outside the while-loop on line 4, in which case it solves the game strongly. It will then only stop building F_p once it converges.

Algorithm 1 Solving a game (ultra-)weakly using retrograde analysis, assuming no ties are possible. Returns true if the game is winning for P_p , false otherwise.

```

1: deadlocks  $\leftarrow$  forall_preimage( $\perp, T_o$ )
2:  $F \leftarrow W_p$ 
3: old  $\leftarrow \perp$ 
4: while  $F \neq old$  do ▷ Stop once converged
5:   old  $\leftarrow F$ 
6:    $F \leftarrow F \cup \text{preimage}(F, T_p)$ 
7:    $F \leftarrow F \cup (\text{forall\_preimage}(F, T_o) \cap \neg \text{deadlocks})$ 
8:   if  $F \cap I \neq \emptyset$  then ▷ Positive if initial state is found
9:     return true
10:  end if
11: end while
12: return false ▷ Negative if converged

```

This algorithm can be implemented with BDDs by supplying BDDs for W_1, W_2, T_1, T_2 and I , which requires some Boolean encoding of configurations using a vector of variables x . The BDDs for the sets W_1, W_2 and I are then defined over x , while those for the relations T_1 and T_2 are defined over both x and its primed copies x' . We describe our encodings and implementation in detail in Section 6.

6 Implementation and results

We have implemented 2048 with binary decision diagrams by means of the open source parallel BDD library Sylvan [16]. In this section, we will first describe how we encode the game and explain other implementation details, and then analyse our results using this implementation.

Unless otherwise specified, experiments were run on a machine using 16 2.40GHz threads and an initial table size of 1536MB, using a maximum of 196GB of memory.

6.1 Encodings

Recall the functions described in Section 5.3. Since 2048 does not allow for ties, we use Algorithm 1, we take $P_1 = \text{Slider}$ and $P_2 = \text{Dropper}$, and only use W_1 , T_1 , T_2 and I , whose encodings we will describe shortly. Admittedly, using W_2 instead of W_1 would also work and doing both would allow us to determine the game-theoretical value of every legal configuration. We do not have reason to believe that using W_2 would lead to significantly better results than W_1 and determining the game-theoretical value of the initial configuration shows to be troublesome enough already, so we have not implemented this. However, since the encoding of W_2 is not complex we will still provide it, and it might be worth implementing and experimenting with.

Variables and ordering

Our encodings are defined over a vector of Boolean variables (1/0) representing a configuration and, for the relations, an additional copy of these:

- $\lceil \log M \rceil$ variables per tile with M the value of the objective tile, collectively denoted as $V_{i,j}$ for the tile in row i and column j , where the values of the $\lceil \log M \rceil$ bits represent the binary value of the tile. Recall that the upper left corner is $(1, 1)$.
- A single variable P to keep track of the turn order, where a value of 1 denotes it is Slider's turn to move and 0 denotes Dropper's.
- Primed copies $V'_{i,j}, P'$ for use in describing relations.

For a 4×4 board with 16 cells and $M = 11$ ($\lceil \log M \rceil = 4$), this gives us a total of 65 unprimed and 65 primed variables. We use a rowwise ordering for the $V_{i,j}$, starting at the uppermost row, preceded by P . As is usual, the primed variables are interwoven with the unprimed ones: $P, P', V_{1,1}, V'_{1,1}, V_{1,2}, V'_{1,2}, \dots, V_{m,n-1}, V'_{m,n-1}, V_{m,n}, V'_{m,n}$.

Our first implementation used a one-hot encoding, where each cell is represented by 12 bits, exactly one of which is set to 1, and its value is $n + 1$ if the n th bit is 1 and 0 in the case of the first bit. In our early experiments the binary encoding has shown to be more efficient by a factor of approximately 3, which is why we

choose to use it instead. We have not run more thorough experiments comparing the two encodings and we do not expect the one-hot encoding to outperform the binary encoding in our current implementation (either significantly or at all), but we feel it might be worth trying the one-hot encoding in a zero-suppressed decision diagram [17] instead of a regular BDD.

We have also not experimented extensively with other variable orderings. We believe our rowwise ordering to be efficient for the transition relation since it provides horizontal locality within rows, but it does not provide vertical locality within columns. We provide a workaround for locality in vertical moves, but not for representing configurations, which requires both horizontal and vertical locality. It might thus very well be that another encoding would result in a smaller representation of result sets and this might be worth looking into.

Configuration sets

We give the following encoding for the sets of configurations W_1 , W_2 and I :

$$I = \{b \in B : \underbrace{\overline{P}}_{\text{Dropper starts}} \wedge \underbrace{\forall i, j : V_{i,j} = 0}_{\text{All cells empty}}\} \quad (1)$$

$$W_1 = \{b \in B : \underbrace{\overline{P}}_{\text{Slider finishes}} \wedge \underbrace{\exists i, j : V_{i,j} = 11}_{\text{Value 11 (2048) tile}}\} \quad (2)$$

$$W_2 = \{b \in B : \underbrace{P}_{\text{Slider's turn}}\} \cap \underbrace{\forall \text{preimage}(\perp, T_1)}_{\text{Slider has no follow-up move}} \quad (3)$$

Move relations

We give the following encoding for Dropper's move relation:

$$T_2 = \underbrace{\overline{P} \wedge P'}_{\text{Turn order}} \wedge \exists i, j : \left(\underbrace{V_{i,j} = 0 \wedge V'_{i,j} = 1}_{\text{1 dropped in empty cell}} \wedge \underbrace{\forall k, \ell, (k, \ell) \neq (i, j) : V_{k,\ell} = V'_{k,\ell}}_{\text{Other cells unchanged}} \right) \quad (4)$$

Naturally, the second part can be easily extended to $(V'_{i,j} = 1 \vee V'_{i,j} = 2)$ to also allow for value 2 tiles to drop, thus simulating the original game.

Slider's move relation is more complicated as it involves many potential changes to the board. We encode it as the relational composition of an alignment step, a merge step and a second alignment step, disjuncted for the four directions. Due to the ordering of the variables, the resulting BDDs for the vertical moves are much larger than the horizontal ones. For that reason, we instead implement vertical moves by mirroring configurations in the main diagonal, performing the corresponding horizontal move and mirroring the resulting configuration back. We thus only directly encode horizontal moves. Furthermore, since the encoding

of a move to the right is completely analogous to one to the left, we will only describe the latter. The full horizontal move relation T_h is given by

$$T_h = \underbrace{P \wedge \overline{P'}}_{\text{Turn order}} \wedge \exists d \in \{\text{Left, Right}\} : \underbrace{A(M(A(d)))}_{\text{Move}} \wedge \underbrace{\exists i, j : V_{i,j} \neq V'_{i,j}}_{\text{Force change}} \quad (5)$$

where A is the alignment function and M the merge function.

We implement the alignment function by disjuncting² the 2^n possible combinations of zeroes and nonzeros in a row and their corresponding left-aligned results and conjuncting these for m rows. We implement the merge function by hard coding the options for rows of size 2, 3 and 4. Because of our mirroring approach for performing vertical moves, our implementation only works for square boards, i.e., when $m = n$.

Additional constraints

To speed up the computation, we further constrain the move relations T_1 (T_h) and T_2 by conjuncting their outcomes with two additional BDDs:

1. Any configuration prior to a slide must have at least $i - 1$ cells with a value smaller than i for each value i on the board, following Corollary 7.
2. Any configuration prior to a slide or drop must not have a value m tile, where m is the current objective tile — generally 11. This ensures that the search process stops as soon as this is found.

Constraint 2 naturally helps quite a lot by ensuring the program’s termination on configurations deep in the game tree. Constraint 1, interestingly enough, leads to slightly smaller BDDs and speeds up the program for 3×3 boards, but inflates the BDDs and slows down the search for 4×4 while in the end leading to the same result. All other constraints we have tried, such as Lemma 1, are directly tied to the existence of a previous or follow-up move, which the program would have discovered within one or two steps even without our help, so they do not seem to make any significant difference one way or another. For this reason, we have not added more constraints, such as Corollary 4, as we do not expect these to make any significant difference.

6.2 Sweep line approach

In order to keep the BDDs at a smaller size, we use a sweep line approach [18] which carves up the search space in lines (k, ℓ) , where sweep line (k, ℓ) is confined to the boards containing exactly ℓ value k tiles and no tiles with a value larger than k . Using this approach, we keep performing moves within the current sweep line until it converges when no more moves are possible within the

²Recall that a BDD uniquely represents a Boolean formula, so while it also represents its set of outcomes, we feel comfortable using the terms ‘disjunction’ and ‘conjunction’ instead of ‘union’ and ‘intersection’.

same line. We then compute all predecessors and propagate these to the other relevant sweep lines, and discard the current sweep line as it will not be visited again. After all, tiles can never split up, only merge into higher-valued ones. Furthermore, configurations in sweep line (k, ℓ) only have predecessors in sweep lines $(k, 1 \dots \ell - 1)$ and $(k - 1, 2\ell \dots n^2)$ ³, with n the height and width of the board. The search is then more localised.

However, a significant disadvantage is that the sweep lines with a high value of k and especially with a low value of ℓ are disproportionately large. Including all illegal configurations by Corollary 7, the total number of possible configurations for (k, ℓ) is $\binom{n^2}{\ell} \cdot k^{n^2 - \ell}$. The total number of legal configurations is $\binom{n^2}{\ell}$ multiplied by the number of legal configurations with only tiles valued lower than k and at least ℓ empty cells, which is a summation of parts of all earlier sweep lines, except those with an identical value for k . As it turns out, at least using our encodings, the BDDs corresponding to these sweep lines also grow exponentially. The bulk of the computations will therefore still be performed in those sweep lines instead of being distributed evenly. While it is possible to use a yet more fine-grained approach and use, for instance, sweep lines (k, ℓ, p) containing exactly ℓ value k tiles and p value $k - 1$ tiles, and so on, our attempt to do so did not result in any results indicating a speedup.

Table 2 and Table 3 show comparisons in run time and memory usage between our implementations of the sweep line approach and the straightforward, accumulative approach without sweep lines. We see that the sweep line approach is faster and has a smaller memory footprint both in the size of the largest BDD, which correlates with the largest memory usage during the entire execution, and in the final size of the hash table. Note that for both approaches, after some point the BDDs do start converging and decreasing in size.

Approach	Run time (s)	Largest #nodes	Final table size (MB)
Accumulative	295	11 445 847	49 152
Sweep	176	5 043 195	24 576

Table 2: A comparison of the run time and memory usage of two approaches, on a 3×3 board looking for a value 9 tile, using 4 bits to represent values.

Approach	Run time (s)	Largest #nodes	Final table size (MB)
Accumulative	17	10 346 719	6 144
Sweep	8	3 555 795	3 072

Table 3: A comparison of the run time and memory usage of two approaches, on a 4×4 board looking for a value 4 tile, using 3 bits to represent values.

³Or, following Corollary 7, $(k - 1, 2\ell \dots n^2 - k + 1)$.

6.3 Results

With an $N \times N$ board and M the value of the objective tile, our program answers positively for $N = 2, M = 4$ and $N = 3, M = 9$, confirming Zeegers' results on those boards. By Lemma 6 a higher value of M is not possible. As our program only works on square boards, we could not confirm $M = 12$ for a 3×4 board or other non-square cases.

Figure 11 shows the number of iterations and time needed for sweep lines to converge with $N = 3, M = 9$. The plots suggest an exponential growth in both iterations and time when comparing the sweep lines $(1 \dots k, 1)$, with higher values of k taking more time, and also an exponential or at least very steep growth when comparing the sweep lines $(k, 1 \dots \ell)$, with lower values of ℓ taking more time — at least for high values of k .

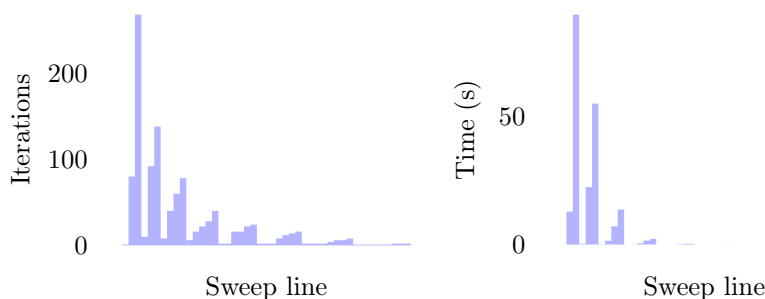


Figure 11: Plots showing the number of iterations and time needed for sweep lines to converge for $N = 3, M = 9$.

While running the program with $N = 4, M = 11$, while the sweep lines $(11, \ell)$ all converge in under a second, the sweep line $(10, 7)$ takes almost 20 seconds for 6 iterations and $(10, 6)$ takes almost 35 minutes for 22 iterations. Given the growth suggested by Figure 11, while we expect the sweep line $(10, 1)$ to be the largest hurdle, we do not deem it feasible to finish this computation within reasonable time.

When running the program with lower values of M , it does terminate in reasonable time, as shown in Figure 12 and Figure 13; for $M = 4$, the program returns positively in about 15 seconds, while for $M = 5$ it does so in about 30 minutes. This again does not lead us to believe that $M = 11$ would be feasible. However, the plots for $N = 4$ do give us some additional insights. The most iterations still occur for sweep lines with $\ell = 1$ — the single decrease from $(2, 5)$ to $(2, 4)$ seems to be an exception rather than a rule. Interestingly enough though, these are not the sweep lines taking the most time; that honour now goes to $(4, 2)$ and $(3, 3)$. This might actually also be suggested by Figure 11 where $(4, 2)$ and $(4, 1)$ both take 0.27 seconds and $(3, 3)$, $(3, 2)$ and $(3, 1)$ respectively take 0.02, 0.03 and 0.02, although these differences are too small to say anything conclusive. When taken with the rest of Figure 11, it does seem to suggest that the timewise importance of the sweep lines $(k, 1)$ decreases with the value of k .

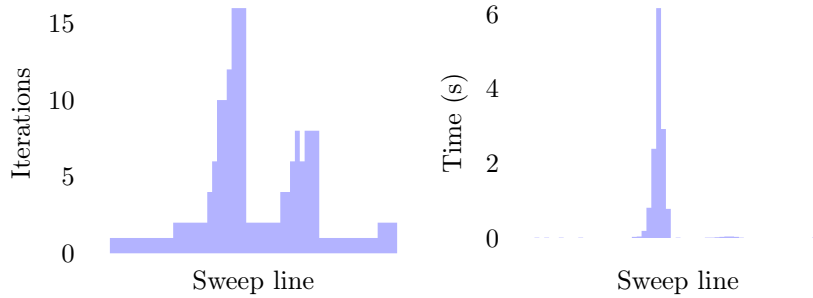


Figure 12: Plots showing the number of iterations and time needed for sweep lines to converge for $N = 4, M = 4$.

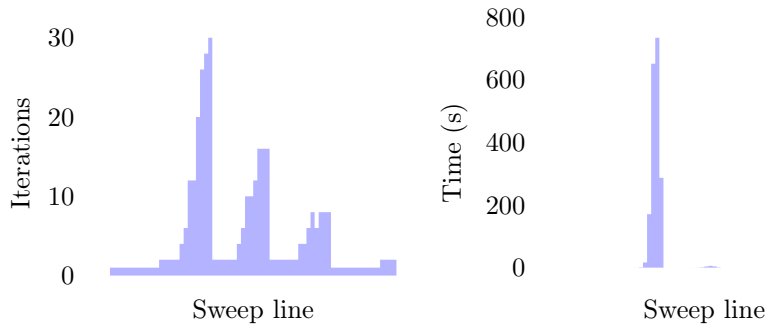


Figure 13: Plots showing the number of iterations and time needed for sweep lines to converge for $N = 4, M = 5$.

During the computation of a sweep line, the BDDs explode in size and finally converge back to a smaller size. An attempt to only save the ‘frontier’, the most recently found nodes, and propagate their predecessors instead of those of the entire sweep line only resulted in even larger BDDs and slowed down the computation further. Another observation is that, due to our implementation, we first construct the entire BDD of predecessors and only afterwards constrain it by conjuncting it with the current sweep line. It might be worth implementing a combined `preimage_and` operator (`relprev_and` in Sylvan) and a `forall_preimage_and` operator that apply the conjunction while constructing the predecessor instead of afterwards. However, as these BDDs have shown to typically not be that different in size, we do not expect that this would speed up the program by more than a (small) constant factor, if at all significantly.

The compression rate during the program seems to vary a lot, as shown in Table 4. While the compression rate for the first sweep lines with $N = 4, M = 11$ is impressive, storing hundreds of thousands of boards per byte, the results for $N = 3, M = 9$ suggest that this rate will keep decreasing as the program progresses. As the memory usage of the program scales linearly with the number of nodes in the hash table, Table 4 also shows the program’s memory bottlenecks:

with $N = 4, M = 11$, the sweep line (10, 6) already requires a single BDD with a size of approximately 1.0GB. We expect the corresponding number for the sweep line (10, 1) to be significantly larger.

N	M	When	Nodes	States	Bytes per state
3	9	Cumulative end result	$1.1 \cdot 10^7$	$3.6 \cdot 10^8$	0.749
3	9	Sweep line (8, 2)	$1.1 \cdot 10^6$	$3.6 \cdot 10^7$	0.746
3	9	Sweep line (8, 1)	$4.4 \cdot 10^6$	$4.5 \cdot 10^7$	2.371
3	9	Sweep line (7, 2)	$1.5 \cdot 10^6$	$1.2 \cdot 10^7$	2.929
4	11	Sweep line (10, 7)	$4.6 \cdot 10^5$	$7.5 \cdot 10^{12}$	$1.470 \cdot 10^{-6}$
4	11	Sweep line (10, 6)	$4.4 \cdot 10^7$	$7.5 \cdot 10^{13}$	$1.402 \cdot 10^{-5}$

Table 4: The compression rates at various points during execution. A node takes 24 bytes of memory in the Sylvan framework.

Finally, we give some ideas for possible improvements and alternative approaches to possibly speed up the computation:

- As mentioned before, it might be the case that the BDDs for intermediate results remain smaller if instead computing the prefixpoint F_2 , i.e., Dropper’s winning boards.
- It might prove more efficient to use other sweep lines. For instance, one could use sweep lines based on the total value of the board, i.e., the sum of the values of all tiles, instead of using the maximum value, to obtain a more evenly distributed state-space. Note that the total value of the board is directly correlated to the number of performed moves, so for a forward sweep from the initial state this is effectively a BFS with a sliding window. However, there is a difference when performing a backward sweep with retrograde analysis, which might be worth trying.
- A final option would be to view the problem as a hypergraph, where each outgoing edge targets a set of states instead of a single one. Xinxin Liu and Scott Smolka [19] give a linear time algorithm to compute the minimal fixpoint of a hypergraph. Adding the previous move to a state allows us to apply this to BDDs, which might be more efficient than our current approach using \forall preimage.
- As a more practical point, it might be possible that it is already not possible to always reach a value 10 tile. It might thus suffice to solve 2048 for a lower value of M to answer the question for $M = 11$, if there exists a lower value of M for which the algorithm returns false.

7 Conclusions and further research

In this final section we summarise our findings and results, and present some suggestions for further research.

7.1 Conclusions

We have given several properties of 2048 in Section 3 and used these to prove in Section 4 that it is not, in general, possible to reach a checkerboard pattern of alternating value 1 and 2 tiles, and give systematic ways to generate them on the boards where it is possible. This puts a lower bound on the minimal number of moves that a game of 2048 consists of before no further slides are possible. Finally, we suggest an upper bound on certain board sizes which is significantly lower than the one given by Zeegers in 2016; however, we give no strategy to reach this upper bound or even a proof of existence on general board sizes, so for now it remains a conjecture.

In Section 5 we have discussed how to apply fixpoint computation using binary decision diagrams (BDDs) to ‘solve’ 2048 — to determine whether it is always possible to reach a specific tile value, regardless of the opponent’s moves. Our encoding and implementation using a sweep line approach, described in Section 6, have shown to work on a 3×3 board and a 4×4 board with lower-valued objective tiles, but fail to clear the hurdles for solving the 4×4 board with a value 11 objective tile. We believe that this suggests that (most) multidimensional games do not scale well with BDDs, as we pointed out that our encoding lacks vertical locality due to the variable ordering. This view is strengthened by the similar results of Stef van Dijk’s 2019 master thesis on Othello [20]. We gather from personal communication that BDDs do scale well for at least some onedimensional games. That being said, we believe that our results also suggest that it might be possible to apply methods using symbolic representation, such as BDDs, to construct endgame databases and perform retrograde analysis up to some point even on divergent games, due to these methods’ ability to represent large sets of states in a single structure. In fact, in 2014 Stefan Edelkamp et al. proposed a hybrid approach using both symbolic and explicit representations to study the game Connect Four [12]. H. Jaap van den Herik et al. stated in 2002 [21] that divergent games are immune to retrograde analysis; given these new developments we would like to suggest to weaken this to ‘resistant’.

7.2 Further research

With regard to checkerboard patterns and short games, it might be interesting to attempt to find a sharp lower bound on the length of games on arbitrary board sizes; we believe that our preliminary results in Section 4.4 might be a good starting point for this. Furthermore, it might be possible to extend the properties described in Section 3, such as Lemma 10, and to some extent those in Section 4.3, to find similar, possibly more general properties for boards with higher-valued tiles, which might enable easier identification of (in)valid configurations, which might in turn be used to reduce the state-space of search

algorithms trying to analyse the game — or in other, possibly more general proofs about the reachability of certain patterns or about the game in general.

As for BDDs, we have stated before that there are many possibilities for alternatives and variations on our implementation. It might be possible to find a variable ordering more suited for representing result sets, leading to smaller BDDs; there might be other sweep lines that are more evenly distributed; we have only attempted to show the (non)existence of a least prefixpoint containing the initial state and the winning states for Slider, while it might turn out to be more efficient to compute this for Dropper — or there might be a way to find some (not necessarily least) prefixpoint for one of the two players that does *not* contain the initial state, although probably not purely computational. Furthermore, it might be possible to apply Liu and Smolka’s algorithm [19] to compute the minimal fixpoint of a hypergraph. Finally, with regard to 2048 in particular, it might be interesting to explore the possibilities for a hybrid symbolic and explicit representation approach, such as used by Edelkamp et al. on Connect Four [12].

References

- [1] Stefan Langerman and Yushi Uno. Threes!, Fives, 1024!, and 2048 are Hard. *Theoretical Computer Science* 748:17–27, 2018.
- [2] Naoki Kondo, Kiminori Matsuzaki. Playing Game 2048 with Deep Convolutional Neural Networks Trained by Supervised Learning. *Journal of Information Processing* 27:340–347, 2019.
- [3] Mathé Zeegers. Theoretical Properties of 2048. MSc thesis, Leiden Institute of Advanced Computer Science, Leiden University, 2016.
- [4] David Eppstein. Making Change in 2048. *Proc. 9th Int. Conf. Fun with Algorithms (FUN 2018)*. *LIPIcs* 100:21:1–12, 2018.
- [5] 2048 — Gabriele Cirulli
www.gabrielecirulli.com/2048
Accessed 2019-06-11.
- [6] THREES - A tiny puzzle that grows on you
www.asherv.com/threes
Accessed 2019-06-11.
- [7] Aaron N. Siegel. *Combinatorial Game Theory*. AMS, 2013.
- [8] L.V. Allis. *Searching for Solutions in Games and Artificial Intelligence*. Ph.D. Thesis, University of Limburg, Maastricht, 1994.
- [9] Stack Overflow — What is the optimal algorithm for the game 2048?
www.stackoverflow.com/a/22498940
Accessed 2019-04-02.
- [10] Sheldon B. Akers. Binary Decision Diagrams. *IEEE Transactions on Computers*, C-27(6):509–516, 1978.
- [11] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.
- [12] Stefan Edelkamp, Peter Kissmann, Martha Rohte. Symbolic and Explicit Search Hybrid through Perfect Hash Functions — A Case Study in Connect Four. *ICAPS*, 2014.
- [13] Donald E. Knuth. *The Art of Computer Programming Volume 4A: Combinatorial Algorithms, Part 1*. Addison-Wesley, 2011.
- [14] A. Tarski. A Lattice-Theoretical Fixpoint Theorem and its Application. *Pacific Journal of Mathematics* 5:285–309, 1955.
- [15] Richard Huybers and Alfons Laarman. A Parallel Relation-Based Algorithm for Symbolic Bisimulation Minimization. *VMCAI* 20:535–554, 2019.
- [16] Tom van Dijk and Jaco van de Pol. Sylvan: Multi-Core Decision Diagrams. *TACAS* 21:677–691, 2015.

- [17] Shin-ichi Minato. Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems. *DAC* 30:272–277, 1993.
- [18] Kurt Jensen, Lars Michael Kristensen, and Thomas Mailund. The Sweep-Line State Space Exploration Method. *Theoretical Computer Science* 429:169–179, 2012.
- [19] Xinxin Liu and Scott A. Smolka. Simple Linear-Time Algorithms for Minimal Fixed Points. *ICALP* 25:53–66, 1998.
- [20] Stef van Dijk. Solving Othello using BDDs. MSc thesis, Leiden Institute of Advanced Computer Science, Leiden University, 2019.
- [21] H. Jaap van den Herik, Jos W.H.M. Uiterwijk, Jack van Rijswijck. Games Solved: Now and in the Future. *Artificial Intelligence* 134:277–311, 2002.