2048 Without New Tiles Is Still Hard

Ahmed Abdelkader[1], **Aditya Acharya**[2], **and Philip Dasler**[3]

1 **Department of Computer Science, University of Maryland, College Park, Maryland 20742, USA**
akader@cs.umd.edu

2 **Department of Computer Science, University of Maryland, College Park, Maryland 20742, USA**
acharya@cs.umd.edu

3 **Department of Computer Science, University of Maryland, College Park, Maryland 20742, USA**
daslerpc@cs.umd.edu

Abstract

We study the computational complexity of a variant of the popular *2048* game in which no new tiles are generated after each move. As usual, instances are defined on rectangular boards of arbitrary size. We consider the natural decision problems of achieving a given constant tile value, score or number of moves. We also consider approximating the maximum achievable value for these three objectives. We prove all these problems are NP-hard by a reduction from 3SAT.

Furthermore, we consider potential extensions of these results to a similar variant of the *Threes!* game. To this end, we report on a peculiar motion pattern, that is not possible in *2048*, which we found much harder to control by similar board designs.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases Complexity of Games, 2048

Digital Object Identifier 10.4230/LIPIcs.FUN.2016.1

1 Introduction

2048 is a single-player online puzzle game that went viral in March 2014. The game is played on a 4×4 board as in Figure 1. Each turn, the player picks a move from $\{\leftarrow, \rightarrow, \uparrow, \downarrow\}$ to slide all tiles on the board. Tiles slide as far as possible in the chosen direction until they hit either another tile or an edge of the board. When a sliding tile runs into a stationary one of equal value, they merge into a tile of double this value. Trailing tiles following a tile that just merged continue to slide uninterrupted and may merge among themselves as they come to rest one after the other. However, newly merged tiles cannot merge again in the same move. After each move, a *2* or *4* tile is generated in one of the empty cells. The player wins when a *2048* tile is created, hence the name of the game. Otherwise, the player loses when the board is full and no merges can be performed.

2048 combines features from two families of games: Candy Crush Saga [6] and PushPush [5]. Prior to our work, an attempt to prove that *2048* is PSPACE-complete appeared in [9]. On the other hand, a proof of membership in NP appeared in a blog post by Christopher Chen [4], which applies to the games we study in this paper. The first draft of this work was made available in [3] and a revised version was presented in the Computational Geometry: Young Researchers Forum (CG:YRF) [2], held in conjunction with The 31st Symposium on Computational Geometry. In the meanwhile, another draft came out by Langerman and Uno

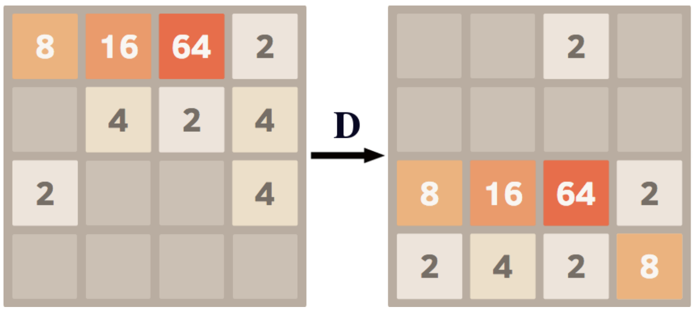

Figure 1 The result of taking the **Down** action (↓). Notice that the two *4*s on the right have merged and a new *2* has been randomly inserted.

[7] establishing the NP-completness of the original *2048* game, with new tiles generated after each move. Their construction extends easily to similar games including *Threes!*.

The game we analyze differs from the original *2048* game in the following two aspects: (1) The input encodes the complete board configuration and no new tiles are generated. (2) The board is a rectangular grid of arbitrary size. In this paper, we are primarily concerned with the following decision problem:

▶ **Definition 1.** (2048-TILE) Given a configuration of tiles on an $m \times n$ board, is it possible to obtain a tile of value 2048? (More generally, 2^k for a constant integer $k \geq 8$.)

Our construction can be augmented to study two related decision problems: 2048-SCORE and 2048-MOVES. The former asks if it is feasible to achieve a given score and the latter asks if it is feasible to achieve a given number of moves. As in [8], we define the score as the sum of all new tiles the player creates by merges during the game. In our setting, the number of moves is taken to mean the number of effective moves that change the board by merging at least two tiles.

The crucial piece of proving membership in NP is to bound the number of moves between two consecutive merges. Using a canonical orientation [4], all moves are interpreted as flips of the board, which send tiles along orbits of $O(mn)$ length. A pair of tiles that end up merging requires no more than $LCM(O(mn), O(mn)) = O(m^2n^2)$ moves.

In this paper, we prove NP-hardness by a reduction from 3SAT and obtain the main result.

▶ **Theorem 2.** 2048-TILE *is NP-Complete.*

Using the same reduction, we obtain similar results for both 2048-SCORE and 2048-MOVES. Furthermore, encouraged by the inapproximability results in [8] for the maximization version of these problems, where a new tile is generated after each move, we show similar results in our setting without new tiles. We implemented our gadgets and full reduction as an online game to aid the presentation [1].

2 Reduction from 3SAT

Given an instance of 3SAT with n variables and m clauses, we produce an instance of 2048-TILE. The board is filled using a 2-4 lattice to provide a rigid base for placing gadgets and planning their movements. We allow no merges using lattice tiles, which requires preserving their parity. This confines all merges to *blocks*, where a block is defined as a 2×2 arrangement of tiles. We typically use the words *row* and *column* to denote two consecutive rows or columns, respectively. In devising the gadgets presented here, we experimented with different lattice patterns before we settled for this one. We would like to note however that the 2-4 lattice was first described in [9].

In the rest of this section, we describe the gadgets we use in the reduction. A full annotated reduction is shown in Figure 2.

2.1 Displacers

These are the building blocks of all gadgets which allow us to communicate signals across the board. They come in two main forms: horizontal D and vertical D^T. Typically, a displacer starts in an *inactive* state where the middle 2×2 block, highlighted below, is shifted by one (or two) blocks along the axis orthogonal to the displacer's axis of action. An inactive displacer cannot merge, by any sequence of moves, before it is activated. The only way to activate a displacer is to use another properly aligned displacer to engage its middle block. Collapsing tiles in a displacer shrinks it to a single block, which results in a *parity-preserving pull* in a *row* or a *column*.

$$D = \left[\begin{array}{c|cc|c} 8 & 8 & 16 & 16 \\ 32 & 32 & 64 & 64 \end{array}\right]$$

2.2 Variable Gadget

Each variable is represented by two horizontal displacers on the same *row*. This enables variables to move the portion of its *row* between their two displacers to the right or left. We enforce the assignment of variables in the order of their indices. A variable is assigned T or F using a $\rightarrow$ or $\leftarrow$ move, respectively. The displacers of x_0 come activated in the initial configuration to allow the game to start.

To activate the variable gadget of x_{i+1}, two *connector displacers* are placed in the *row* of x_i. These connectors are activated regardless of the chosen truth assignment of x_i and allow the two displacers of x_{i+1} above them to be activated by a $\downarrow$ move in the following turn.

The variable gadgets of x_i for $i \in \{0, 1, 2, 3\}$ are annotated in Figure 2. Note that each variable has one gadget on the left and another on the right.

2.3 Clause and Literal Gadgets

A clause occupies a single *column* with a distinguished block near the top. Satisfying the clause corresponds to pulling down this block, which will be made possible by literal gadgets in the center of the reduction. Clause gadgets can be seen at the top of Figure 2.

Literals are encoded using a similar mechanism to the connectors in the variable gadget, but are only activated by the appropriate assignment. This is achieved by a *connector lattice*. Each active literal is a vertical displacer. Observe that both permutations of the columns of such a displacer can equally perform the required downward pull. We call this the *parity* of the displacer. The reduction uses the appropriate parity to distinguish positive and negative literals. A displacer may only be activated by providing a middle block of the same parity. As such, the displacers of positive (negative) literals will have positive (negative) parity and will only be activated by positive (negative) blocks in the connector lattice when the variable in question is assigned `True` (`False`) by a $\rightarrow$ ($\leftarrow$) move. For each clause *column*, a complete literal displacer is only included for each of the three rows corresponding to the three variables of the literals making up each clause. Otherwise, the connector blocks will have no effect on this *column*.

With variables as *rows* and clauses as *columns*, literal gadgets are situated exactly at the intersection of these *rows* and *columns* to communicate the relevant signals. This is easily seen at the center of Figure 2.

2.4 Key-Lock Gadget

To check that all clauses are satisfied, it helps to arrange for a special event to happen only after all variables have been assigned. To achieve this, an auxiliary variable $x_{aux} = x_n$ activates the lock portion of this gadget. Satisfying all clauses corresponds to using the correct key. Together, the activated key-lock gadget is a sequence of displacers that can activate a distinguished displacer with two special 2^{k-1} tiles. Collapsing this distinguished displacer creates the desired 2^k tile.

The lower portion of the lock gadget is composed of a sequence of $m+1$ vertical displacers that can only be activated by x_{aux}. When activated, the lock, in turn, activates a sequence of up to m horizontal displacers, at the upper portion of the gadget, utilizing the blocks of satisfied clauses.

As the lock gadget overlaps all clause *columns*, it must not be affected when any one clause is satisfied. To achieve this, the middle blocks of the vertical displacers in the lock gadget are shifted by two blocks away. This means that each middle block for such a displacer will be nestled in the displacer next to it. In order for this to work, the parity of these displacers alternate such that they are only activated by the displacers of x_{aux}.

The lower portion of the lock gadget is aligned with the rows of x_{aux}, as can be seen in Figure 2. The distinguished block lies to the right on the *row* where satisfied clause blocks are pulled.

2.5 Core and Padding

All gadgets live in the center of the board produced by the reduction, which we call the *core*. A crucial invariant for our reduction to work is that any row or column may not move unless it has an active displacer. This requires that any gaps created by such displacers are immediately collected away on the next move.

To this end, the core is surrounded by a *padding* of tiles on both axes. This padding ensures that any number of gaps produced by merging tiles in such a reduction can be completely isolated in one corner away from the rows and columns containing any gadget.

Keeping in mind that the board produced by the reduction is initially full, a gap is created iff two tiles merge. As the number of active gadgets is $\Theta(m + n)$ and each gadget contributes a constant number of gaps, a padding of $\Theta(m + n)$ thickness suffices.

2.6 Properties of the Reduction

Size: As variables are stacked on top of each other all the way up to x_{aux} and the key-lock gadget, the number of rows is $\Theta(n)$. Then, each variable has to activate the connectors to the next variable. We get a pyramid shape with variable displacers on both sides and literals in the middle, plus the unique displacer taking up $2(m + 1)$ columns far to the right, for a total of $\Theta(m + n)$ columns. It follows that the total size of the reduction is $\Theta(n(m + n))$.

Game Play: When no merges happen, two consecutive moves in opposite directions leave the board unchanged, e.g., $[\leftarrow, \rightarrow, \leftarrow]$ is effectively reduced to $[\leftarrow]$. *Effective* moves alternate between horizontal and vertical. The alternation accumulates newly created gaps, resulting

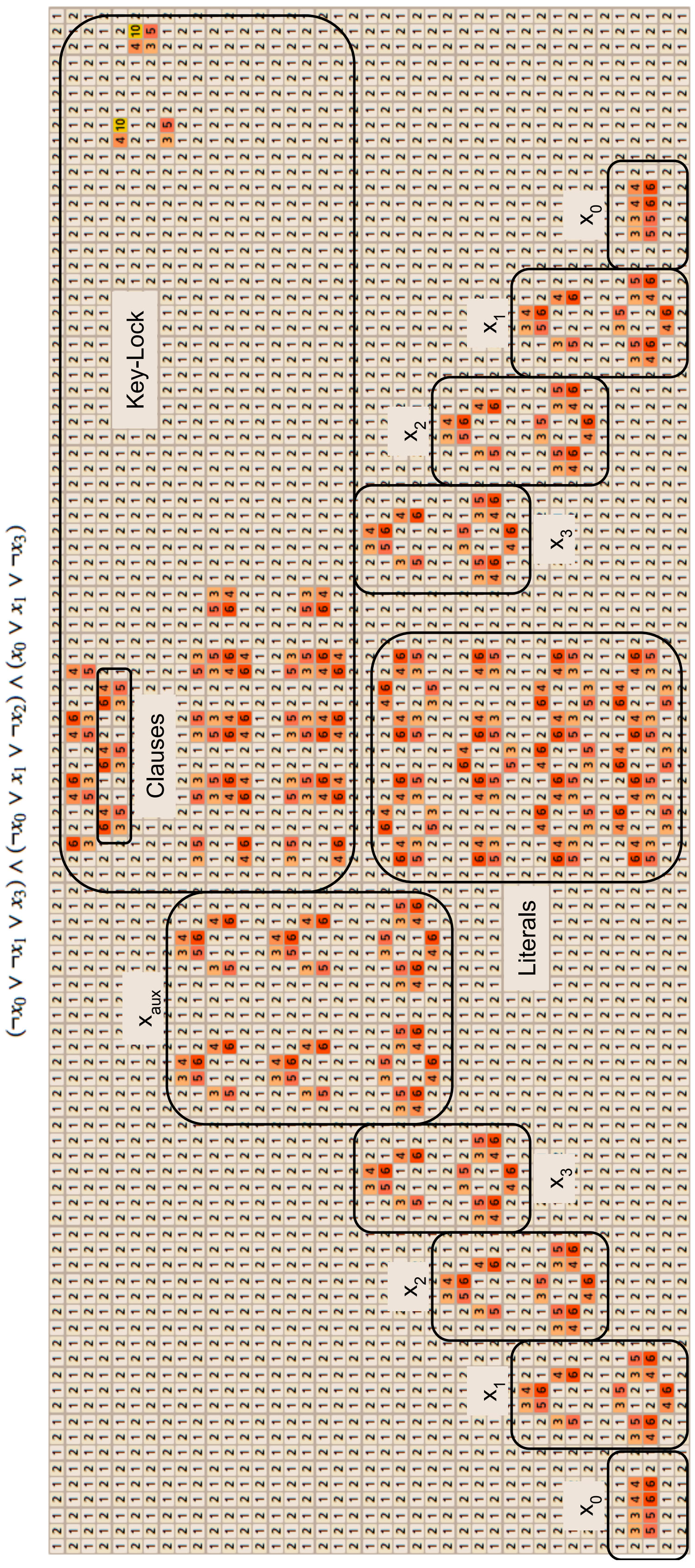
$(\neg x_0 \lor \neg x_1 \lor x_3) \land (\neg x_0 \lor x_1 \lor \neg x_2) \land (x_0 \lor x_1 \lor \neg x_3)$
Key-Lock
Clauses
x_{aux}
Literals
x_0
x_1
x_2
x_3
x_3
x_2
x_1
x_0

Figure 2 Annotated reduction. Only x_0 is active. We apply $\log_2$ and hide paddings to help display a large board. Note that this is only the core of the reduction, padding is not shown.

from the merge, at the corners so the decision encoded by the previous move cannot be altered. Furthermore, any row or column may witness merges during at most one turn. In particular, clause columns cannot experience more than one $\downarrow$ pull. This implies consistent assignments. Finally, $\uparrow$ moves are useless since they must be canceled or otherwise the player cannot win. Figures 6 through 17 show a complete play sequence through the reduction in Figure 2.

Hardness: Aligning the two 2^{k-1} tiles requires a $2m$ shift, which only satisfied clauses can provide with each satisfied clause contributing 2. Hence, the 2^k tile can be created iff the 3SAT instance is satisfiable. This proves Theorem 2.

Furthermore, regardless of the truth assignment of variables, the player can always activate and collapse all variable gadgets, including x_{aux}, and the lower portion of the lock gadget. Doing so takes $2(n + 1)$ effective moves. If any clauses were satisfied by the chosen assignment, the player will be able to perform one additional move and collapse up to m horizontal displacers in the upper portion of the lock gadget. Only if all m displacers were collapsed will the player be able to make one last move and collapse the special displacer. In such a winning sequence, there will be a total of $4n + 3m + 5$ active displacers plus 1 special displacer. Merging all these displacers takes $2(n + 2)$ moves and creates a set of new tiles with a total score $(2^4 + 2^5 + 2^6 + 2^7)(4n + 3m + 5) + (2^4 + 2^5 + 2^6 + 2^k)$. Again, this is possible iff the 3SAT instance is satisfiable. It follows that 2048-MOVES and 2048-SCORE are both NP-complete.

3 Inapproximability

Rather than placing a 2^{k-1} tile in the special displacer, we can use a normal displacer that activates a *pot of gold* gadget. In this section, we present two such gadgets: one for MAX-2048-MOVES and another for both MAX-2048-TILE and MAX-2048-SCORE. The size of the pot will be controlled by a parameter S. We let $K = n + m$, so the size of the 3SAT reduction is $N_0 = O(K^2)$.

3.1 Pot of Moves

A sequence of horizontal and vertical displacers can be added on top of the distinguished displacer as in Figure 3. Adding S such displacers allows S more moves.

Setting $S = 2^K$, the input size will be $N = \Theta((K+S)^2)$. If the 3SAT instance is satisfiable, the player can make $S = \sqrt{N} - O(\log N)$ moves, and $O(K) = O(\log N)$ otherwise. It follows that it is NP-hard to approximate MAX-2048-MOVES within a factor of $o(\sqrt{N}/\log N)$.

3.2 Pot of Value

Parity constraints are rather restrictive to allow merging a large number of blocks into one another, as required to create a tile of arbitrarily high value from a collection of constant value tiles. Instead, we opt to create a containment gadget where parity can be violated only inside it without disturbing the rest of the reduction. This allows us to align a large number of tiles into a single column where they can be collapsed into a single tile.

The design principle used here is to make two merges in a single column each of making an offset of exactly one. This means that the portion of the column between the two merges would experience an odd offset, altering its parity, while everything else in the board is unchanged. The same can be done for rows. Observe however that such a shift exposes an

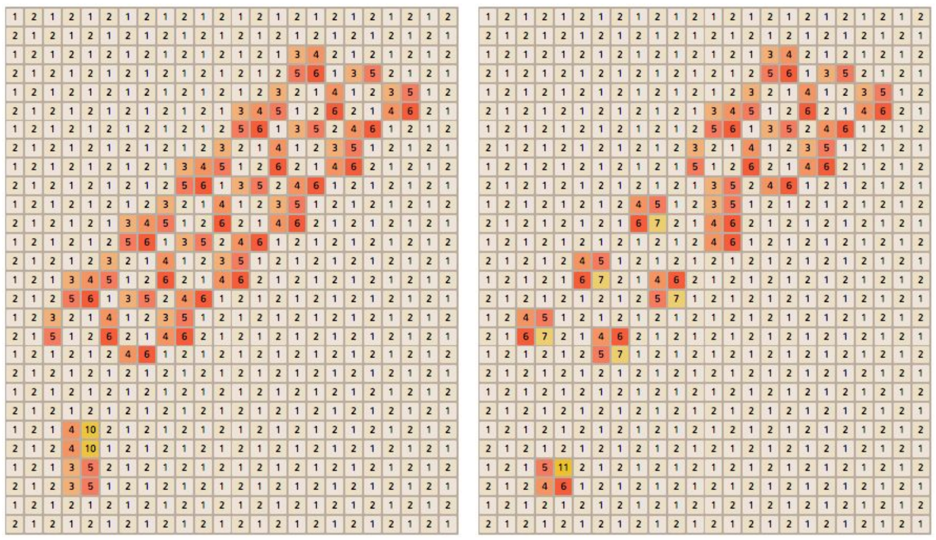

Figure 3 The Pot of Moves on top of the distinguished displacer (left) and after 6 moves (right).

even length portion of the neighboring rows and columns on both sides which have the same parity. To disallow such potential merges in lattice tiles, we need to make these shifts to an even number of consecutive rows and columns.

An example of such a gadget is shown in Figure 4. For $S = 2^p$, the gadget allows $p = \Theta(\log S)$ consecutive merges starting with 2^c tiles, for a constant c, and ending with tiles of value $2^{p+c} = \Theta(S)$. The total value of merged tiles add up to $\Theta(S)$ and the size of the augmented board will be $\Theta(K(K + S))$.

Setting $S = 2^K$, the input size will be $N = \Theta(K^2 + K2^K)$. If the `3SAT` instance is satisfiable, the player can create a tile of value $S = N/O(\log N) - O(\log^2 N)$, and 2^c otherwise, for a constant c. It follows that it is NP-hard to approximate `MAX-2048-TILE` within a factor of $o(N/\log N)$. Using the same parameters, if the `3SAT` instance is satisfiable, the player can achieve a score $S = N/O(\log N) - O(\log^2 N)$, and $O(K) = O(\log N)$ otherwise. It follows that it is NP-hard to approximate `MAX-2048-SCORE` within a factor of $o(N/\log^2 N)$.

4 The case for *Threes!*

For our purposes, the key difference between *Threes!* and *2048* is that in *Threes!*, tiles only move one step at a time instead of sliding all the way till they cannot go any further. It turns out that the difficulty of controlling such tiles comes from the presence of gaps. As in *2048*, gaps are created after each merge, but unlike *2048* where they are consumed on the next effective move, the gaps in *Threes!* persist for multiple moves. This allows the player to move these gaps to different locations creating a series of nontrivial shifts in the board.

We attempted a similar approach for *Threes!* in [3] and hoped that by spreading out the gadgets such a gap cannot travel from the row or column of the gadget that created it to a different row or column containing another gadget. For example, this may allow the player to make inconsistent truth assignments by altering a previously committed assignment of some variables or directly satisfying some clause without true literals. However, upon further examination we realized that the behavior of such gaps is richer than we thought.

For such a board game, a reduction like the one we use for *2048* alternates horizontal and vertical merges to communicate signals between the different gadgets. For *2048*, we did

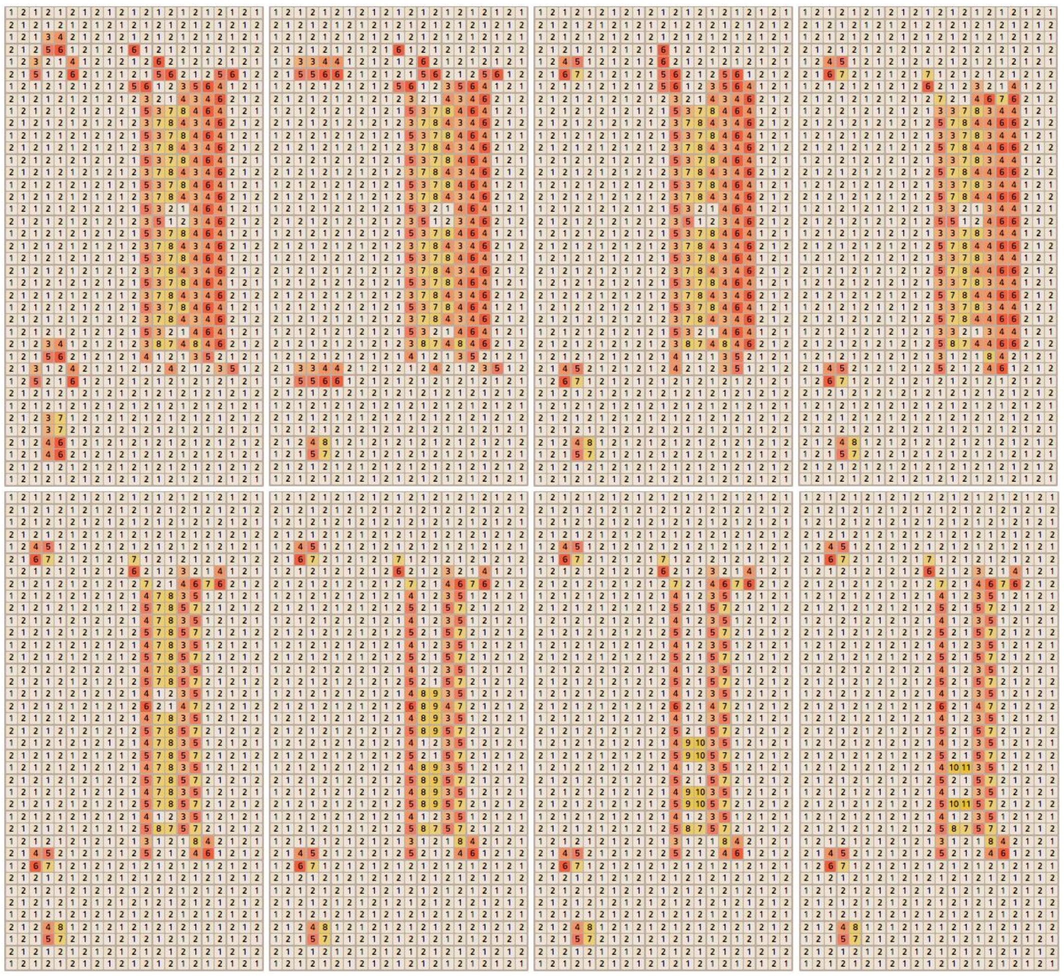

Figure 4 The Pot of Value: activation sequence and subsequent merges into a large value.

not need to keep track of how many gaps are created after each move or how to get rid of them. One possible solution that comes to mind is to create at most one gap per move and activate gadgets one by one as in [8]. However, this does not seem to resolve the issue as the example we discuss here shows.

Figure 5 shows the simplest such scenario. With just two gaps, one on the top side and one on the left side, the player can start a sequence of *clockwise moves*, cycling through [$\leftarrow, \uparrow, \rightarrow, \downarrow$]. One such full cycle is denoted by a winding arrow. This in turn creates two touching cycles of tiles that partition the board into two islands. Adding more gaps on either side results in a grid of islands separated by touching cycles.

One would need to show that the player gains nothing from such cycles. However, as the example shows, tiles that are initially far apart can come together after a number of rounds. The constraints on the lengths of such possible cycles and the safe locations of gadgets call for a more formal approach to the design. The example also shows an island that is not affected by the cycle. It would be interesting to explore whether one can exploit such protected islands to create gadgets that are easier to control and argue about.

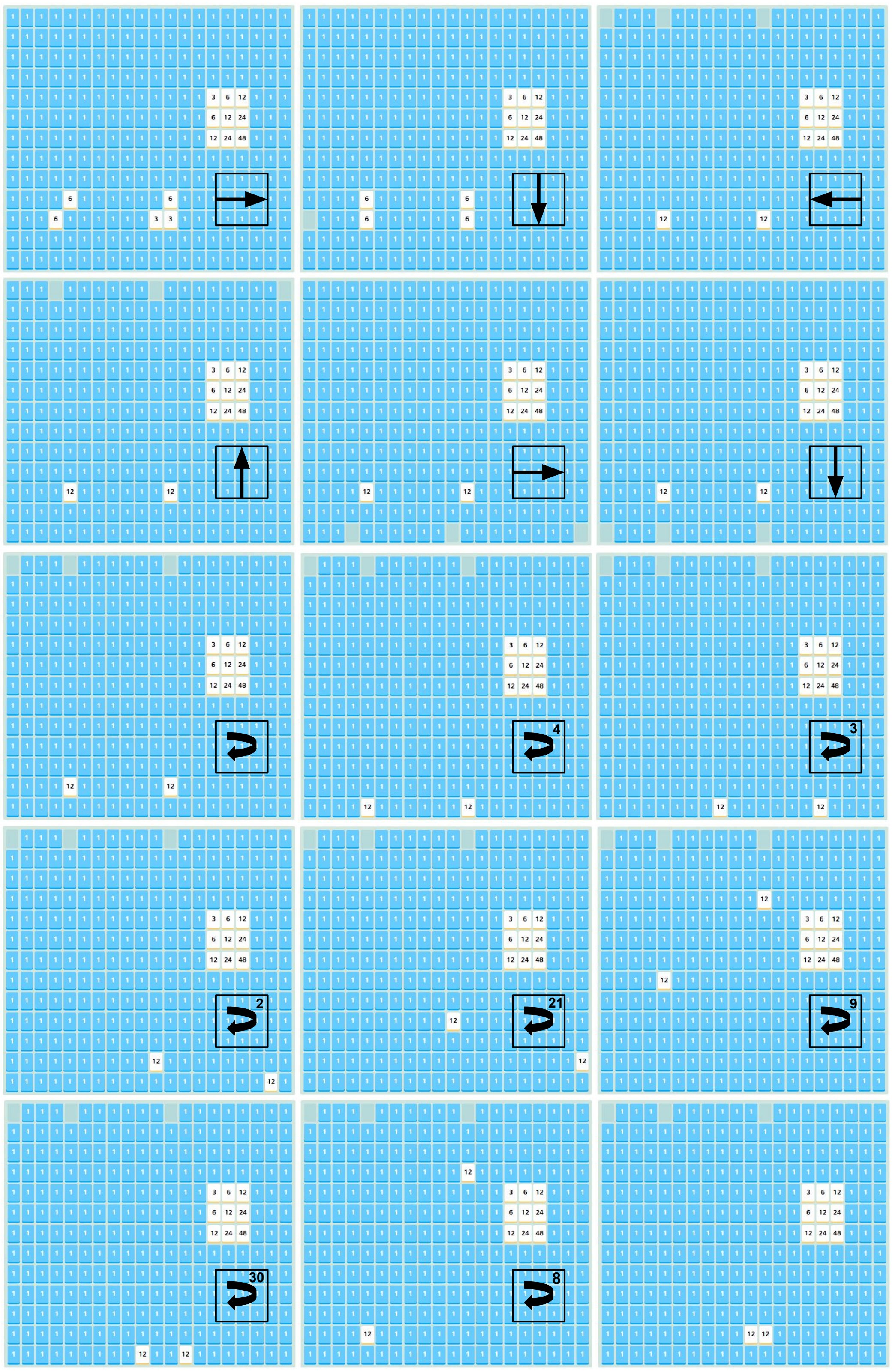

Figure 5 An example of curious motion patterns in the variant of *Threes!* without new tiles.

Acknowledgements. The authors would like to thank William Gasarch, Rahul Mehta, Erik Demaine and Stefan Langerman for helpful discussions through this work. The authors are especially grateful to MohammadTaghi Hajiaghayi for teaching the class that provided the opportunity to work on this project. The authors wish to thank the reviewers of the Computational Geometry: Young Researchers Forum (CG:YRF) for their valuable input that helped improve the presentation. Finally, the authors acknowledge Zhao HG and Angela Li for their open source implementations of *2048* and *Threes!*, respectively, and Christopher Chen for his proof.

References

1 Ahmed Abdelkader. 2048 gadgets. `http://cs.umd.edu/~akader/projects/2048/index.html`.

2 Ahmed Abdelkader, Aditya Acharya, and Philip Dasler. 2048 is NP-Complete. *CGYRF*, 2015.

3 Ahmed Abdelkader, Aditya Acharya, and Philip Dasler. On the Complexity of Slide-and-Merge Games. *CoRR*, abs/1501.03837, 2015. URL: `http://arxiv.org/abs/1501.03837`.

4 Christopher Chen. 2048 is in NP. `http://blog.openendings.net/2014/03/2048-is-in-np.html`.

5 Erik D. Demaine, Martin L. Demaine, and Joseph O'Rourke. PushPush is NP-hard in 2D. *CoRR*, cs.CG/0001019, 2000. URL: `http://arxiv.org/abs/cs.CG/0001019`.

6 Luciano Guala, Stefano Leucci, and Emanuele Natale. Bejeweled, Candy Crush and other Match-Three Games are (NP-)Hard. In *2014 IEEE Conference on Computational Intelligence and Games*, pages 1–8. IEEE, 2014.

7 Stefan Langerman and Yushi Uno. Threes!, Fives, 1024!, and 2048 are Hard. *CoRR*, abs/1505.04274, 2015. URL: `http://arxiv.org/abs/1505.04274`.

8 Stefan Langerman and Yushi Uno. Threes!, Fives, 1024!, and 2048 are Hard. In *8th International Conference on Fun with Algorithms (FUN 2016)*, volume 49 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 22:1–22:14, 2016.

9 Rahul Mehta. 2048 is (PSPACE) Hard, but Sometimes Easy. *CoRR*, abs/1408.6315, 2014. URL: `http://arxiv.org/abs/1408.6315`.

A Appendix

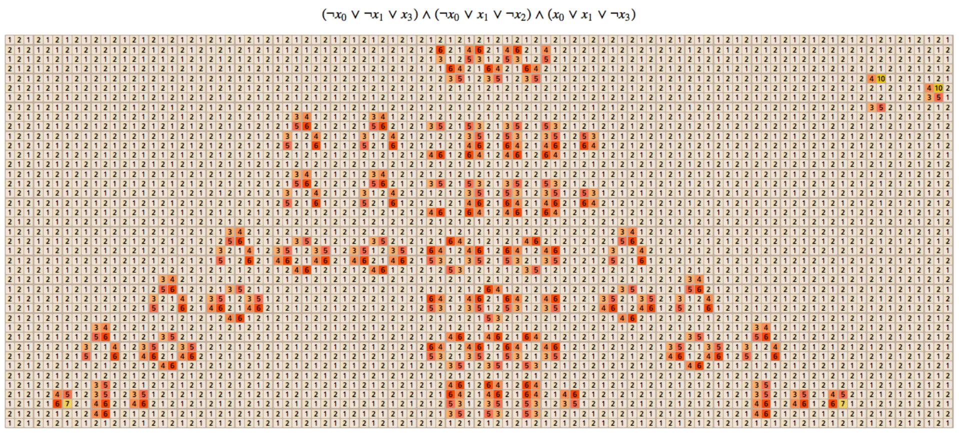
$(\neg x_0 \vee \neg x_1 \vee x_3) \wedge (\neg x_0 \vee x_1 \vee \neg x_2) \wedge (x_0 \vee x_1 \vee \neg x_3)$

Figure 6 Board(1): *move*($\rightarrow$). x_0 assigned T.

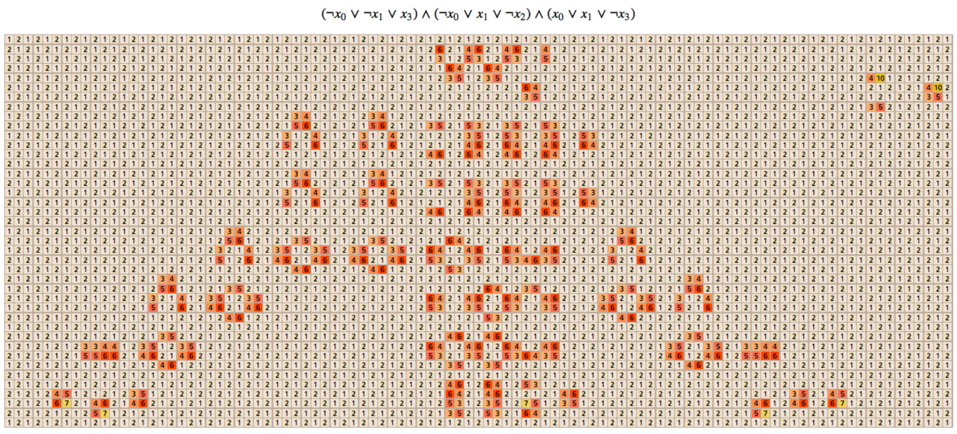
$(\neg x_0 \vee \neg x_1 \vee x_3) \wedge (\neg x_0 \vee x_1 \vee \neg x_2) \wedge (x_0 \vee x_1 \vee \neg x_3)$

Figure 7 Board(2): $move(\downarrow)$. c_2 satisfied. x_0 fixed to T. x_1 activated.

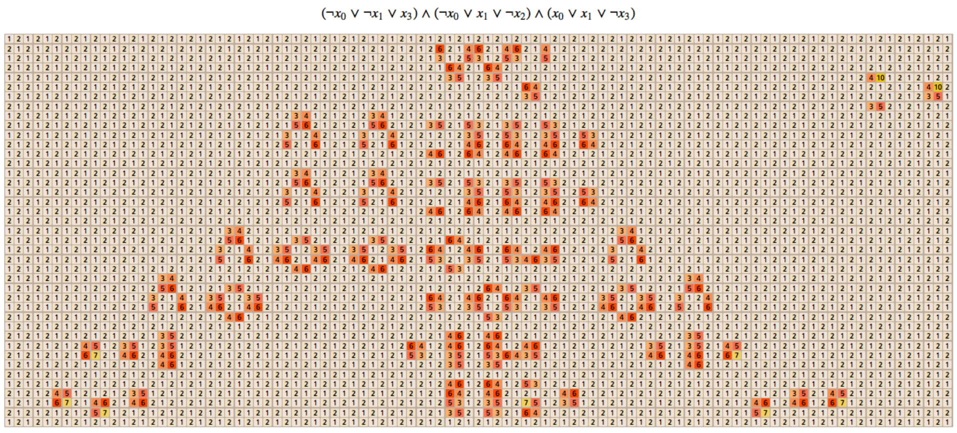
$(\neg x_0 \vee \neg x_1 \vee x_3) \wedge (\neg x_0 \vee x_1 \vee \neg x_2) \wedge (x_0 \vee x_1 \vee \neg x_3)$

Figure 8 Board(3): $move(\leftarrow)$. x_1 assigned F.

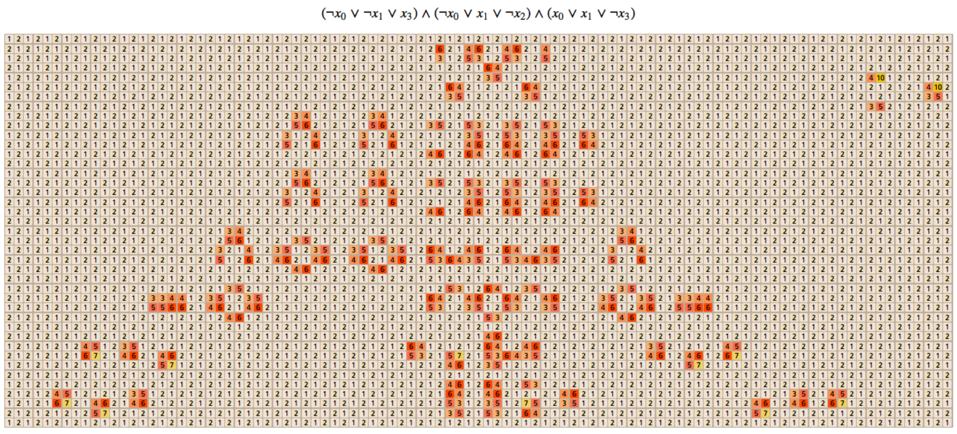
$(\neg x_0 \vee \neg x_1 \vee x_3) \wedge (\neg x_0 \vee x_1 \vee \neg x_2) \wedge (x_0 \vee x_1 \vee \neg x_3)$

Figure 9 Board(4): $move(\downarrow)$. c_0 satisfied. x_1 fixed to F. x_2 activated.

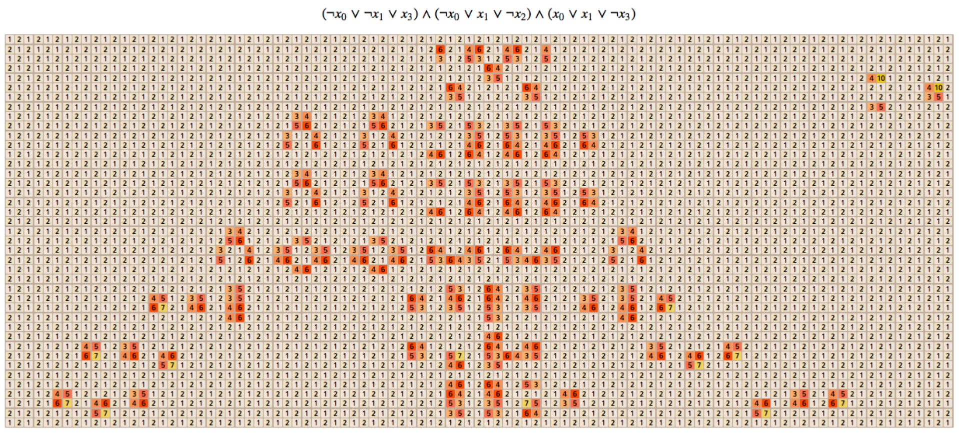
$(\neg x_0 \lor \neg x_1 \lor x_3) \land (\neg x_0 \lor x_1 \lor \neg x_2) \land (x_0 \lor x_1 \lor \neg x_3)$

Figure 10 Board(5): $move(\leftarrow)$. x_2 assigned F.

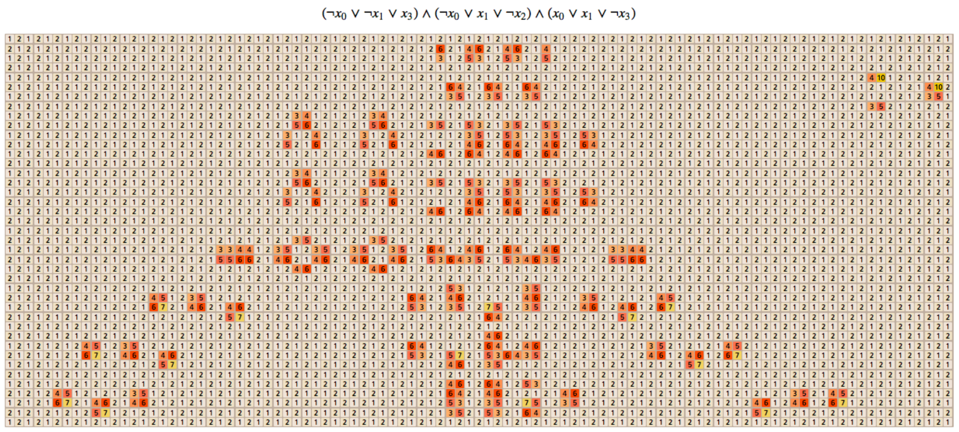
$(\neg x_0 \lor \neg x_1 \lor x_3) \land (\neg x_0 \lor x_1 \lor \neg x_2) \land (x_0 \lor x_1 \lor \neg x_3)$

Figure 11 Board(6): $move(\downarrow)$. c_1 satisfied. x_2 fixed to F. x_3 activated.

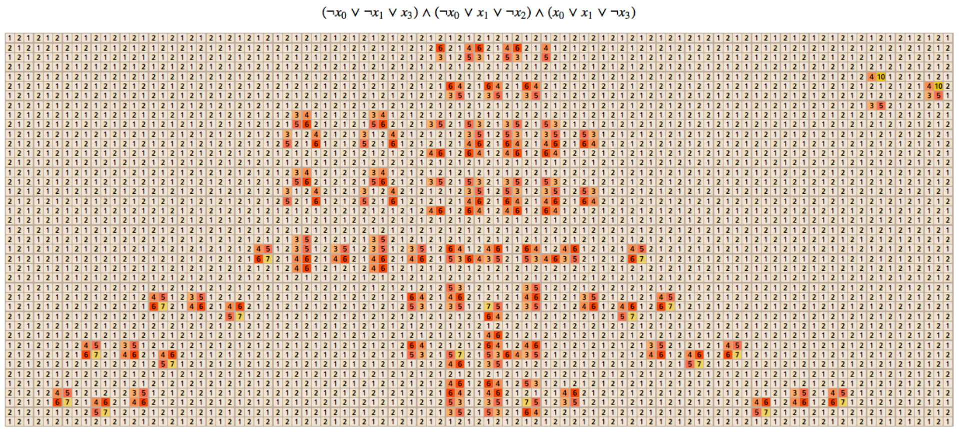
$(\neg x_0 \lor \neg x_1 \lor x_3) \land (\neg x_0 \lor x_1 \lor \neg x_2) \land (x_0 \lor x_1 \lor \neg x_3)$

Figure 12 Board(7): $move(\rightarrow)$. x_3 assigned T.

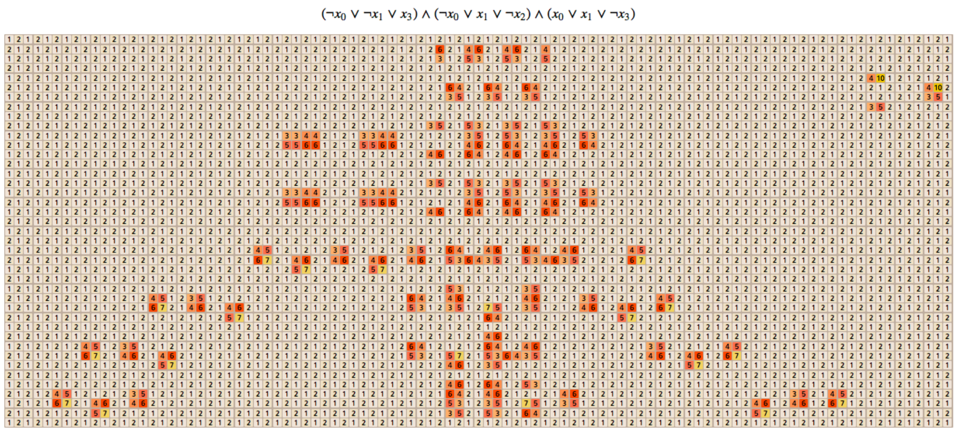
$(\neg x_0 \vee \neg x_1 \vee x_3) \wedge (\neg x_0 \vee x_1 \vee \neg x_2) \wedge (x_0 \vee x_1 \vee \neg x_3)$

Figure 13 Board(8): $move(\downarrow)$. x_3 fixed to T. x_{aux} activated.

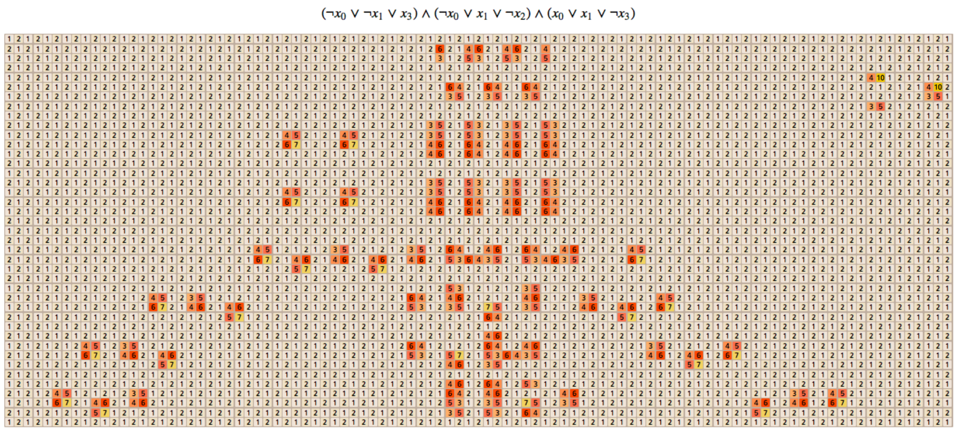
$(\neg x_0 \vee \neg x_1 \vee x_3) \wedge (\neg x_0 \vee x_1 \vee \neg x_2) \wedge (x_0 \vee x_1 \vee \neg x_3)$

Figure 14 Board(9): $move(\leftarrow)$. Key-lock sequence initiated.

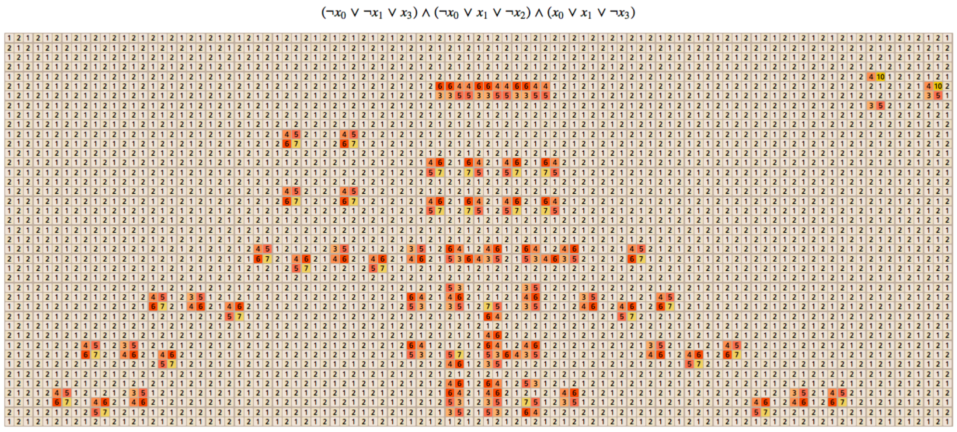
$(\neg x_0 \vee \neg x_1 \vee x_3) \wedge (\neg x_0 \vee x_1 \vee \neg x_2) \wedge (x_0 \vee x_1 \vee \neg x_3)$

Figure 15 Board(10): $move(\downarrow)$. Key-lock ready.

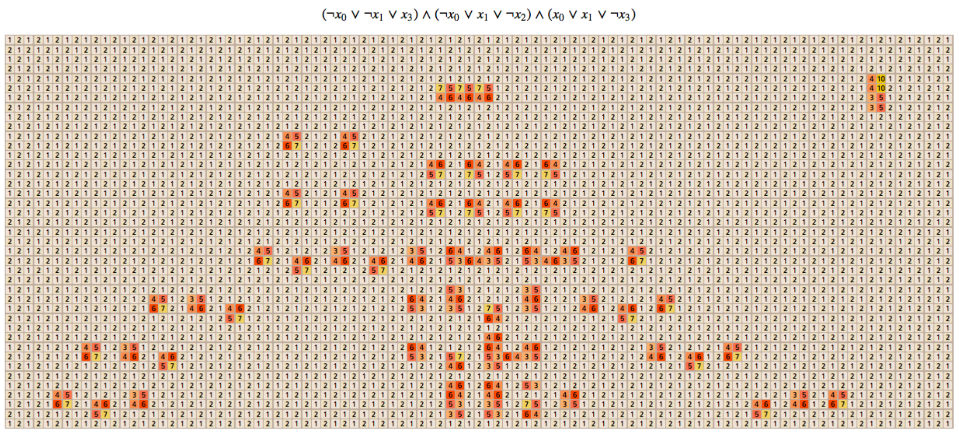
$(\neg x_0 \vee \neg x_1 \vee x_3) \wedge (\neg x_0 \vee x_1 \vee \neg x_2) \wedge (x_0 \vee x_1 \vee \neg x_3)$

Figure 16 Board(11): *move*(←). All clauses satisfied? Unlock.

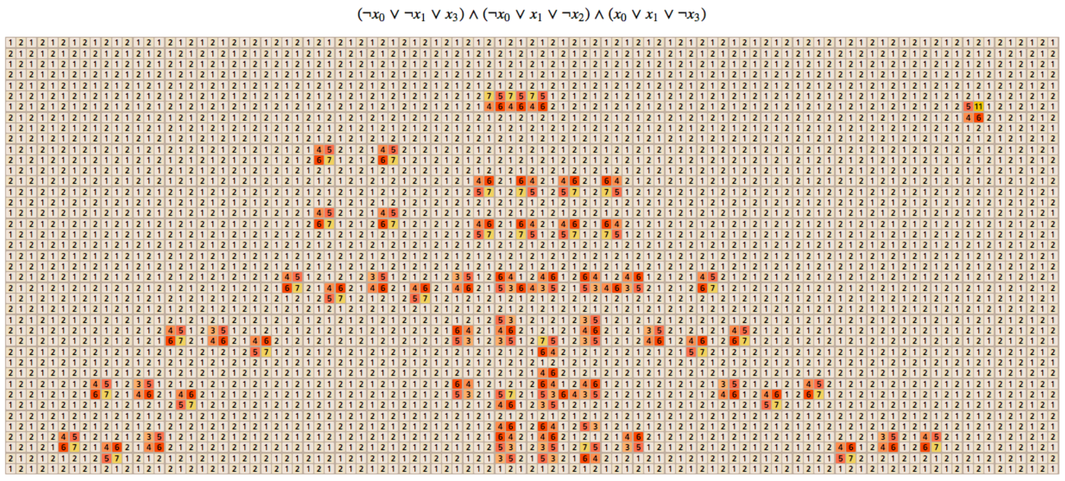
$(\neg x_0 \vee \neg x_1 \vee x_3) \wedge (\neg x_0 \vee x_1 \vee \neg x_2) \wedge (x_0 \vee x_1 \vee \neg x_3)$

Figure 17 Board(12): *move*(↓). Win.