EPiC Series in Computing

Volume 46, 2017, Pages 162–170

LPAR-21. 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning

Quantified Boolean Formulas: Call the Plumber!

Josef Lindsberger[1], Alexander Maringele[2], and Georg Moser[2]

[1] Department of Computer Science, Universitt Innsbruck, Austria
josef.lindsberger@web.de
[2] Department of Computer Science, Universitt Innsbruck, Austria
{alexander.maringele,georg.moser}@uibk.ac.at

Abstract

In this tool paper we describe a variation of Nintendo's Super Mario World dubbed Super Formula World that creates its game maps based on an input quantified Boolean formula. Thus in Super Formula World, Mario, the plumber not only saves his girlfriend princess Peach, but also acts as a QBF solver as a side. The game is implemented in Java and platform independent. Our implementation rests on abstract frameworks by Aloupis et al. that allow the analysis of the computational complexity of a variety of famous video games. In particular it is a straightforward consequence of these results to provide a reduction from QSAT to Super Mario World. By specifying this reduction in a precise way we obtain the core engine of Super Formula World. Similarly Super Formula World implements a reduction from SAT to Super Mario Bros., yielding significantly simpler game worlds.

1 Introduction

Video games have established themselves over the last decades to be a huge part of society. Not only as a funny way to spend spare time but also as interesting and challenging programming tasks and as an interesting field in complexity theory.

For example the computational complexity of Minesweeper [6], Sokoban [3], various Nintendo games, (for example Super Mario Bros) [1, 4], Candy Crush [11, 12], and Lemmings [10] has been analysed. To this avail in recent years several schemes (or frameworks) for the analysis of the computational complexity of famous video games have been proposed. Based on pioneering work by Viglietta [9], Aloupis et al. [1] have introduced two such frameworks, one reducing from SAT yielding NP hardness of the game subject to the framework and one reducing from QSAT, which yields PSPACE hardness.

In this tool paper, we describe a variation of Nintendo's Super Mario World (SMW for short), dubbed Super Formula World (SFW for short), that creates its game maps based on an input quantified Boolean formula. Thus in SFW, Mario, the plumber not only saves his girlfriend princess Peach, but also acts as a QBF solver as a side. The game is implemented in Java and platform independent. SFW is freely available at https://github.com/DwarfVader/mario.

The SNES game SMW was released in 1990 by Nintendo. The game extended Super Mario Bros to the then latest technical possibilities. In particular, several game elements like *climbing vines*, *rotating blocks*, *stone balls*, or *trampolines* were incorporated into the game. The presence

of these game elements allows the natural representation of *door gadgets* and consequently significantly increases the computational complexity of the game. In particular it is an easy consequence of Aloupis et al. that SMW is PSPACE complete, cf. [1]. This observation forms the theoretical underpinning of our game SFW.

We recall the frameworks introduced by Aloupis et al., where we focus on the framework for PSPACE hardness for brevity. This framework reduces from the satisfiability of quantified Boolean formulas (QSAT for short) to the reachability problem in graphs representing the given formulas. Thus playing a game suited in the formalism of the framework represents the separate and individual search for a satisfying assignment for existentially quantified variables in each possible assignment for preceding universally quantified variables [5] of a QBF formula φ: the formula φ is satisfiable if and only if the strategic and stubborn player manages to reach the finish location. In order to cast this idea into code, a more precise statement of the PSPACE framework proposed in [1] becomes necessary. Thus we slightly extend the framework. In particular we introduce a grid-like topology for the game maps that allows an efficient map generation and generalise the door gadgets suitably (see Section 3).

Then we instantiate the framework for SMW. As in the literature, the only modification of the game is the generalisation of map size and the assumption that the logical screen size captures the whole game map, that is, no parts of the game are ever reset to their initial state while playing. Based on this instantiation, we have implemented our game SFW in Java. For sprites we relied on the *The Spriters Resource*.[1] We summarise the contributions of this tool paper.

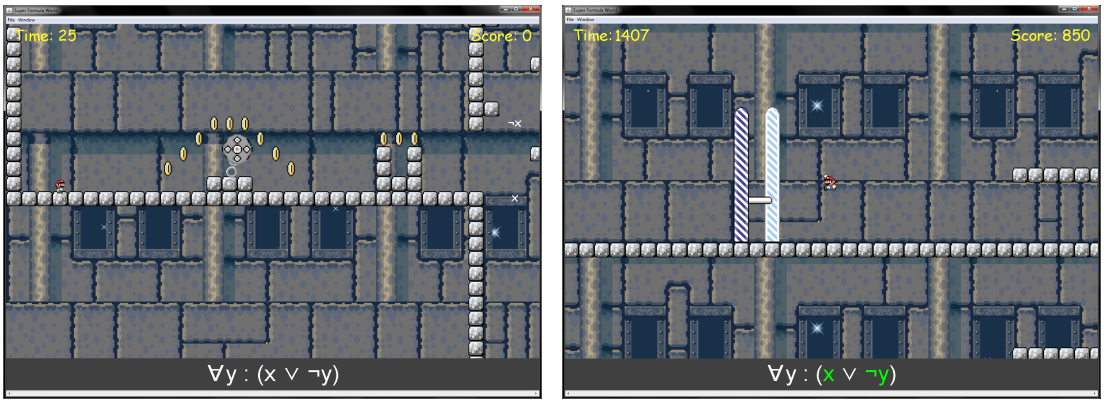
- Based on earlier work by Aloupis et al., we describe a framework to show PSPACE hardness of video games. We instantiate this framework for Super Mario World and establish PSPACE completenes (see Section 3).
- Our implementation of the framework, Super Formula World, takes as input a QBF formula φ in prenex form and renders an equivalent game map. The skilful and enduring player can successfully finish the level if and only if φ is satisfiable (see Section 4).
- The game is equipped with a standard GUI and allows specialisation to the framework for NP or PSPACE. Furthermore *cheat*, *challenge*, and *speed-up* modi are available to improve the user's experience (see Section 2).

In sum, we provide a (hopefully) entertaining video game that showcases the almost inscrutable hardness of QBF solving: In Super Formula World, Mario, the plumber, not only saves his girlfriend princess Peach, but also renders the solution to a highly intractable problem as a side.

For presentational purposes, we restrict our attention to QBF solving in this paper. However, we emphasise that SFW implements (precise versions of) both frameworks established by Aloupis et al. in [1]. If the game is started with a propositional formula only gadgets of the framework for NP are employed. In particular, in this case SFW implements a variant of the classic Super Mario Bros. as no features of SMW are employed. Thus, the generated game map of SFW precisely represents a SAT or QBF solver, respectively.

The above mentioned PSPACE completeness for SMW is superseeded by the recent results that already Super Mario Bros is PSPACE completeness, cf. [4]. This result is based on a clever encoding of *loop commands*, a game feature that makes Mario repeat earlier screens if he doesn't follow a specific path through the game. Our encoding does not make use of *loop commands*.

[1] See http://www.spriters-resource.com/.



(a) Start frame. (b) Finish frame.

Figure 1: Start and Finish for Super Formula World on $\forall y \ (y \to x)$.

Furthermore, unlike the here employed framework the hardness proof in [4] seems to crucially rely on assumed specifics of the physics engine in Super Mario Bros. In our point of view this complexity result is thus less suitable a basis for a *serious game*, like SFW.

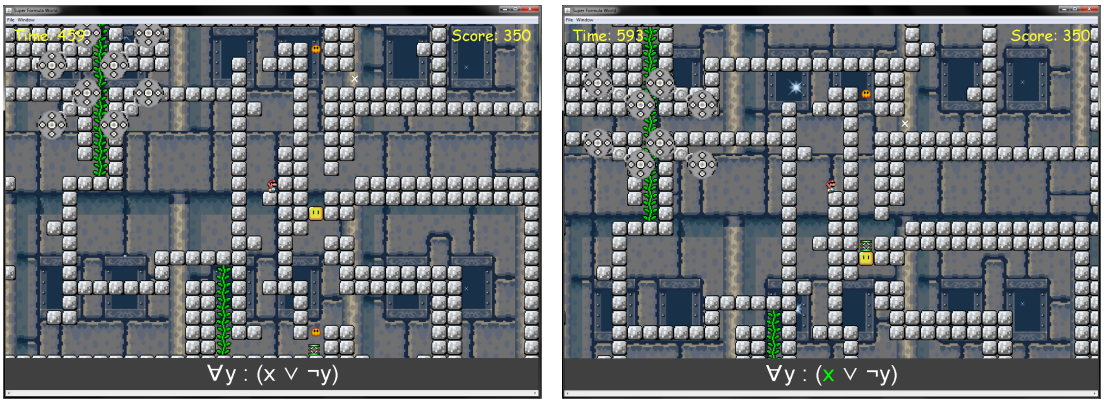
This tool paper is structured as follows. In the next section we showcase the developed video game in an example run. In Section 3, we present the theoretical basis of Super Formula World and comment on its implementation in Section 4. Finally, in Section 5, we conclude and mention future work.

2 Example Run

In this section an example run of our game Super Formula World is shown. The input to the game is a QBF in prenex form, allowing for standard notions for propositional connectives. We consider the QBF $\forall y \, (y \to x)$. The free variable x is treated as existentially quantified, that is, the generated game represents the prenex normal form $\exists x \forall y \, (x \vee \neg y)$.

The game offers three modi: *normal*, *moderate* and *challenging*. In *normal* mode the formula is displayed in prenex normal form, where the matrix is transformed into CNF. In *moderate* mode the original formula is displayed as the player entered it. During the game the assignments are highlighted in those modi. In *challenging* mode the formula is not shown, and the player has to keep track of the assignments herself. Furthermore various cheats are provided. Mario can be made invincible, allowed to jump repeatedly and potentially walk through walls. Furthermore the player can slow down or speed up Mario to a certain degree. For the example run we employ the *normal* mode.

Figure 1(a) shows Mario shortly after the start of the game. To make the game more entertaining, coins can be won that improve the score. These features are independent of the PSPACE hardness of the game. At the bottom of the screen the input formula is shown in prenex normal form. In particular the implicit existential quantifier $\exists x$ for the free variable x is not shown. On the right, one can see the entrance of the first quantifier gadget, in this case the one for $\exists x$. Mario decides to assign the variable x to true, which is achieved by falling down. In Figure 2(a), Mario reaches the door gadget representing literal x in the clause gadget from the left. Eventually Mario wants to traverse this gadget, coming from the upper middle path



(a) Door for literal x. (b) Door for literal x opened.

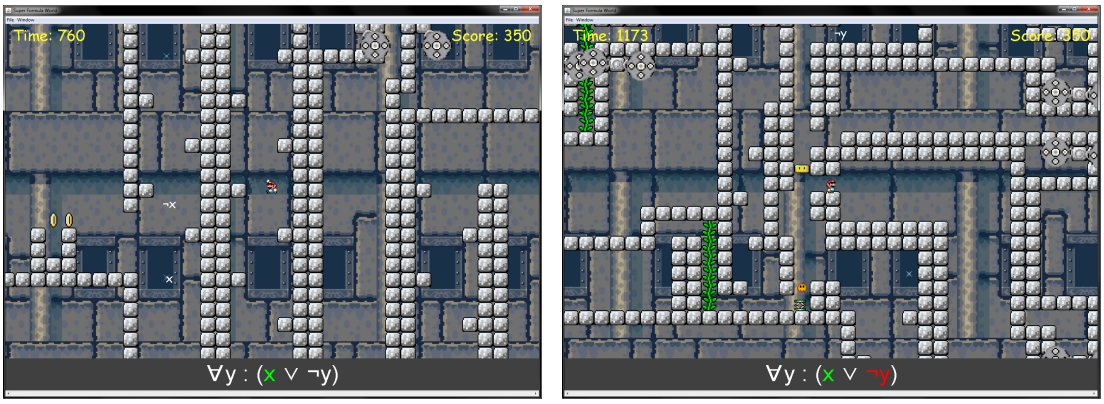
Figure 2: Door Gadget and Literal Path for Super Formula World.

on the right and leaving via the top right (see Section 3). However, this is at the moment not possible, as Mario cannot jump high enough to reach the top right exit. In order to do so, he needs a trampoline on top of the rotating block depicted in the middle of the figure. Currently the trampoline is at the bottom and below the rotating block, that is, the door is closed. In order to open the door, Mario fetches the trampoline from the bottom, and carries it via the left shaft to the top, so that he can throw it down the right shaft. It will land on top of the rotating block. The door is now open, and can be traversed eventually, see Figure 2(b). This is also indicated by marking the literal x in green. The literal path for x (see Figure 3(a)) leads Mario back to the existential quantifier gadget from which he started. Following internal paths through this gadget, Mario reaches the universal quantifier gadget for y, where he is forced to assign true to y and therefore has to close the door for literal $\neg y$ in the clause gadget. In Figure 3(b), Mario has already traversed the close path of that door. For that Mario had to jump against the rotating block from below which began turning, whence the trampoline falls to the bottom of the second shaft again. Thus the door is closed, and the literal $\neg y$ is set to false, which is shown by marking the literal $\neg y$ in red. As the (only) clause is satisfied, Mario can traverse the check path of this clause and head back to the universal quantifier gadget which now forces him to assign the variable y with false.

Repeating a similar sequence of steps as above, Mario now is able to traverse the check path for the clause $x \vee \neg y$ a second time, this time as the literal $\neg y$ is true. Finally, Mario reaches the finish (Figure 1(b)) and wins the level. On the side, he has acted as a QBF solver for the formula $\exists x \forall y\, (y \to x)$.

3 Super Formula World

As mentioned in the introduction, Aloupis et al. introduced a framework for PSPACE hardness in [1], which reduces from the PSPACE complete problem QSAT. The framework defines a graph which models the decision problem of reachability. Instantiations of this framework, for example for Donkey Kong Country and The Legend of Zelda, have been established. In this section we present a precise definition and slight extension of the framework suited to our needs. For brevity, we assume (at least nodding) acquaintance with the general construction in [1] and focus on the necessary changes in our construction after we have given a short general



(a) Literal path for x. (b) Door for literal $\neg y$ closed.

Figure 3: Door Gadget and Literal Path for Super Formula World (cont'd).

explanation of the framework.

The framework encodes a reduction from QSAT to a game represented as a graph that is built of gadgets. There are different kinds of gadgets, providing different kinds of behaviour. The framework combines them in such a way that the resulting graph represents a QBF φ and its solving process. The goal of the game is to reach a defined finish node, given a start node, by traversing the graph and making decisions, for example which one of two literal paths to follow at existential quantifier gadgets. Naturally these decisions represent the variable assignments in the formula. By traversing literal paths, door gadgets get visited and operated, that is, opened or closed, such that later regions of the graph get traversable. In particular, iff previous variable assignments satisfy a clause, the corresponding clause gadget becomes thus traversable. In our setting clause gadgets consist of multiple door gadgets, each representing a literal of the clause.

Universally quantified variables are handled via universal quantifier gadgets in such a way that both assignments have to be validated, one after the other. Therefore, these gadgets introduce a certain amount of structural complexity to the graph. A reroute through the graph is necessary, such that the intended recursive behaviour is achieved. In general paths in the graph might cross each other. These collisions need to be resolved by crossover gadgets. They allow traversing the crossing region without the possibility of leaking from one path to the other. In sum, the construction guarantees that the finish node is reachable iff the QBF φ is satisfiable. All the gadgets used in the framework have to fulfil one requirement: They have to be non-exhaustive, which means they may not change their behaviour upon usage. Non-exhaustive gadgets are needed to handle the repeating/recursive behaviour of the framework. In order to implement this framework, a few extensions are necessary. Most importantly we require a grid-like layout of the game map, where each gadget is assigned a two-dimensional location. Thus the location of each gadget becomes easily calculable based on the input QBF φ. We only sketch the layout in the following, full details can be found in [7]. The start and finish gadgets are located on the top left of the grid. To the right of these, the quantifier gadgets are placed as they appear in the prefix, while the clause gadgets are placed at the bottom in a row. The horizontal spacing is done so that the row of clause gadgets is to the right of the quantifier gadgets. Together with an order on the (quantified) variables, this set-up allows to compute the precise location of all gadgets, including crossover gadgets. Observe that such a precise layout is not required for the soundness of the framework. However, it simplifies the

implementation and as it minimises the number of crossover gadgets needed it also simplifies the correctness proof of the reduction.

Since the width of the map is bound by the number of quantifiers and clauses and the height by the number of quantifiers the size of the map is quadratically bound by the size of the formula in prenex CNF. Due the naive implementation within SFW the transformation to CNF may introduce an exponential blow-up. This could be easily fixed by the use of the linearly bound Tseitin transformation of satisfiability.

Furthermore, we emphasise the need for *locality* of door gadgets. The actions needed to change the state of the door must only have local effect. This means, everything that can be done in the open or close path is not allowed to alter the state of the game anywhere else. For instance, the trampoline in the Super Mario World instantiation in Figure 4 may not be carried out of the door gadget as it could be used to break the framework by opening later doors. We remark that our precise definition of the framework given in [1] highlighted minor (and easily correctable) shortcomings of the original framework, in particular in the framework for NP hardness. In particular the construction of the crossover gadgets for Super Mario Bros and Metroid requires more precise arguments, cf. [7, Chapter 3].

In the remainder of this section, we sketch the instantiation of the PSPACE framework to SMW[2] The crucial step in the application of the framework is to create door and crossover gadgets that follow the given game mechanics, using only non-exhaustive, local game elements without allowing any leakage. The straightforward construction of the other gadgets is left to the reader. Full details are given in [7, Chapter 4]. As an easy corollary we obtain that SMW is PSPACE-complete.

Figure 4: Door Gadget for SMW.

SMW offers a large set of game elements, though the implementation of the gadgets only requires a few, non-exhaustive ones: *climbing vines*, *rotating blocks*, *stone balls*, and *trampolines*. Mario can jump while climbing vines but he cannot climb while carrying something. If Mario jumps at a rotating block from below, the block begins to rotate and gets passable for a short amount of time, which allows trampolines to fall through.

Stone balls are placed at the end of a chain that rotates around a given point. If Mario gets hit by the stone ball, he dies. Trampolines can be picked up when approaching them horizontally, they can be carried around and dropped down shafts. Trampolines can be used arbitrarily often and allow Mario to bounce off and jump higher. There is no collision if Mario touches a trampoline from the bottom, in particular trampolines can fall through Mario without collision. We observe that if big Super Mario performs a spinning jump on top of the rotating block, it gets destroyed. Thus for the soundness of the construction it is essential to

[2]See http://en.wikipedia.org/w/index.php?title=Super_Mario_World&oldid=656286985.

guarantee that Mario is small throughout the course of the game. Thus super mushrooms are not allowed (and not needed) in the implementation of the framework.

Door gadget Note that the door gadget consists of three paths: (i) a path to open the door, (ii) a path to close the door and (iii) the path that actually traverses the door. Thus its functionality is split according to that paths. In Figure 4, the door gadget for SMW is shown.

Open path. Mario enters from the top left and jumps down the left shaft. At the bottom, Mario heads right where a trampoline blocks his way. Mario carries the trampoline via the left shaft to the top and throws it down the right shaft. The trampoline falls until it reaches the rotating block. Mario cannot leak to the traverse path via the right shaft as the fire on the top would kill him.

Mario again jumps down the left shaft and leaves via the bottom left. If the door is already in its opened state, Mario immediately leaves the gadget via bottom left. As already observed in [1], Mario can choose to leave the door in closed state without opening it. Leakage to the close path is not possible as the fire on the bottom would kill Mario. It is important to prevent the player from carrying the trampoline outside the gadget as a closed door could be opened via its traverse path by placing the carried trampoline. Therefore, a vine at the exit of the open path forces Mario to climb which is not possible while carrying a trampoline. This fulfils the locality constraint for actions needed to operate doors.

Traverse path. Mario enters from the upper middle path on the right and needs to jump up and leave via the top right. To overcome this height, the door has to be opened before by throwing the trampoline down the right shaft. Using the trampoline, Mario can reach the upper path and leave.

Leakage to the open path is not possible as there is fire placed on the top. Leakage down to the close path is not possible, as even if there is no trampoline, allowing Mario to reach the rotating block, Mario is small and cannot destroy it with a spinning jump. The traverse path represents the actual door: If it is traversable, the door is open, otherwise the door is closed.

Close path. Mario enters from the bottom right and jumps on the ledge at the left of the shaft to perform a second jump to reach the upper horizontal tunnel and leave the gadget. Mario's jump height forces him to jump against the rotating block before he can leave horizontally, such that the trampoline (if previously placed on the rotating block) falls to the ground.

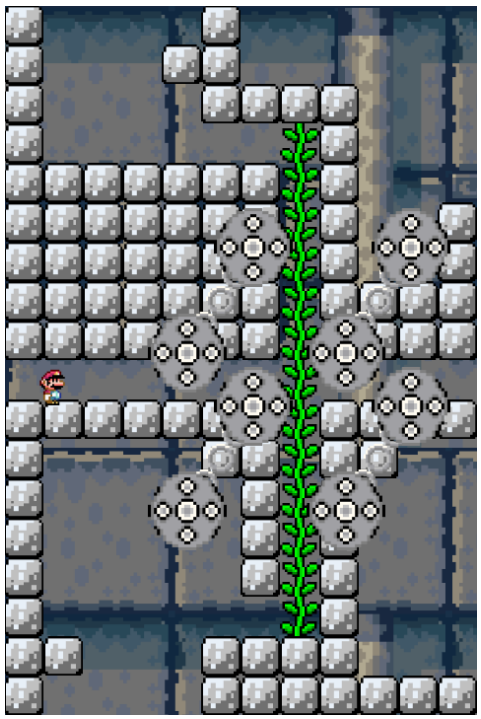


Figure 5: Crossover Gadget for SMW.

Crossover gadget Given trampolines and spiky, rotating stone balls, a non-exhaustive crossover gadget can be created. In Figure 5, the crossover gadget for SMW is shown. All stone balls in the crossing section are rotating clockwise with the same speed. There are four locations on which those balls are placed. Each ball has a twin that rotates with an angle offset of 180 degrees. This

rotating behaviour leads to short phases in which Mario can quickly pass through without getting hit by a stone ball. Leakage is not possible, since if one path is free, the other is blocked. Waiting in the crossing region for the other path to get traversable will not work as there always is a ball that will pass Mario's location before he gets a chance to leak to the other path. When traversing the vertical path bottom to top, Mario uses a vine to quickly climb to the top. When traversing top to bottom, an alternative version of this gadget is used, in which Mario simply falls through the crossing region in a well-timed moment. Although the version with the vine could be used to climb or fall down as well, this alternative version is used for the sake of variety.

Finally, we observe that generalised SMW is a member of PSPACE. To describe a state of the game, the following information is needed: Mario's location, the locations of the trampolines, the states of the rotating blocks (traversable or not) and whether Mario is carrying a trampoline. This describes everything that can change while playing and is a relevant part of the game. This clearly is within polynomial bounds. The goal is to move Mario to a defined finish location in order to win the game. The algorithm non-deterministically guesses an assignment which allows the player to reach this finish location. By Savitch's theorem NPSPACE = PSPACE, so the claim follows and we immediately arrive at the PSPACE completeness of (generalised) SMW. We emphasise that the only generalisation performed here is the map size of SMW. By the above observation, SMW is in PSPACE. Furthermore, applicability of the framework yields PSPACE hardness. In conjunction SMW is PSPACE complete.

4 Implementation

In this section, we briefly comment on the implementation of SFW. The code is written in Java, compatible with versions 1.6 and later. Thus the game is platform independent. It comprises about 11400 LOC in 115 files and is freely available online: https://github.com/DwarfVader/mario.

The game provides a help window, explaining the usage/controls of the program. The user can choose between English and German language. The program allows to enter propositional formulas or QBF formulas in prenex form, allowing for standard notions for propositional connectives. Alternatively (multiple) DIMACS *.cnf* files can be read in. If multiple files are read in, each DIMACS file corresponds to a different level. Based on the considered formula, the game creates a world following either the framework for NP or PSPACE hardness. Both frameworks provide individual incarnations of their corresponding gadgets. The start, finish and crossover gadgets are not shared between the frameworks. Thus, the generated game map precisely represents a SAT or QBF solver, respectively. If we rate the gameplay experience of these variants it is an unfortunate result of the repetitive nature of the latter framework that Mario seems most of the time to be busy with travelling between the different gadgets, rather than actual problem solving. This is in contrast to the subgame representing SAT solving, implementing a variant of the original Super Mario Bros. Here the game experience is quite satisfactory. In order to partly overcome this, the game provides speed-up facilities to increase Mario's speed. Additionally, we provided a feature that allows Mario to directly teleport from the beginning to the end of the literal and check paths, avoiding those long travels.

5 Conclusion and Future Work

In this paper we have refined a framework to show PSPACE hardness for video games established by Aloupis et al. [1]. By precisely defining the framework and fixing a topology of it, we have established an instantiation of it for SMW, which directly allows an implementation in an actual game. The purpose of this game, dubbed SFW, is to exhibit the computational complexity of video games like SMW explicitly. Playing SFW is equivalent to the search for a satisfying assignment for a quantified Boolean formula φ, that is, φ is satisfiable if and only if the player is able to reach the finish location. Thus the player actually acts as a QBF solver. Furthermore, we have also refined and implemented in SFW the corresponding framework for NP. That is, if the game is started with a propositional formula then the player acts as a SAT solver. We are employing the game in teaching and in public events directed towards prospective students. So far the experiences have been promising.

The repetitive nature of the PSPACE frameworks somewhat hinders the gameplay experience. Currently we overcome this effect by extending Mario's capabilities. In future work one could study whether one can overcome this defect of the game more principally. Here we could either come up with an implementation based on a direct PSPACE hardness proof specialised for SMW, for example exploiting [4] or preferably develop an alternative to the framework which does not rely on the current repetitive structure (present in [1, 9]). On a more theoretical level it may be of interest to relate known benchmarks for QBF solvers to research on the difficulty in logic puzzles (like Sudoku) or video games in general [2, 8].

Acknowledgements We want to thank *The Spriters Resource* for providing the images that are used in this paper and the implementation.

References

[1] G. Aloupis, E.D. Demaine, A. Guo, and G. Viglietta. Classic Nintendo Games are (Computationally) Hard. *TCS*, 586:135–160, 2015.

[2] M-V. Aponte, G. Levieux, and S. Natkin. Measuring the level of difficulty in single player video games. *Entertainment Computing*, 2(4):205–213, 2011.

[3] J.C. Culberson. Sokoban is PSPACE-complete, 1997. Available online https://webdocs.cs.ualberta.ca/~joe/Preprints/Sokoban/.

[4] E.D. Demaine, G. Viglietta, and Aaron Williams. Super Mario Bros. is Harder/Easier Than We Thought. In *Proc. 8th FUN*, volume 49 of *LIPIcs*, pages 13:1–13:14, 2016.

[5] H. Samulowitz and F. Bacchus. *Using SAT in QBF*, pages 578–592. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[6] R. Kaye. Minesweeper is NP-complete. *Math. Intelligencer*, 22(2):9–15, 2000.

[7] J. Lindsberger. Classic Nintendo Games are Completely Hard. Master's thesis, Universitt Innsbruck, 2016. Available online at https://github.com/DwarfVader/mario.

[8] R. Pelánek. Difficulty Rating of Sudoku Puzzles by a Computational Model. In *Proc. 24th FLAIRS*. AAAI Press, 2011.

[9] G. Viglietta. Gaming Is a Hard Job, but Someone Has to Do It! *TCS*, 54(4):595–621, 2014.

[10] G. Viglietta. Lemmings is PSPACE-complete. *TCS*, 586:120–134, 2015.

[11] T. Walsh. Candy Crush is NP-hard. *CoRR*, abs/1403.1911, 2014.

[12] T. Walsh. Candy Crush's Puzzling Mathematics. *American Scientist*, 102(6):40–43, 2014.