

Super Mario Bros. Is Harder/Easier than We Thought

Erik D. Demaine¹, Giovanni Viglietta², and Aaron Williams³

- 1 Computer Science and Artificial Intelligence Laboratory, MIT, 32 Vassar St., Cambridge, MA 02139, USA
edemaine@mit.edu
- 2 School of Electrical Engineering and Computer Science, University of Ottawa, Canada
viglietta@gmail.com
- 3 Division of Science, Mathematics, and Computing, Bard College at Simon's Rock, 84 Alford Rd, Great Barrington, MA 01230, USA
haron@uvic.ca

Abstract

Mario is back! In this sequel, we prove that solving a generalized level of Super Mario Bros. is PSPACE-complete, strengthening the previous NP-hardness result (FUN 2014). Both our PSPACE-hardness and the previous NP-hardness use levels of arbitrary dimensions and require either arbitrarily large screens or a game engine that remembers the state of off-screen sprites. We also analyze the complexity of the less general case where the screen size is constant, the number of on-screen sprites is constant, and the game engine forgets the state of everything substantially off-screen, as in most, if not all, Super Mario Bros. video games. In this case we prove that the game is solvable in polynomial time, assuming levels are explicitly encoded; on the other hand, if levels can be represented using run-length encoding, then the problem is weakly NP-hard (even if levels have only constant height, as in the video games). All of our hardness proofs are also resilient to known glitches in Super Mario Bros., unlike the previous NP-hardness proof.

1998 ACM Subject Classification F.1.3 Complexity Measures and Classes

Keywords and phrases video games, computational complexity, PSPACE

Digital Object Identifier 10.4230/LIPIcs.FUN.2016.13

1 Introduction

Super Mario Bros.¹ is one of the most famous and iconic video games, launching the 1985 revolution in home console gaming known as the Nintendo Entertainment System (NES/Famicon), and pioneering the genre of side-scrolling platform video games. Super Mario Bros. also launched a series of over 20 games, in total selling over 260 million copies, though the original game remains the best selling at over 40 million copies [11] (and for many years, was even the best-selling video game of all time).

Super Mario Bros. is also well known for being a challenging game, starting the adjective “Nintendo Hard” [6]. To formalize this concept mathematically, we study the computational complexity of a generalized form of Super Mario Bros. Previous work presented at FUN

¹ Super Mario Bros. is a trademark of Nintendo. Sprites are used and stripped and modified ROMs are presented here under Fair Use for the educational purpose of illustrating mathematical theorems.



2014 [1] proved that Super Mario Bros. 1–3, The Lost Levels, and Super Mario World are all NP-hard, but left open whether they were in NP. Here we show that membership in NP is unlikely: Super Mario Bros. is PSPACE-complete (Section 4). This proof uses a monster’s location to store state and implement the doors in the PSPACE-completeness metatheorem of [1, 9, 10].

Both our PSPACE-hardness proof and the original NP-hardness proof [1] generalize Super Mario Bros. to have levels of arbitrary size, but also to have the entire level fit “on screen”, or equivalently, to have a game engine that remembers the state of off-screen sprites (i.e., *persistent* monsters and items). While the generalization in level size is necessary to study computational complexity, in the real games the screen is much smaller than the level, and the game engine forgets the state of everything substantially off-screen.² To model this more “realistic” setup, we consider Super Mario Bros. generalized to have arbitrary level size but only a constant screen size as well as allowing only a constant number of on-screen sprites.³ (Our “realistic” model also excludes fantastical gameplay elements like horses, boats, and extrasolar objects that were present in Vargomax’s (humorous) investigation of Super Mario Bros. and graph coloring [8].)

Within this bounded-screen-size model, we consider two possibilities for how levels can be specified. First, if every tile is encoded explicitly, then we show that Super Mario Bros. becomes solvable in polynomial time (Section 2). Second, if tiles can be encoded implicitly using run-length encoding (where a row of k identical tiles gets encoded as one copy of the tile with a binary encoding of k), then we show that Super Mario Bros. becomes weakly NP-hard (Section 3).⁴ This hardness proof also works for levels of constant height, which is a property shared by most Super Mario Bros. games; for example, even the very flexible Super Mario Maker allows levels of height only double that of the screen, or 27 blocks, the same as Super Mario Bros. 3, and about double that of Super Mario Bros. It also relies on many mechanics of Super Mario Bros. previously not exploited: pipes, time limits, multiple lives, 100 coins grant a free life, and levels have checkpoints. Furthermore, the underlying decision problem is not whether Mario can complete a single level (referred to as “reaching the flagpole”), but it is whether Mario can complete a sequence of levels (referred to as “rescuing the princess”).

Both of our hardness proofs are resilient to known *glitches* [2, 5], where the implementation of Super Mario Bros. is counter to the intuitive Mario physics with which most players are familiar. Figure 1 shows examples of such glitches; see Section 2 for details. By contrast, the previous NP-hardness proof [1] breaks when such glitch behaviors are permitted.

1.1 Playable gadgets

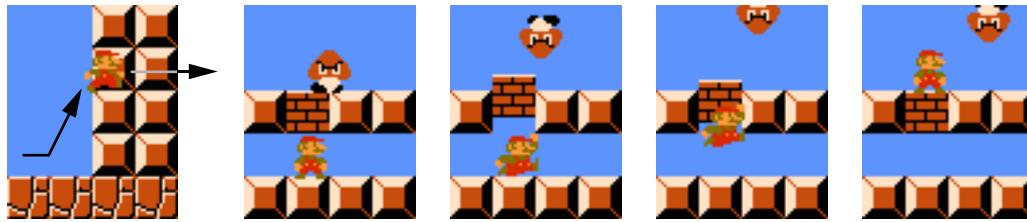
Our reductions have been implemented and tested in the original NES Super Mario Bros. game. Our modified ROMs can be downloaded from <http://giovanniviglietta.com/files/SMB/gadgets.zip>. We also implemented the two gadgets of Section 4 in Super Mario Maker, although with some minor modifications to cope with the slightly different physics.

² The source code [3] defines two screen sizes—the *visible screen* and the somewhat larger *relevant screen*—and all sprites outside of the relevant screen are forgotten.

³ The Nintendo Entertainment System could draw only eight sprites per scanline (row of the screen) without flickering.

⁴ The source code [3] implements such a run-length encoding for runs of blocks or coins of length up to 16, so we consider the natural generalization to runs of blocks or coins of arbitrary length with an efficient encoding.

The level IDs are 92F8-0000-005D-F452 (crossover gadget) and A8B5-0000-005D-E79A (open-close door gadget).



■ **Figure 1** Two glitches: going through a wall and jumping through a brick with a monster on top

2 Standard SMB is polynomial-time solvable

In this section we consider a generalization of the standard Super Mario Bros. (from now on, *SMB*) game that is still quite close to the original, and we prove that the solvability of its levels, and even sequences of levels, can be decided in polynomial time.

2.1 Basic gameplay of SMB-standard

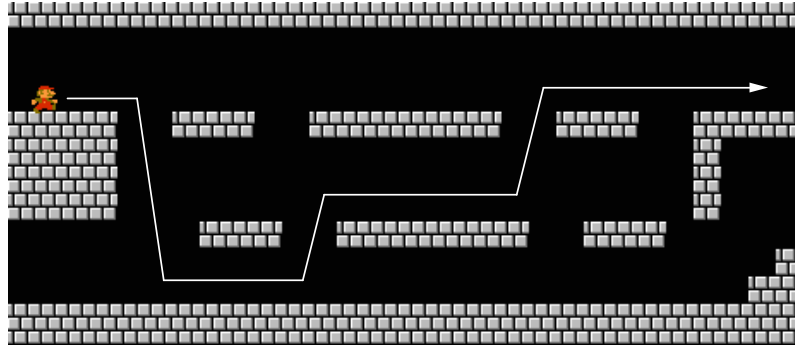
In each level, Mario starts from a specific location, and his goal is to reach a *flagpole* (or an *axe*, in boss levels), which marks the end of the level and leads to the next one, or to the endgame sequence. If Mario dies in a level, he loses one life (of the initial three) and has to play the same level again from the beginning. Nonetheless, each level has a *checkpoint* which, when reached, becomes the new starting location. Occasionally there are hidden *warp pipes* that let Mario skip some levels.

Next we describe *SMB-standard*, which is our first generalization of SMB. We allow the designer to construct arbitrarily large 2-dimensional levels with arbitrarily many objects in them. Accordingly, we allow the initial timer and the amount of lives of Mario to be exponentially large integers (with respect to the size of the level). However, we keep the physics and local effects unaltered, and we keep the screen's size constant, allowing only a constant number of objects on screen, as in the original SMB (where the screen's size is 16×16 tiles, and there can be at most six sprites on screen, including Mario). The screen follows Mario's movements, and all the objects and enemies that exit the screen are forgotten by the game. This means that there is rectangular region in which the action takes place and objects are animated. When this region moves and new elements enter the active area, they are loaded from a read-only level file. On the other hand, the elements that exit the screen are no longer active. As a consequence, if Mario kills an enemy, goes to some other area and then comes back, the enemy is reloaded from the level file in its original position and state, as if it was never killed. In SMB the screen can only scroll to the right, but nothing changes if we allow it to move in all directions in SMB-standard.

There is some pseudo-randomness in SMB: a 7-byte memory area is used as a pseudo-random register; it is deterministically updated at every frame (starting from a fixed seed), and looked up whenever a random number is needed. The illusion of randomness comes from the fact that the frame-by-frame sequence of button presses of a human player is hardly ever the same. In other words, the player's input is the only real source of randomness in the game.

The types of enemies and game elements of SMB are too many to be listed here. A complete definition of all the game mechanics can be found in the game’s commented source code [3]. However, for the purpose of this paper, it is sufficient to highlight just a few aspects of the game. Some of them will be described in this section; others will be introduced when they are needed.

An interesting game element is the *Loop Command* object, which is an invisible tile that is used in some castle levels to make Mario go back a few screens if he does not follow the right path, as Figure 2 exemplifies. When a Loop Command object enters the screen, Mario’s position in the screen is checked: if he is in the wrong place, the screen’s coordinates change (typically, the x coordinate is decreased by a constant amount). Sometimes, Loop Command objects are *cumulative*: in World 7-4, Mario can be sent back only every third Loop Command object that he encounters, and only if he failed the test on at least one of the previous three Loop Command objects. In SMB-standard, we can generalize this “three” to any (exponentially large) number.



■ **Figure 2** World 7-4: if Mario does not guess the correct path, he is sent back a few screens

2.2 Glitches

There are also several *glitches* in SMB, some of which are documented in [2, 5]. None of the known glitches affects the results of this section, so we may as well include them in SMB-standard. However, a couple of them will be relevant in the next sections, as they make the game behave counter-intuitively in certain situations (see Figure 1). Namely, when a vertical wall is at least two tiles high, Mario can slightly penetrate between two tiles by jumping towards them at the right speed. Then, if he suddenly steers in the opposite direction, the wall pulls him inside instead of pushing him out. This makes Mario potentially able to walk through every wall consisting of more than a single tile. Also, if Mario is small (i.e., he has not eaten a Super Mushroom) and he hits a brick block with his head twice fast enough, he can jump through the block, provided that there is a monster on top of it.

2.3 Reaching the flagpole in SMB-standard

We start by proving that solving a single level of SMB-standard is a polynomial-time task. For this, we only need a very small set of assumptions, which the interested reader can verify by inspecting [3].

► **Theorem 1.** *Reaching the flagpole in a level of SMB-standard without losing lives is a polynomial-time task.*

Proof. Mario’s position in the screen is encoded as a pair of integer coordinates (with a precision of $1/16$ of a pixel). Hence the possible positions of Mario within the screen are finitely many. Mario’s velocity vector is encoded similarly, and his speed never exceeds a constant value. The same holds for all monsters and moving objects. All the state transitions of each object, as well as all possible interactions between objects, can be computed in constant time. Hence, the transition from one frame to the next can be computed in constant time, because the screen has constant size and there can be only a constant number of interacting objects in it.

It follows that there is only a polynomial number of possible *configurations*, where a configuration represents the state of the game at a given frame, and is characterized by a screen position within the level, at most a constant number of objects on screen (each with a constant-size state), a constant-size pseudo-randomly-generated number, a polynomial-size Loop Command object counter, a Loop Command “failure flag”, and the player’s input for that frame (which is a combination of button states). Moreover, all the possible one-frame transitions between configurations can be determined in polynomial time. This means we can efficiently construct a polynomial-size *configuration graph*, with several starting vertices s_1, \dots, s_k , each corresponding to an initial pseudo-random register, and several finish vertices $t_1, \dots, t_{k'}$, each corresponding to a configuration in which Mario touches the flagpole.

Now, because each directed edge in this graph corresponds to a feasible move (assuming the player hits the right buttons), and such a move makes the timer decrease by a constant fraction $1/f$ of a time unit, solving the level amounts to finding a shortest path from each s_i to any t_j , which is done via a breadth-first traversal of the configuration graph. If any of the shortest paths is longer than f times the available number of time units, then the level is unsolvable. ◀

2.4 Rescuing the princess in SMB-standard

In order to extend the previous proof to sequences of levels, we have to recall how lives work in SMB. Mario gets an extra life whenever he collects a 1-Up Mushroom, or repeatedly jumps on enemies without touching the ground, or collects (a multiple of) 100 coins. He loses a life whenever he fails to reach the flagpole, either because he is killed or because he runs out of time. When he loses the last life, the game is over. Note that Mario does not lose his coins when he dies or completes a level.

The classic SMB game has a fixed limit on the maximum number of lives that Mario can have. In SMB-standard we allow Mario’s number of lives to grow unboundedly, which makes our next theorem a little stronger.

► **Theorem 2.** *Rescuing the princess in SMB-standard is a polynomial-time task.*

Proof. We will reason as in Theorem 1, with the additional difficulty that now we have to keep track of the number of lives gained and lost.

Let the game have n levels. Each level has a checkpoint, which splits it into two *chunks*. When Mario reaches the same horizontal coordinate of a checkpoint (regardless of his vertical coordinate), the checkpoint is considered reached. Then, the next time he dies in the level, he will restart from the checkpoint, at the smallest possible vertical coordinate. So, the first chunk of each level can only be played from one initial location. The second chunk of a level can be played from at most v possible initial locations, where v is the (polynomial) number of possible vertical coordinates. Indeed, the location with smallest vertical coordinate is the one that occurs after Mario dies in the second chunk of the level, and the other initial locations may occur the first time that Mario enters the second chunk from the first chunk.

Recall that the pseudo-random register is deterministically updated at every frame, and let k be the constant number of possible values that the pseudo-random register can assume. The crucial observation is that, if the game is beatable with an infinite initial supply of lives, then it is beatable with an initial supply of $2kvn$ lives. Indeed, there are $2n$ level chunks in total, each of which can be replayed several times before moving on to the next one. In turn, there are at most v possible initial positions for Mario in each chunk, and at most k possible initial values of the pseudo-random register. A chunk may be impossible to beat from some initial vertical positions, or for some initial values of the register. Moreover, even if a chunk is always beatable, it may be necessary to finish it when the pseudo-random register has a specific value, because that initial value may be required to beat the next chunk, etc. However, if a specific final value is attainable at all, it must be attainable at the cost of at most k lives.

Because 100 coins grant an extra life and it is impossible to lose coins, we can equivalently think of a coin as $1/100$ of a life. For each of the $2n$ chunks, we build a configuration graph G as in Theorem 1, also adding to the state the initial vertical coordinate of Mario, and the (fractional) number of lives ℓ that Mario has. By the above argument, if at any point $\ell \geq 2kvn$, we can safely assume ℓ to be infinite. Hence we only need encode $200 \cdot kvn + 1$ possible values for ℓ , which can be done using only $O(\log n)$ bits.

Now we connect together the graphs corresponding to different chunks, in such a way that each final vertex of a chunk leads to the initial vertex of the next chunk having a matching pseudo-random register in its state. We also add similar edges corresponding to warp pipes. Furthermore, we add edges corresponding to Mario's deaths, which either lead to the beginning of the current chunk, with one less life, or to a special Game Over vertex having no outgoing edges. We add an extra “zeroth” chunk corresponding to the title screen, with k vertices, one for each possible initial value of the pseudo-random register. Finally, we put a last vertex t , which is reached after beating the last chunk.

The game starts from the vertex s in the zeroth chunk having the appropriate pseudo-random seed, and then proceeds by following the edges in this “macro-graph” corresponding to the player's inputs (or lack thereof) at each frame. For instance, waiting in the zeroth chunk without pressing buttons only makes the pseudo-random register change, while pressing the Start button leads to the appropriate initial vertex of the first chunk. Because the macro-graph has polynomial size, verifying if the vertex t can be reached from s is a polynomial-time task. ◀

3 SMB with run-length encoding is weakly NP-hard

Levels in the original SMB game are encoded using a limited form of compression. For example, a run of up to 16 consecutive coins can be encoded using the same space as a single coin. In this section we consider the hardness of a SMB variation with levels that allow arbitrary *run-length encoding* (RLE), meaning that m consecutive blocks of a fixed type along an individual row can be encoded using $O(\log m)$ bits. We extend this concept to entire levels, as well: we can encode m consecutive copies of the same level by attaching an $O(\log m)$ -bit number to the encoding of the level. We refer to this variation as *SMB-RLE*, and we prove that rescuing the princess in this game is weakly NP-hard.

Our reduction is from the Knapsack problem. The main idea is to model item weights by time, and item values by coins. We create a level in which Mario is faced with n independent choices of either collecting v_i coins at a cost of w_i time, or skipping past the coins by using a pipe. Each of these choices corresponds to either adding item i to the knapsack or not,

respectively. We set a time limit of W to the level to restrict Mario's choices, and we force Mario to collect at least V coins—and thus $\lfloor V/100 \rfloor$ extra lives—by separating Mario from the princess by a sequence of levels, each of which will cause him to lose one life.

Besides the addition of RLE, our new variation of SMB is similar to the SMB-standard variation introduced in Section 2. In particular, we do not require the persistence or non-persistence of any game elements. However, to make our result stronger, we use levels of constant height that cannot scroll to the left (such as the ones in the original SMB).

The remainder of this section formalizes our Knapsack problem, clarifies our assumptions about the various game elements, and then describes individual levels and overall reduction. Ultimately, we briefly discuss the variation of SMB in which run-length encoding can be applied to blocks but not to whole levels. This version is also weakly NP-hard, provided that the number of coins that grant an extra life (which normally is 100) can be configured arbitrarily.

3.1 Knapsack problem

The following version of the Knapsack problem is also known as the *0-1 Knapsack problem* because each item is indivisible and is either inserted into the knapsack or not.

Knapsack problem

Input: Item weights w_1, w_2, \dots, w_n and values v_1, v_2, \dots, v_n , a total weight capacity of the knapsack W and a target value V .

Output: Is there a subset of items with total weight $\sum w_i \leq W$ and value $\sum v_i \geq V$?

The problem is NP-complete even if each item's weight is proportional to its value [4], and there is also a well-known dynamic-programming algorithm for solving it. The algorithm runs in *pseudo-polynomial time*, meaning that its running time is polynomial with respect to the magnitude of the numbers involved (as opposed to the number of bits used to represent them). Thus, the problem is weakly NP-complete. We will find it convenient to reduce from a constrained version of the problem in which each item weight is at least 100 times as large as its value.

► **Lemma 3** (Knapsack with Large Weights). *The Knapsack problem is weakly NP-complete when restricted to inputs with $w_i \geq 100 \cdot v_i$ for all $1 \leq i \leq n$.*



Proof. The decision problem is unchanged by multiplying the item weights and total capacity by a fixed positive integer. If the integer is an item value, then the maximum magnitude of the numbers in the resulting problem is at most squared. Therefore, the result follows from multiplying every item weight by $100 \cdot v'$, where v' is the maximum value in $\{v_1, \dots, v_n\}$. ◀

3.2 Game elements and controls

The only game elements we use are solid blocks, pipes, coins, checkpoints, and the timer. We clarify our assumptions about these elements and other aspects about the levels below.

- A level can have arbitrary width, but we only use levels that have a constant height. This differs from the NP-hardness result in [1], which relied upon having levels of arbitrary height and width. In fact, we only use levels that are two blocks high.
- A level consists of a single room. In other words, our hardness result does not require the use of any side rooms or bonus areas.

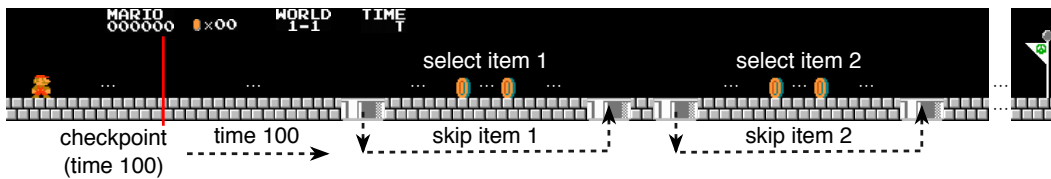
- A level can have an arbitrary number of pipes. For a Knapsack problem with n items we create a level with n pairs of pipes. The pipes do not nest, in the sense that the x coordinates of the entrances and exits satisfy $in_1 < out_1 < in_2 < out_2 < \dots < in_n < out_n$.
- Going through a pipe takes constant time, regardless of the distance traveled.
- The screen never scrolls left. This fact, combined with our use of pipes, implies that no game element can be revisited. Thus, persistence is a non-issue, and our reduction will work regardless of whether or not it is assumed.
- The timer for each level can be arbitrarily large. However, we assume that it is always a multiple of 100 to keep it as close to the original game as possible.
- There is one checkpoint per level. If Mario dies after the checkpoint, but before the flagpole, then he is respawned at the checkpoint instead of the beginning of the level. As in the original game, the timer after respawning can differ from the original timer.

Mario will never have reason to jump, move left, move up, or stop running in the levels we create. Furthermore, our designs have natural “walking analogues” that are functionally equivalent if Mario is forbidden from running. Thus, our problem will remain weakly NP-hard if Mario’s controls are limited to any subset that includes  and .

3.3 Choice level

A *choice level* is parameterized by $2n + 1$ integers: $w_1, \dots, w_n, v_1, \dots, v_n$, and W . The level is comprised of $n + 3$ sections (see Figure 3) in the following order.

- The *checkpoint section* begins with a blank screen followed by a checkpoint that will reset the timer to 100 upon Mario’s death.
- The *empty section* requires 100 time units to traverse and is otherwise empty.
- There are n *decision sections*, each of which begins with one pipe and ends with one pipe. Taking the first pipe will transport Mario to the second pipe, thereby skipping over the entire section. Otherwise, Mario travels from the first pipe to the second pipe on foot and can collect a series of coins. The number of coins along the i th decision section is v_i and these coins are surrounded by at least one empty screen to the left and right. The path from the first pipe to the second pipe requires w_i units of time to traverse.
- The final section is a *flagpole section* and is the end of the level.

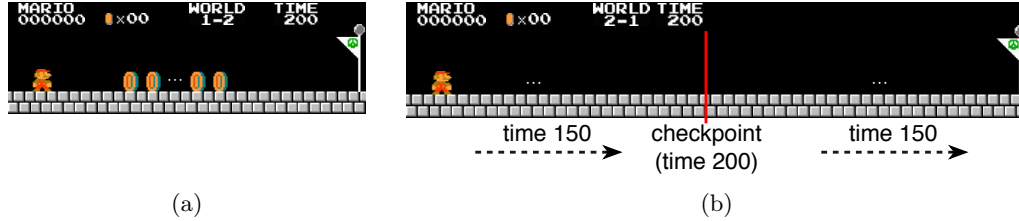


■ **Figure 3** Sketch of a choice level including the first two decision sections

Notice that the first two sections force Mario to complete the level in one life. More specifically, if he respawns at the checkpoint, then he cannot pass the empty section due to the timer. Also notice that the i th decision section is only well defined if t_i is large enough to accommodate c_i coins and the empty screens to the left and right. This will not be an issue with our reduction because we use the constraint discussed in Lemma 3. The level’s timer is $T = W + \varepsilon$, where ε is the time required to complete the level using all n pipe pairs.

3.4 Coin level

A c -coin level contains c coins for some $0 \leq c \leq 99$ (see Figure 4 (a)). Its time limit of 200 is sufficient for completing the level either by running or walking for all c .



■ **Figure 4** (a) A coin level, and (b) a kill level

3.5 Kill level

A *kill level* takes 300 time units to complete but only provides a timer of 200 (see Figure 4 (b)). It also has a checkpoint halfway through the level that provides 200 time units after respawning. Mario can complete the level by expending one life to reach the checkpoint, and then by continuing to the flagpole after respawning.

3.6 Weak NP-hardness of SMB-RLE

Now we prove the main result of this section.

► **Theorem 4.** *Rescuing the princess in SMB-RLE is weakly NP-hard, even when using levels of height 2, any subset of buttons including \oplus and \ominus , and the elements in Section 3.2.*

Proof. Consider an instance of the Knapsack problem satisfying Lemma 3. Let $\lceil V/100 \rceil = \ell$. We create an SMB-RLE game consisting of the following levels:

- World 1-1 is a choice level using the integers $w_1, \dots, w_n, v_1, \dots, v_n$, and W from the Knapsack problem instance.
- World 1-2 is a c -coin level, where $c = 100 \cdot \ell - V$.
- World 2- x is a kill level for all $1 \leq x \leq \ell + 2$.

These levels are well defined by Lemma 3 and the discussion in Section 3.1. For the $\ell + 2$ copies of the kill level in World 2 we make use of run-length encoding, as they could be exponentially many. Notice that Mario can rescue the princess if and only if he has at least $\ell + 3$ lives after completing World 1-2. Considering that Mario starts the game with three lives, this is equivalent to collecting $V + c$ coins on Worlds 1-1 and 1-2, which in turn is equivalent to collecting V coins on World 1-1. Mario can collect V coins on World 1-1 if and only if there is a solution to the Knapsack problem. ◀

3.7 Weak NP-hardness without using run-length encoding for levels

In this section, we show that weak NP-hardness holds even when run-length encoding is allowed only for the blocks within a single level, and not for encoding an entire sequence of levels. (The proof above uses run-length encoding on the level sequence to (just) create the kill levels that force Mario to lose a certain number of lives.) To achieve this goal, we add a different assumption: the amount of coins that Mario has to collect to get an extra life is no

13:10 Super Mario Bros. Is Harder/Easier than We Thought

longer 100, but it can be any number (decided by the game designer). Then we can modify our previous NP-hardness reduction as follows:

- V coins grant an extra life.
- World 1-1 is a choice level using the integers $w_1, \dots, w_n, v_1, \dots, v_n$, and W from the Knapsack problem instance.
- Worlds 2-1, 2-2, and 2-3 are kill levels.

Because there are only three kill levels, they can be represented explicitly without using run-length encoding. In order to pass them, Mario needs to have at least four lives by the time he reaches World 2-1. In turn, this is possible if and only if he can collect at least V coins in World 1-1, considering that he starts the game with three lives, and V coins grant an extra life. Hence he can finish all the levels if and only if the Knapsack problem is solvable.

As a consequence, Theorem 4 also extends to this more natural variation of the game.

4 SMB with no reset of off-screen objects is PSPACE-complete

In this section, we assume that the screen does not move in a fixed direction, and that objects off-screen are not reset. In particular, we assume that the engine remembers the positions of all monsters. Reasoning as in Section 2, it is not hard to prove that this variation of SMB, which we call *SMB-general*, is solvable in PSPACE. Indeed, each game configuration can be stored in polynomial space, and the configuration graph can be efficiently navigated. This shows that SMB-general is in NPSPACE, and thus in PSPACE by Savitch's Theorem. In the rest of this section, we prove that SMB-general is PSPACE-complete.

4.1 PSPACE-hardness framework

To prove that SMB-general is PSPACE-hard, we use the “open-close door” framework from [1], similar to metatheorems of [9, 10]. The framework is based on a reduction from Quantified Boolean Formula involving the following elements:

- a player-controlled avatar,
- a starting location and an exit location,
- arbitrarily oriented paths arranged in a plane,
- crossover gadgets, and
- open-close door gadgets.

A *crossover gadget* is a region in which two paths A and B cross each other without leakage. That is, if the avatar is walking along A and traverses the crossover gadget, it cannot end up on B, and vice versa.

An *open-close door gadget* is an object with two states—open and closed—and is intersected by three disjoint paths:

- a *traverse* path, which can be traversed by the avatar if and only if the door is in the open state;
- a *close* path which, when traversed by the avatar, causes the door to close; and
- an *open* path which, when traversed by the avatar, allows the player to open the door, but may not force them to. (The open path may also be a dead end.)

If all the above elements can be implemented in a game, then it is PSPACE-hard [9, 10, 1].

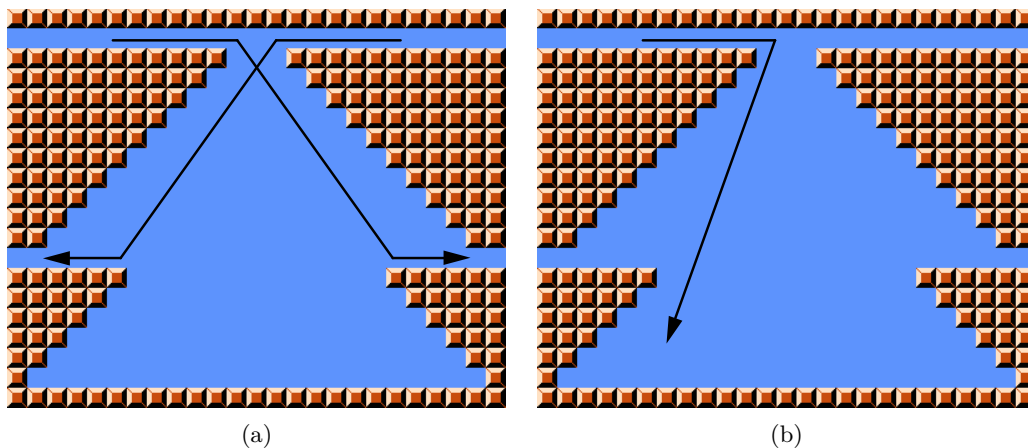
关键就是实现
这里的 open-close
door 部件

4.2 Starting/exit locations and paths

Of course SMB already has the first three elements, while arbitrarily oriented paths can be implemented by using solid blocks. We will put no Super Mushrooms in our levels, so we may assume that Mario will always be small, and therefore it is sufficient to implement 1-tile-high paths going horizontally and diagonally. It is not hard to construct them without creating walls that are higher than one tile, thus avoiding the walk-through-walls glitch.

4.3 Crossover gadget

Crossover gadgets are easy to implement with pipes. However, if we insist on not using pipes, we can achieve the same result by using only Mario's physics, thanks to the gadget in Figure 5. A bonus feature of this gadget is that it can be easily adapted to work in almost any platform game. As the first figure shows, Mario can go from the top-left path to the bottom-right path by simply running at full speed. He cannot jump across the 3-tile-wide gap, due to the low ceiling. Moreover, if he tries to go from the top-left path to the bottom-left path, he has to switch direction in mid-air, which makes him lose all his momentum, and makes him fall into the pit, as shown in the second figure. Once in the pit, he can never get out, because it is too deep. Similarly, if he tries to jump from the bottom-right to the bottom-left path, he ends up falling into the pit, because the 13-tile gap between the two paths is too wide. Of course, the gadget works symmetrically in the right-to-left direction.



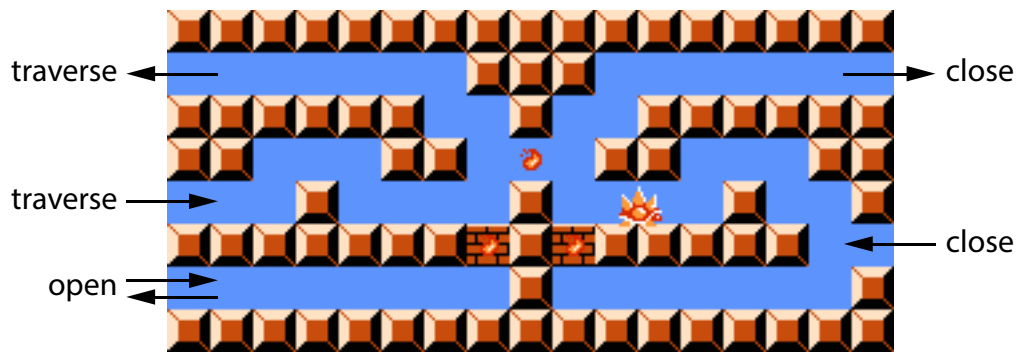
■ **Figure 5** PSPACE-hardness reduction: crossover gadget. (a) Mario can avoid the pit by simply running forward at full speed. (b) If Mario tries to change path, the lack of momentum makes him fall into the pit.

4.4 Open-close door gadget

Our implementation of the open-close door gadget is illustrated in Figure 6. The three flames represent tiles where rotating firebars are attached (for clarity, only the first flame of each firebar is drawn). Note that firebars do not necessarily have to be attached to solid blocks in SMB (an example is the underwater part of World 8-4).

The monster on the right is a Spiny, which keeps walking back and forth in the 4-tile-wide tunnel. When the Spiny is in this area, the door is considered open. Indeed, in this case Mario can freely walk through the traverse path. However, Mario cannot walk through the close path, because the Spiny is in the way. This is a tough monster, which can be killed

图例



■ **Figure 6** PSPACE-hardness reduction: open-close door gadget (in the open state)

only by the fireballs that Mario shoots after picking up a Fire Flower, which is not available in our level. In particular, Small Mario dies if he jumps on a Spiny. If, however, Mario takes the tunnel that runs below the Spiny, he can hit the brick block from below, as in Figure 7. If this is done with the correct timing, it makes the Spiny bounce through the central firebar, and to the other side of the gadget. Note that the brick block does not break, because Mario is small. The purpose of the firebar attached to the brick block is to kill Mario if he triggers the jump-through-brick glitch. Similarly, the central firebar prevents Mario from taking the shortcut between the traverse and the close paths, but it does not harm or obstruct the Spiny.

When the Spiny is on the left-hand side of the gadget, Mario can safely go through the close path (see Figure 7). Then the Spiny starts walking back and forth in the 4-tile-wide tunnel on the left, and the door is considered closed. Indeed, by a symmetric argument, Mario is now unable to go through the traverse path because the Spiny is in the way, and he has to reach the open path underneath if he wants to send the Spiny back to the right-hand side, thus opening the door again, and resetting the gadget to its original state.

4.5 PSPACE-hardness of SMB-general

To finalize our construction, we make sure to give Mario enough time to complete the level (provided that it is feasible), by setting the timer to some linear function of the configuration space's size.

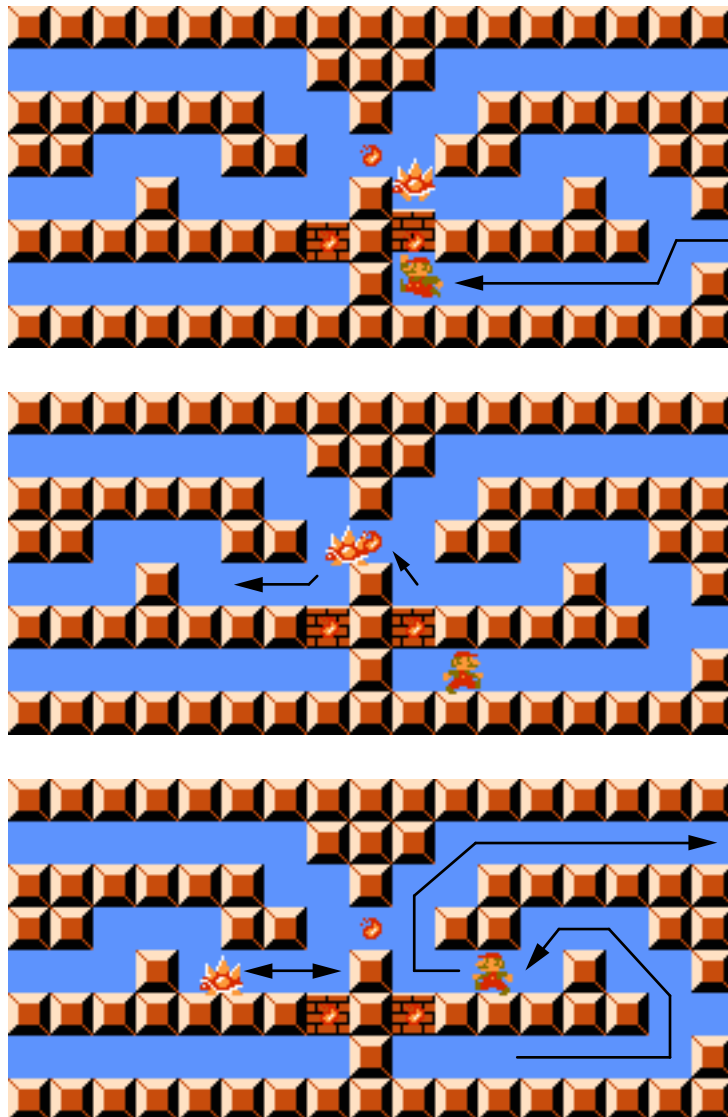
Note that the walk-through-walls glitch cannot be exploited in our crossover and open-close door gadgets, because none of them contains 2-tile-high vertical walls. Therefore, as explained in Section 4.1, we have our main result.

► **Theorem 5.** *Reaching the flagpole in SMB-general is PSPACE-complete.* ◀

4.6 Proper generation of Spinies

A possible point of criticism on our open-close door gadget is that, although the SMB engine allows the designer to place Spinies anywhere in the levels, the official SMB levels never make use of such a feature. Spinies are never found at specific locations, but they are dynamically thrown at Mario by floating *Lakitus*, as Figure 8 exemplifies.

It is not too hard to incorporate this feature into our levels, if we align all the door gadgets in the upper part of the construction. As the level starts, we can force Mario to walk above each door gadget, where a Lakitu throws a Spiny at him, which eventually falls into a pit that leads to the door gadget below. We can time everything in such a way that the



■ **Figure 7** If the Spiny is in Mario's way, it has to be moved to the other side of the gadget. This can be done by hitting the brick block from below.

Lakitu can throw only one Spiny per door gadget, if Mario runs at full speed. We can also prevent Mario from entering door gadgets through these pits, by placing firebars in them.

5 Open Problems

There are several natural open problems that arise naturally from our results. We name a few.

Is reaching the flagpole of run-length-encoded constant-height Super Mario Bros. weakly NP-hard? Is it also NP-complete? Our reduction in Section 3 critically relies on having multiple levels, so we wonder about the complexity of a single level. We note that the strategy of Section 3 could be altered to use *multiple checkpoints* in a single level instead of multiple levels, but we view this as “cheating” because the original Super Mario Bros. uses at most



■ **Figure 8** Lakitu throwing Spiny Eggs, which hatch as they hit the ground, becoming Spinies

one checkpoint per level (while Super Mario Maker allows two per level).

To prove Theorem 4, we assumed that run-length encoding could be used to represent sequences of levels, as well as sequences of tiles. Then, in Section 3.7 we were able to remove this assumption, by introducing the ability to configure the number of coins that have to be collected to gain an extra life. Does Theorem 4 hold if run-length encoding for sequences of levels is not allowed (but it is for blocks), and exactly 100 coins grant an extra life?

Our PSPACE-hardness reduction of Section 4 produces levels of unbounded size in both dimensions. Note that the same result can be obtained for levels of constant height, provided that pipes are used. Is Super Mario Bros. PSPACE-complete also for levels of constant height that do not contain pipes? There is hope for proving this given that constant-width Nondeterministic Constraint Logic is PSPACE-complete [7].

Finally, we suspect that our proofs can be adapted to the many Super Mario Bros. sequels, but this remains to be explored.

References

- 1 Greg Aloupis, Erik D. Demaine, Alan Guo, and Giovanni Viglietta. Classic Nintendo games are (computationally) hard. *Theoretical Computer Science*, 586:135–160, 2015.
- 2 Speed Demos Archive. Super Mario Bros. http://kb.speeddemosarchive.com/Super_Mario_Bros.
- 3 dopleganger. A comprehensive Super Mario Bros. disassembly. <http://giovanniviglietta.com/files/SMB/source.asm>.
- 4 Michael Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.
- 5 TASVideos. Super Mario Bros. <http://tasvideos.org/GameResources/NES/SuperMarioBros.html>.
- 6 TV Tropes. Nintendo Hard. <http://tvtropes.org/pmwiki/pmwiki.php/Main/NintendoHard>.
- 7 Tom C. van der Zanden. Parameterized complexity of graph constraint logic. In *Proceedings of the 10th International Symposium on Parameterized and Exact Computation (IPEC 2015)*, pages 282–293, 2015. doi:10.4230/LIPIcs.IPEC.2015.282.
- 8 Vargomax V. Vargomax. Generalized Super Mario Bros. is NP-complete. In *Proceedings of the 6th Biannual Workshop about Symposium on Robot Dance Party of Conference in Celebration of Harry Q. Bovik's 0x40th Birthday (SIGBOVIK 2007)*, pages 87–88, 2007.
- 9 Giovanni Viglietta. Gaming is a hard job, but someone has to do it! *Theory of Computing Systems*, 54(4):595–621, 2014.

- 10 Giovanni Viglietta. Lemmings is PSPACE-complete. *Theoretical Computer Science*, 586:120–134, 2015.
- 11 Video Game Sales Wiki. Mario. <http://vgsales.wikia.com/wiki/Mario>.