Tracks from hell – when finding a proof may be easier than checking it

Matteo Almanza
Dipartimento di Informatica, Sapienza Università di Roma, Italy.
almanza.1597415@studenti.uniroma1.it

Stefano Leucci
Institute of Theoretical Computer Science, ETH Zürich, Switzerland.
stefano.leucci@inf.ethz.ch
https://orcid.org/0000-0002-8848-7006

Alessandro Panconesi
Dipartimento di Informatica, Sapienza Università di Roma, Italy.
ale@di.uniroma1.it

Abstract

We consider the popular smartphone game Trainyard: a puzzle game that requires the player to lay down tracks in order to route colored trains from departure stations to suitable arrival stations. While it is already known [Almanza et al., FUN 2016] that the problem of finding a solution to a given Trainyard instance (i.e., game level) is NP-hard, determining the computational complexity of *checking* whether a candidate solution (i.e., a track layout) solves the level was left as an open problem. In this paper we prove that this verification problem is PSPACE-complete, thus implying that Trainyard players might not only have a hard time finding solutions to a given level, but they might even be unable to efficiently recognize them.

2012 ACM Subject Classification Theory of computation → Problems, reductions and completeness

Keywords and phrases puzzle games, solitaire games, Trainyard, verification

Digital Object Identifier 10.4230/LIPIcs.FUN.2018.4

1 Introduction

The relationship between the Catholic Church and science and technology in the course of history has been a long and complex one. On the one hand, there are dark and immensely sad episodes like the condemnation of Galileo Galilei. On the other, one must acknowledge the fantastic contribution to science and mathematics throughout the centuries at the hands of Catholic clergymen, an all star team that includes the likes of Gregor Mendel, Francis Bacon, Nicolaus Copernicus, and Bernard Bolzano, to name just a few.

Perhaps nothing better than the history of the railways illustrates the ambivalent relationship of the Catholic Church toward science and technology. Early adoption within the Papal States of this revolutionary means of transportation was stymied by Pope Gregory XVI (1765-1846) who famously warned *"Chemins de fer, chemin d'enfer"* ("road of iron, road of hell") [4, p. 164]. His successor Pious IX however, realized the potential of railway transportation for the purposes of the Holy See. The roads of iron could not only lead to hell, but also to holy places like the sanctuary of Lourdes. The steam engine became a facilitator of mass pilgrimages. The all powerful Roman Curia finally gave in on October 2, 1934, when the stately Vatican City Central Station opened [14, p. 653]. This imposing building has been to this day the headquarters of the smallest railway system in the world—300 meters of

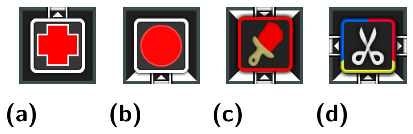

Figure 1 Different types of tiles: (a) a red departure station, (b) a red arrival station, (c) a red painter, (d) a splitter.

tracks in total!—an apparent contradiction that epitomizes an ambivalent attitude toward a profound dilemma.

One must wonder why such a potentially useful and, from the point of view of Catholic orthodoxy, apparently innocuous technology was met with such a high degree of suspicion. The answer, it turns out, is hidden in the odd meanderings of the computational complexity of games. The puzzle game Trainyard beautifully captures the inherent tension between two moral imperatives, finding the right path and making sure that the path taken is indeed the virtuous one.

In a landmark paper, Almanza et al. proved that finding the layout of a railway network in Trainyard—a computational task easily seen to be in PSPACE—is NP-hard [2]. They left open the question of verification, namely the computational complexity of checking whether a given track layout delivers all trains safely to destination. In this paper we show that, surprisingly, this task is PSPACE-complete. Interestingly, it is still possible that Trainyard lies in NP. If this were the case then checking certificates for Trainyard would be more difficult than finding them, unless NP = PSPACE. In moral terms, walking along the path of virtue could prove impossibly arduous, even when you have found it. This is a bizarre state of affairs that vindicates the cautious approach of the Roman Curia toward the railway question.

2 Problem Definition

In a Trainyard level we are given a rectangular board in which each of the cells is either empty or occupied by a *tile*. The two main tile types are departure and arrival stations: a departure station (shown in Figure 1 (a)) hosts a single train, initially colored either blue or red, while an arrival station (shown in Figure 1 (b)) accepts one train of a given color. The player's task is to route trains from departure stations to arrival stations by placing (possibly rotated versions of) the *rail pieces* of Figure 2 into empty cells. The rail pieces of Figure 2 (a)–(d) are traversed by trains in the straightforward way, while the pieces of Figure 2 (e) and (f) are called *switches* and route trains going in the bottom-to-top direction towards two neighboring tiles in an alternating fashion, i.e., each transiting train flips the *state* of the switch. Trains traveling in the opposite directions are routed to the tile immediately below the *switch* but they still affect the switch's state.

To further complicate[1] things, the grid also contains other special tiles that interact with incoming trains in various ways (see Figure 1 (c) and (d)), namely:

Painters: trains traversing this tile will acquire the color of the painter tile itself. A painter can be either red or blue. Note that a painter gadget has only two entry points located on opposite sides of the cell.

[1] Actually, the original game is even more complex that the one described in the present work. The subset of rules and tiles described here, however, suffices for the purposes of our reduction.

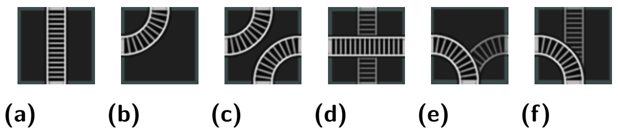

Figure 2 Different kinds of rail pieces. Each of the pieces can be rotated by 90, 180 or 270 degrees.

Splitters: 1 splitters act both on the number of trains and on their color. A splitter has only a single one-way entry gate, and two one-way exit gates located at the left and right side of the cell. When a train enters a splitter, it vanishes but two new trains are created and exit from the sides. If the incoming train was red or blue, then the outgoing trains will also have that same color. If the incoming train was purple,[2] then the new train exiting from the left side will be blue, while the other outgoing trail will be red.

After all the rail pieces have been placed by the player, the design is put to the test: trains exit from departure stations simultaneously and they travel on rails at a speed of one tile per second. When a train moves from a tile to the next, the directions of the involved rails must match. If this does not happen then the train will *crash* and the player loses. Moreover, whenever two trains simultaneously occupy the same rail piece while traveling in the same direction, they merge into a single train. The color of this resulting train depends on the color of the two merged trains. We are only interested in the following cases: if the merged trains had the same color, then the new train will also retain that color; if one merged train was blue and the other was red, then the resulting train will be purple. Two trains going in different directions can also occupy the same tile at the same time: in this case no merge occurs but the color of both trains is still changed according to the previous rules.

The player wins iff this process eventually reaches a state in which there are no traveling trains, each arrival station has received exactly one train of the associated color, and no crash has occurred.

We study the computational complexity of the TRAINYARD-VERIFICATION problem, i.e., the problem of deciding whether a solution (i.e., a placement of the rail pieces) to a Trainyard level results in a win for the player (see Figure 3). In some sense we investigate how hard it is for the player to recognize a correct solution to a Trainyard level. Unfortunately for the player, this tasks can not be performed efficiently, unless PSPACE = P. Indeed, we are able to show the following:

▶ **Theorem 1.** TRAINYARD-VERIFICATION *is* PSPACE-*complete.*

3 Other related works

Trainyard belongs to the broad class of casual games: these games are characterized by an intuitive gameplay which is usually organized into a series of small puzzles of increasing difficulty. Interestingly enough, many casual games are *hard* to solve, not only for human players, but also for machines: it is often the case that the computational problem of finding a solution to a given level (instance) of a casual game turns out to be at least NP-hard. Notable examples include e.g., Candy Crush [8], 2048 [12], Flow-Free [1], Sokoban [3], Rush

[2] Purple trains can appear when a red and a blue train meet, as we explain in the following.

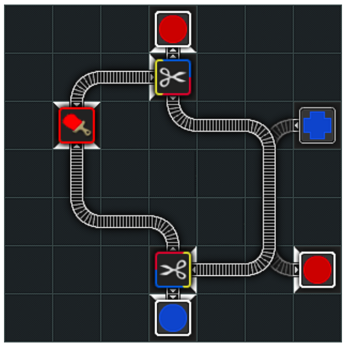

Figure 3 A correct solution to a Trainyard level, i.e., a yes-instance of TRAINYARD-VERIFICATION.

Hour [6], Two-Dots [13], or Peg-Solitaire [15, 9] and it might even be the case that the success of these games is due, in part, to their challenging levels, as suggested in [5]. For a discussion of other NP-hard and PSPACE-hard puzzles and of general tecniques for showing their hardness, we refer the interested reader to [11] and [10].

4 Our Reduction

4.1 Preliminaries

Our reduction is from the *Iterated Monotone Boolean Circuit* problem (IT-MON-BC for short) [7]. A monotone boolean circuit C is a directed acyclic graph whose non-source vertices are labeled with either $\wedge$ ("*and*") or $\vee$ ("*or*"). The source vertices are called *input-vertices*, while all the other vertices are called *gates*. Gates that are also sinks in G are called *output-vertices*. Let $v_1^I, \dots, v_n^I$ (resp. $v_1^O, \dots, v_m^O$) be the input-vertices (resp. output-vertices) of C, in some order. Once a boolean value is associated to each input vertex, it is possible to *evaluate* the circuit by computing a boolean value for each output-vertex. This is done by propagating the input values from the sources towards the sinks, i.e., by considering the gates of C in preorder and assigning a truth value to them depending on their label and on the truth value of their in-neighbors. If $\overline{x} = \langle x_1, \dots, x_n \rangle$ is the boolean vector containing the input values (where x_i is the initial truth value for vertex v_i^I), then the evaluation of C computes a vector $\overline{y} = f_C(\overline{x})$, where f_C is the function from $\{\texttt{True}, \texttt{False}\}^n$ to $\{\texttt{True}, \texttt{False}\}^m$ *implemented* by the circuit and the i-th entry y_i of $\overline{y}$ is the truth value corresponding to vertex v_i^O. Without loss of generality we assume that all the gates of C have in-degree 2. If $n = m$ then we can also compute

$$f_C^t(\overline{x}) = (\underbrace{f_C \circ \cdots \circ f_C}_{t \text{ times}})(\overline{x})$$

by evaluating C t times, where the input of i-th evaluation, for $i > 1$, consists of output of the $i - 1$-th evaluation.

The IT-MON-BC problem asks, given a monotone boolean circuit with $n = m$, an input vector x, and an index h, to determine whether there exists a positive integer t such that $f_C^t(\overline{x})_h = \texttt{True}$, i.e., if iterating C with a $\overline{x}$ as the initial input eventually causes the h-th output to become $\texttt{True}$.

▶ **Theorem 2** (Lemma 3 in [7]). IT-MON-BC *is* PSPACE-*complete even when restricted to circuits of in-degree* 2 *and out-degree* 2.

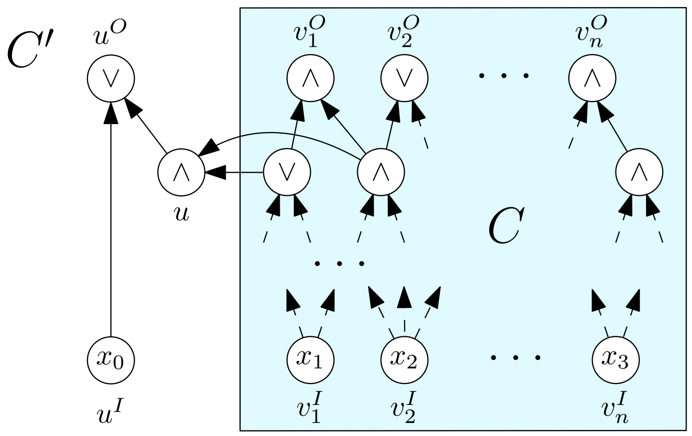

Figure 4 Transformation of the circuit of an IT-MON-BC instance $\langle C, \overline{x}\rangle$ into an equivalent circuit C' having one additional input/output vertex pair (namely, u^I and u^O). Initially, the truth value of the new variable x_0 associated with u^I is `False`. The output value associated to u^O becomes `True` as soon as the value of v_1^O is `True` and remains `True` in all subsequent iterations.

Notice that, since there can be at most 2^n distinct input vectors, if $(f_C^t(\overline{x}))_h =$ `False` for each $t = 1, \dots, 2^n$ then, by the pigeonhole principle, we can conclude that the answer to an instance of IT-MON-BC is false. For technical convenience we assume w.l.o.g. that $h = 1$ and that, once the first output becomes `True`, it will remain `True` in all the subsequent iterations of the circuit. Notice that this latter assumption is not restrictive and can be removed by transforming the circuit C into an equivalent circuit C' having the desired property as follows (see also Figure 4): initially C' is a copy of C, then we add to C' (i) a new input-vertex u^I, (ii) a new gate u having the same label as v_1^O in C, and (iii) a new output-vertex u^O having label $\vee$; then, for each each edge (v, v_1^O) in C, we add a corresponding edge (v, u) to C', and finally we add the two edges (u^I, u_O) and (u, u_O) to C'. It is easy to check that, for every input vector $\overline{x}$ and any $x_0 \in \{$`True`, `False`$\}$, the truth value of the vertices $v_1^O, \dots, v_n^O$ of C with input $\overline{x}$ coincides with the truth value of the corresponding vertices of C' with input $\langle x_0, x_1, \dots, x_n\rangle$ where x_0 is the value initially assigned to u^I. Moreover, as soon as v_1^O become `True`, u_O will also become `True` and will retain the `True` value in all subsequent iterations. Hence, we have obtained a circuit C' having the desired property, modulo a renaming of its input/output vertices.

4.2 Overview

In the following we show how to convert, in polynomial time, an instance $\langle C, x, h\rangle$ of IT-MON-BC into a instance of TRAINYARD-VERIFICATION that is a valid solution iff $(f_C^{2^n}(\overline{x}))_h =$ `True`, thus establishing the PSPACE-hardness of TRAINYARD-VERIFICATION. Notice that TRAINYARD-VERIFICATION clearly belongs to PSPACE as simulating a solution only requires polynomial space. Moreover, since the number of possible states that can occur during the simulation is at most exponential in the instance's size, there exists an upper limit T_0 to the number of simulation steps needed: if the Trainyard level is still not solved after T_0 steps, then the simulation must be stuck in some cyclic sequence of states, hence the TRAINYARD-VERIFICATION instance has a "no" answer.

A high-level picture of our reduction is shown in Figure 5. The initial input $\overline{x}$ of the circuit C is encoded in the color of n departing trains in the "input area", if $x_i =$ `True` (e.g., x_2 in Figure 5) then the i-th departure station is red, while if $x_i =$ `False` (e.g., x_1 and x_3 in Figure 5) the corresponding departure station is blue. The departing trains (moving in the

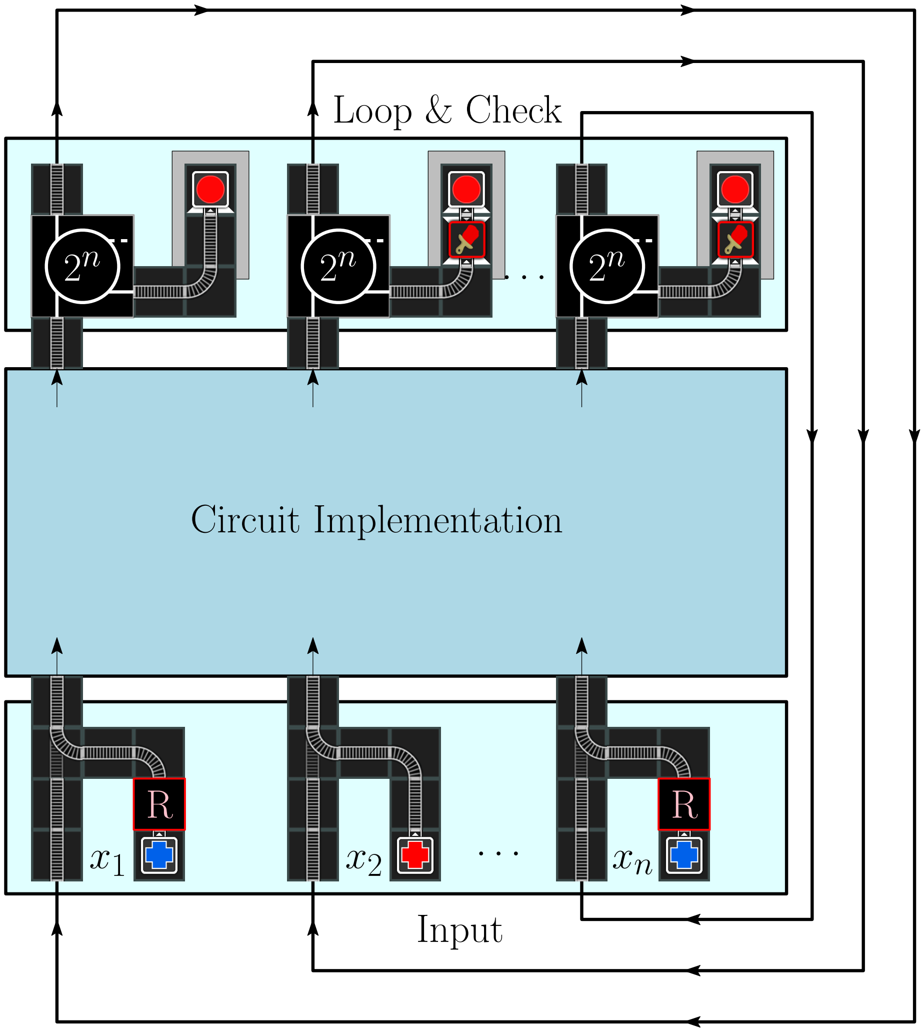

Figure 5 Overview of our reduction from IT-MON-BC to TRAINYARD-VERIFICATION. The input trains departing from the stations in the "Input" region traverse the "Circuit Implementation" area exactly 2^n times (using the outer loops to return to the bottom) and are then routed to the gray *check* gadgets in the "Loop and Check" area.

bottom-to-top direction) then enter the "Circuit Implementation" area where they traverse a series of gadgets that simulate the gates of the circuit C. Eventually, exactly n trains exit from the "Circuit Implementation" area, the i-th of which encodes (in its color) the truth value of the i-th output of the circuit. These trains are now routed to the one final region of our Trainyard level, namely the "Loop and Check" area. This region contains n *loop* gadgets that act *transparently* for the first $2^n - 1$ times they are traversed, i.e., they let any train entering from the bottom exit unaltered from the top. These outgoing trains are then re-routed back into the inputs of the circuit and the whole process repeats. At the 2^n-th iteration, the loop gadgets stop acting transparently and instead divert the incoming trains to the rails exiting from their right. Here each train enters a *check* gadget (shown in gray in Figure 5) containing one arrival station. Collectively, these check gadgets allow the level to be completed if and only if the train corresponding to the first output of the circuit (which represents the first entry of $f_C^{2^n}$) encodes the value `True`.

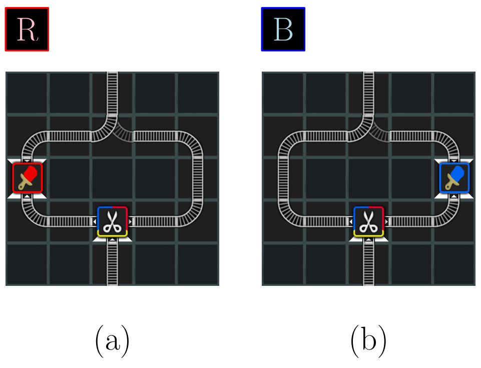

Figure 6 (a) Implementation of Red-Shift gadget. (b) Implementation of Blue-Shift gadget. These gadgets convert the color scheme of a train from CS2 to CS1 and vice-versa, respectively.

An interactive demonstration of our reduction is available at: `https://trainyard.isnphard.com/verification/`

4.3 Gadgets and Color Schemes

We encode the gates of C using a combination of *gadgets*, i.e., combinations of rails and special tiles, and the links of C using rails between the two corresponding gadgets. Gadgets will receive some trains as inputs (usually from the bottom), will perform some operations, and will let the result trains exit (usually from the top). We assume that all the input trains enter a gadget simultaneously: since a train always takes the same number of steps to traverse a gadget[3], this property can be guaranteed by choosing suitable lengths for the rails connecting two consecutive gadgets.

The evaluation of C will be simulated by trains that will carry truth values from one gadget to the other. Those truth values will be encoded using train colors according to two different color schemes.

CS1: A red train represents the value `True`, a purple train represents the value `False`. Blue trains are not allowed.

CS2: A purple train represents the value `True`, a blue train represents the value `False`. Red trains are not allowed.

4.3.1 Converting between color schemes

Here we show how to change the color of a train that is carrying a truth value encoded using CS1 to the correct encoding according to CS2, and vice-versa.

The conversion from CS2 to CS1 is performed by the *Red-Shift* gadget shown in Figure 6 (a). The input train enters the gadget from the rail on the bottom and is split into two trains exiting from the sides of the splitter tile. If the incoming train is blue then these two trains will be blue as well, the one on the left side will pass through a red painter tile and will merge back with the blue train coming from the right side into a purple train. If incoming train is purple then the train exiting the spliter tile from the left side will be blue while the one exiting from the right will be red. Due to the red painter gadget, the left train will also become red, and the two trains will merge thus causing a single red train to exit from the top of the gadget. Notice also that if the input of a Red-Shift gadget is a red train, then the train will retain its red color. This property will be useful in the sequel.

[3] with the exception of loop gadgets, as will be discussed in the following.

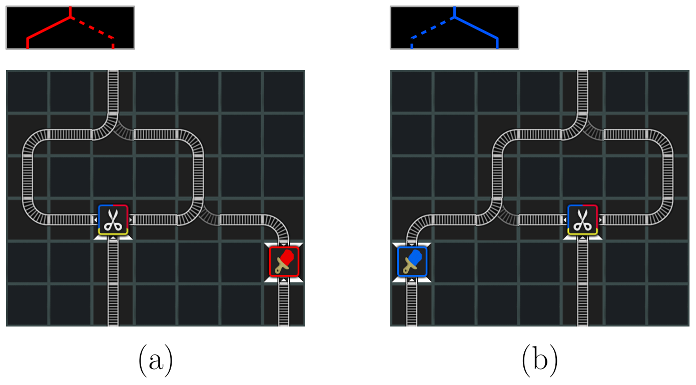

Figure 7 Implementation of red (a) and blue (b) absorption gadgets.

The conversion from CS2 to CS1 is done by the *Blue-Shift* gadget shown in Figure 6 (b), whose operation is symmetric to the Red-Shift gadget. Notice again that blue trains retain their colors when traversing a *Red-Shift* gadget.

These two gadgets allow us to change the color of each train entering a gadget in order to meet the expected color scheme. We can hence assume that all the appropriate color conversions happen on the railways connecting the output of a gadget to the input of next one. This will be particularly useful in implementing gadgets for “and” and “or” gates which require the two input trains to be colored according to different colors schemes. As an example notice how the blue trains that will exit the departure stations of Figure 5 (i.e., the stations corresponding to `False` inputs) immediately traverse a Red-Shift gadget. This ensures that all the trains leaving the “Input” area will be colored according to CS1.

4.3.2 Train Absorption Gadget

Some of the gadgets will generate additional trains as a side effect of their operation. In order to get rid of these spurious trains we would like them to merge with a train that we call a *main* train. This operation should be performed with care to ensure that the color of the main train is preserved. This can be done by using the *absorption gadgets* shown in Figure 7.

Let us focus on the *red absorption gadget* of Figure 7 (a) which shows how a main train colored according to CS1 and entering the gadget from the leftmost rail on the bottom can be merged with any other train entering from the rightmost rail. Once the main train reaches the splitter, the train exiting from the right side will always be red (recall that, according to CS1, the main train must be either red or purple). We can therefore safely merge it with the spurious train, which has been colored red by the red painter tile, to obtain a single red train. This red train is then merged back with the train exiting the left side of the splitter, resulting in a single train of the same color of the incoming main train.

The *blue absorption gadget* gadget of Figure 7 (b) merges a main train colored according to CS2 with any other train (entering from the leftmost rail on the bottom) and its operation is analogous the the red absorption gadget just described.

4.3.3 Crossover Gadget

The *crossover* gadget allows two trains entering (at the same time) from the bottom rails to exit (at the same time) from the top two rails in the opposite order. Its implementation is straightforward and it is shown in Figure 8 (a). The turns in the gadget ensure that the two input trains do not meet when the rails cross (as this might alter their original colors).

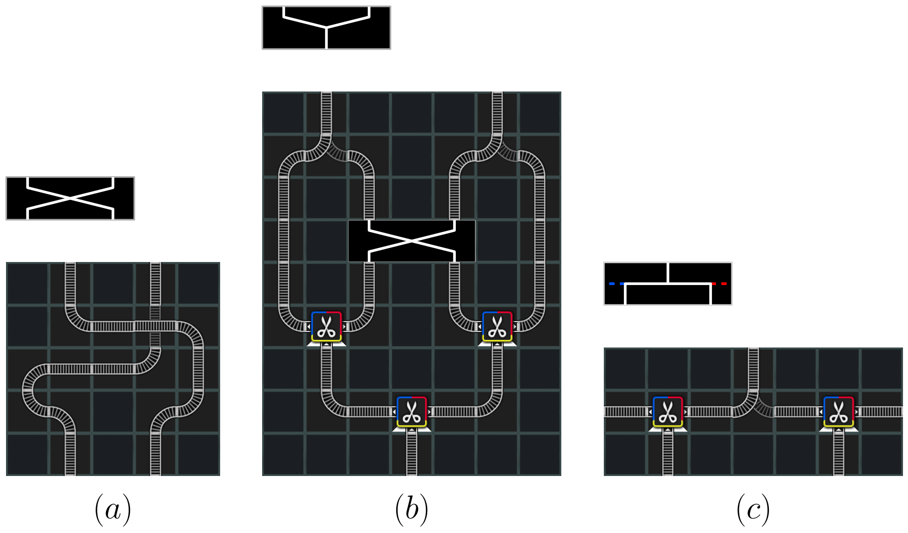

Figure 8 Implementation of the Crossover (a), Fanout (b), and Blend (c) gadgets.

4.3.4 Fanout Gadget

The *fanout* gadget duplicates an input train entering from the bottom rail into two identical trains exiting from the top two rails. This gadget is necessary as, in general, input-vertices and gates might have two or more out-neighbors. By chaining together several fanout gadgets it is possible to create exactly one train for each out-neighbor of a vertex, each encoding the same truth value. The implementation of this gadget is shown in Figure 8 (b).

4.3.5 Blend Gadget

We still have to show how “and” and “or” gates of the circuit C are implemented. To this aim it is useful to first describe an intermediate gadget that we call a *blend gadget* (see Figure 8 (c)). This gadget takes two input trains entering from the bottom side: the leftmost train is colored according to CS2 while the rightmost train is colored according to CS1. There are four possible combinations of (valid) inputs, corresponding to the four values in $\{\texttt{True}, \texttt{False}\}^2$. In all of these cases the gadget will output three trains. Two of them, namely the ones exiting from the left and right side of the gadget, are spurious trains and will be dealt with by the “and” and “or” gadgets. The other train is the actual output of the gadget and its color depends on the truth values of the input trains, as detailed in the following table:

Left Value	**Right Value**	**Left Color (CS2)**	**Right Color (CS1)**	**Output Color**
`True`	`True`	purple	red	red
`True`	`False`	purple	purple	purple
`False`	`True`	blue	red	purple
`False`	`False`	blue	purple	blue

In other words, the output train is blue if both the inputs are false, red if they are both true, and purple if exactly one of the inputs is true. Notice that, so far, this coloring scheme does not adhere neither to CS1 nor to CS2.

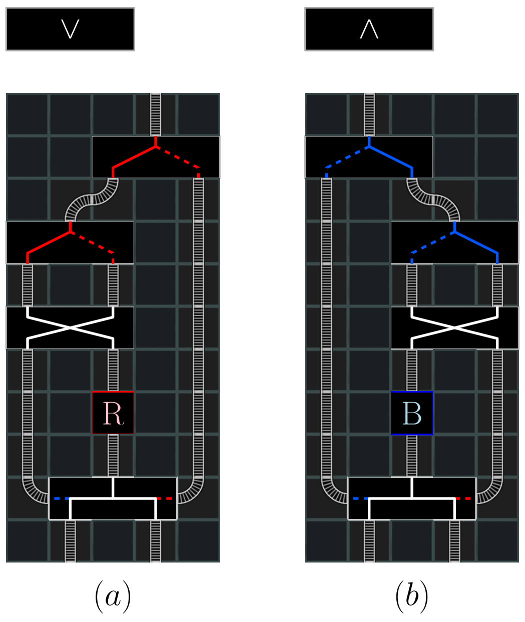

Figure 9 Implementation of the “or” (a) and “and” (b) gadgets.

4.3.6 Or Gadget

The *or* gadget is shown in Figure 9 (a) and implements an “or” gate of the circuit C. It is obtained by appending a Red-Shift gadget to the output of a Blend gadget. It is easy to check that the color of the output train is red if at least one of the two inputs of the blend gadget encodes the values `True`, and purple otherwise. In other words, the gadget computes the logical or between the two signals carried by the input trains and encodes the result according to CS1. To take care of the two extra trains exiting the Blend gadget we use two Absorption gadgets to merge them into the output train.

4.3.7 And Gadget

The implementation of the *and* gadget is similar to the one of the *or* gadget: by appending a Blue-Shift gadget to the output of a Blend gadget we have that the color output train is blue if at least one of the two input trains carries the logical value `False`, and purple otherwise. Thus, the and gadget computer the logical and of the two input signals and encodes the output according to CS2. As before, we dispose of the the two spurious trains exiting from the Blend gadget by using two Absorpion gadgets, as shown in Figure 9 (b).

4.3.8 Loop Gadget

The purpose of this gadget is to ensure that the circuit is evaluated (i.e., traversed by the trains) exactly 2^n times, where the trains exiting from the output-vertices of one iteration constitute the input of the next evaluation. After 2^n evaluations this feedback loop will break and the output trains will be allowed to reach the check area.

The gadget implementation is recursive and it is shown in Figure 10 (a)–(d). Figure 10 (a) shows the details of a 2-loop gadget: the first train to enter the gadget from the single rail on the bottom will exit from the top and will cause the two switches to flip, we call the first encountered switch (i.e., the one closer to the bottom) a *counting switch*. When a second

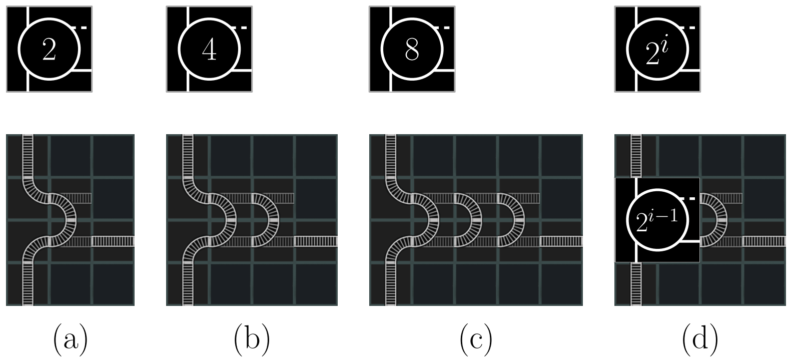

Figure 10 Implementation of a 2^i loop-gadget for $i = 1$ (a), $i = 2$ (b), $i = 3$ (c), and in the general case (d).

train enters the gadget, it will now exit from the rail on the right side due to the position of the counting switch (that will now revert to the initial state). Notice that the position of the other (non-counting) switch plays no role in the gadget operation. Indeed, its sole purpose is to provide an extra input that allows any train entering *from* the right side (i.e., moving in the right-to-left direction) to exit from the top side of the gadget without affecting the state of the counting switch. This allows a 2-loop to be used as a sub-gadget in the 4-loop construction of Figure 10 (b): here the first train to exit from the right side of a 2-loop traverses once again a counting and a non-counting switch (in order) and is routed back to the top exit of the 2-loop; the second train to exit the 2-loop (which corresponds to the 4-th train entering from the bottom rail) will revert the counting-switch to its original state and it will proceed to the right. By repeating this construction one can easily obtain the 8-loop of figure Figure 10 (c) and, more generally, the 2^i-loop of Figure 10 (d). The first $2^i - 1$ trains entering a 2^i-loop will exit from the top, while the 2^i-th train will exit from the right. Notice how after the j trains have traversed a 2^i-loop, the state of the counting switches encodes the number $i \bmod 2^i$ in binary, where the least-significant-bit corresponds to the innermost counting switch.

We lay the tracks so that all the n trains entering the "Loop and Check" area (see Figure 5) reach the 2^n-loop gadgets at the same time (this can be guaranteed by tuning the length of the railways connecting to the inputs of the loop gadgets). This is because, in contrast to the other gadgets, the time required for a train to traverse a 2^n-loop gadgets depends on the specific iteration (i.e., on the state of the counting switches). By requiring this additional property, we can ensure that, on every iteration, all the 2^n-loop gadgets will always change their state simultaneously and hence the n incoming trains will also exit the gadgets simultaneously. This allows us to also synchronize the trains once they are routed back to the "Input" area.

4.3.9 Check Gadgets

The purpose of the check gadgets (highlighted in gray in the "loop and check" area of Figure 5) is to ensure that the solution to the Trainyard level will be valid if and only if the circuit C of the IT-MON-BC instance is such that the first entry of $\overline{y} = f_C^{2^n}(\overline{x})$ is `True`. When the loop gadgets in the "loop and check" area are traversed 2^n times, the trains encoding the n output values of $f_C^{2^n}(\overline{x})$ will exit from the right side of their respective loop gadget. Since we are not interested in the values $y_2, \dots, y_n$, each of the corresponding trains is colored

red (thus discarding its currently encoded truth value) and fed into an arrival station that expects exactly one red train. As for the output train corresponding to y_1, we assume w.l.o.g. that it is encoded according to CS1 (if this is not the case, then it suffices to use a Red-Shift gadget just after the output of the corresponding 2^n-loop gadget). Then, the train will be red if y_1 is `True` and purple otherwise. Hence, it suffices to route this train into an arrival station expecting one red train. If $y_1 =$ `True`, all the arrival stations will be satisfied and the Trainyard level is won, otherwise the purple train training to enter a red station will crash and the level will be lost.

5 Conclusions

We have proved that the problem of checking whether a candidate solution to a Trainyard instance actually solves the level is PSPACE-complete. What is the exact complexity of *finding* such a solution, however, is still an open question: we know from [2] that this problem is NP-hard and it is easy to check that it also belongs to PSPACE (since it is possible to enumerate all the possible track layouts). If Trainyard belongs to NP, this would mean that computing a solution could be easier than checking it, unless NP = PSPACE. It might be the case that a more involved certificate than the natural one (i.e., the placement of the tracks in a solution) is needed, or that every Trainyard level that admits a solution also allows for a *simple* solution, i.e., a solution that can be recognized in polynomial time.

Other examples of games for which checking the natural solution is at least NP-hard[4] include *Settlers of Catan* and *Carcassonne*[5], as the scoring rules involve longest-path computations.

References

1 Aaron B. Adcock, Erik D. Demaine, Martin L. Demaine, Michael P. O'Brien, Felix Reidl, Fernando Sánchez Villaamil, and Blair D. Sullivan. Zig-zag numberlink is np-complete. *JIP*, 23(3):239–245, 2015. `doi:10.2197/ipsjjip.23.239`.

2 Matteo Almanza, Stefano Leucci, and Alessandro Panconesi. Trainyard is NP-hard. *Theoretical Computer Science*, 2017. `doi:10.1016/j.tcs.2017.09.039`.

3 Joseph Culberson. Sokoban is PSPACE-complete. In *Proceedings of the 1st International Conference on Fun with Algorithms (FUN'98)*, volume 4, pages 65–76, 1998.

4 Giuseppe Pasolini dall'Onda and Pier Desiderio Pasolini. *Memoir of Count Giuseppe Pasolini.* Longmans, Green, and Company, 1885.

5 David Eppstein. Computational complexity of games and puzzles. `https://www.ics.uci.edu/~eppstein/cgt/hard.html`.

6 Gary William Flake and Eric B. Baum. Rush hour is pspace-complete, or "why you should generously tip parking lot attendants". *Theor. Comput. Sci.*, 270(1-2):895–911, 2002. `doi:10.1016/S0304-3975(01)00173-6`.

7 Eric Goles, Pedro Montealegre, Ville Salo, and Ilkka Törmä. Pspace-completeness of majority automata networks. *Theor. Comput. Sci.*, 609:118–128, 2016. `doi:10.1016/j.tcs.2015.09.014`.

[4] More precisely, we are given the final state of the game and we want to check whether a player claiming to be the winner is actually the winner.

[5] With a suitable set of expansions: *Carcassonne: Abbey & Mayor* allows to have biforcations in roads (i.e., vertices of degree 3 in the road graph) and *Carcassonne: King and Scout* awards bonus points to the player that completes the longest road.

8 Luciano Gualà, 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, CIG 2014, Dortmund, Germany, August 26-29, 2014*, pages 1–8. IEEE, 2014. doi:10.1109/CIG.2014.6932866.

9 Luciano Gualà, Stefano Leucci, Emanuele Natale, and Roberto Tauraso. Large peg-army maneuvers. In Erik D. Demaine and Fabrizio Grandoni, editors, *8th International Conference on Fun with Algorithms, FUN 2016, June 8-10, 2016, La Maddalena, Italy*, volume 49 of *LIPIcs*, pages 18:1–18:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/LIPIcs.FUN.2016.18.

10 Robert A. Hearn and Erik D. Demaine. *Games, Puzzles, and Computation.* CRC Press, 2009.

11 Graham Kendall, Andrew J. Parkes, and Kristian Spoerer. A survey of np-complete puzzles. *ICGA Journal*, 31(1):13–34, 2008.

12 Stefan Langerman and Yushi Uno. Threes!, fives, 1024!, and 2048 are hard. In Erik D. Demaine and Fabrizio Grandoni, editors, *8th International Conference on Fun with Algorithms, FUN 2016, June 8-10, 2016, La Maddalena, Italy*, volume 49 of *LIPIcs*, pages 22:1–22:14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/LIPIcs.FUN.2016.22.

13 Neeldhara Misra. Two dots is np-complete. In Erik D. Demaine and Fabrizio Grandoni, editors, *8th International Conference on Fun with Algorithms, FUN 2016, June 8-10, 2016, La Maddalena, Italy*, volume 49 of *LIPIcs*, pages 24:1–24:12. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/LIPIcs.FUN.2016.24.

14 Italia : Consiglio superiore dei lavori pubblici. *Annali dei lavori pubblici.* Eredi di A. De Gaetani. In italian.

15 Ryuhei Uehara and Shigeki Iwata. Generalized Hi-Q is NP-complete. *IEICE Transactions (1976-1990)*, 73(2):270–273, 1990.