



## Trainyard is NP-Hard

Matteo Almanza<sup>a</sup>, Stefano Leucci<sup>b,\*</sup>, Alessandro Panconesi<sup>a,2</sup>



<sup>a</sup> Dipartimento di Informatica, "Sapienza" Università di Roma, Italy

<sup>b</sup> Institute of Theoretical Computer Science, ETH Zürich, Switzerland

### ARTICLE INFO

#### Article history:

Received 14 January 2017

Received in revised form 21 September 2017

Accepted 30 September 2017

Available online 16 October 2017

#### Keywords:

Computational complexity

Puzzle games

Solitaire games

Trainyard

### ABSTRACT

Recently, due to the widespread diffusion of smart-phones, mobile puzzle games have experienced a huge increase in their popularity. A successful puzzle has to be both captivating and challenging, and it has been suggested that these features are somehow related to their computational complexity [6]. Indeed, many puzzle games – such as Mah-Jongg, Sokoban, Candy Crush, and 2048, to name a few – are known to be NP-hard [3,4,8,12]. In this paper we consider Trainyard: a popular mobile puzzle game whose goal is to get colored trains from their initial stations to suitable destination stations. We prove that the problem of determining whether there exists a solution to a given Trainyard level is NP-hard. We also provide an implementation of our hardness reduction.

© 2017 Elsevier B.V. All rights reserved.

## 1. Introduction

The tension between human beings and machines in railroad building dates back to over a century ago, as the famous tale of John Henry testifies. According to the story, John Henry was a *steel-driving man* who challenged the efficiency of steam drill machines in a competition. Eventually John prevailed, reportedly outperforming the rival machine in a contest that lasted more than one day. He unfortunately died of exhaustion shortly after his feat and is now remembered by a statue and a plaque next to the entrance of the Big Bend railroad tunnel in West Virginia. Here, we once again consider a (virtual) challenge between humans and machines in railroad building, except that this time the human ingenuity is put to the test instead of their brute strength. We do so by studying the computational complexity of Trainyard, a smart-phone game where the player is responsible for suitably building railroad tracks. In the words of its author, Trainyard is “a grid-based logic puzzle game where the goal is to get each train from its initial station to a goal station. Every train starts out a certain colour, and most puzzles require the player to mix and merge trains together so that the correctly coloured trains end up at the right stations.” [16].

The game was conceived in 2009 and was first released for iPhones in 2010. In less than five months it climbed the Apple App Store charts becoming the most downloaded application in Italy and the United Kingdom, and the second most downloaded one in the United States [15,17]. It belongs to a class of games known as *casual games*, video games that are targeted at a mass audience, have intuitive rules, and do not require a long-term time commitment to play. The advent

\* Corresponding author.

E-mail addresses: [almanza.1597415@studenti.uniroma1.it](mailto:almanza.1597415@studenti.uniroma1.it) (M. Almanza), [stefano.leucci@inf.ethz.ch](mailto:stefano.leucci@inf.ethz.ch) (S. Leucci), [ale@di.uniroma1.it](mailto:ale@di.uniroma1.it) (A. Panconesi).

<sup>1</sup> The work was done while the author was working in “Sapienza” Università di Roma and was supported in part by the SIR Grant RBS114Q743.

<sup>2</sup> Supported in part by a Google Focused Research Award.



**Fig. 1.** Different kinds of tiles: (a) departure station hosting one red train, (b) red arrival station with a capacity of one, (c) rock, (d) red painter, (e) splitter. Images courtesy of Matt Rix. (For interpretation of the colors in this figure, the reader is referred to the web version of this article.)

of smart-phones boosted the diffusion of casual games and it is estimated that the number of mobile games will surpass 1.8 billions in 2017 [2]. It has been suggested that the reason of this success is somehow related to their computational complexity. As David Eppstein famously said [6]:

*If a game is in P, it becomes no fun once you learn “the trick” to perfect play, but hardness results imply that there is no such trick to learn: the game is inexhaustible. [...]*

*There is a curious relationship between computational difficulty and puzzle quality. To me, the best puzzles are NP-complete [...].*

Over the years several challenging puzzles have been shown to be at least NP-hard, e.g., Mah-Jongg [3], Fifteen-Puzzle [14], Rush Hour [7], Sokoban [4], Super Mario Bros and other classical Nintendo games [1], Bejeweled, Candy Crush and similar match-three games [8], 2048 [12], Two Dots [13], Peg Solitaire [18,9], Portal [5], to cite just a few. We refer the reader to [11] for a 2008 survey, keeping in mind that other puzzles have been proved to be NP-hard ever since, and to [10] for a general framework for showing NP- and PSPACE-hardness of puzzles. It is also known that games exhibiting specific mechanics are NP-hard, as proved in [19]. In this work we show that Trainyard is also NP-hard.

The paper is organized as follows: the next section briefly describes the game rules; in Section 3 we define our problem and state our main theorem, while in Section 4 we describe the technical details of our hardness reduction.

## 2. Game mechanics

The game is played on a board consisting of a rectangular grid divided into square cells. At the beginning, each cell of the board is either empty or it contains a special tile. There are several kinds of tiles, the most important being *departure stations* and *arrival stations* (see Fig. 1 (a) and (b)). The former are train *sources*, i.e., they initially host a certain number of trains that will exit the station as the game proceeds. The latter are train *sinks*: they are initially empty, and they have a maximum capacity (i.e., the maximum number of trains they are able to hold).<sup>3</sup> The player can use empty cells to build different types of tracks. More precisely, the player can place any rail piece on any of the empty cells, where each rail piece can be either a straight track, a 90-degree turn, a crossing, or a switch (see Fig. 2). When the player is satisfied with her design, she can check the solution by simulating it. The simulation proceeds in discrete time steps. At each step the board is updated as follows: all the departure stations that are not empty will output a train in one adjacent tile (depending on the initial orientation of the station), trains will move by one tile following the rails, and arrival stations that are not full will receive incoming trains provided that they are coming from the correct adjacent tile (again, depending on the orientation of the arrival station).

If a train moves into an empty cell, in a cell where the rails have been misplaced, or if reaches the boundaries of the board, it crashes and the game is lost. Trains will also crash if they encounter a rock tile (Fig. 1 (c)), if they try to enter a special tile from the wrong direction (as we will discuss in the following), or if they try to enter an arrival station that is at its maximum capacity. If no crash occurs and the simulation ends up in a state where all the trains have reached their arrival stations, and each arrival station is at full capacity, then the level is successfully completed.

When two trains happen to be traveling on the same rail in the same direction, they *merge* into a single train. If two trains *touch* in any other way, they just proceed unobstructed, i.e., they pass through each other rather than crashing.

**Colors.** To add even more complexity, departure and arrival stations are colored: all the trains exiting a departure station will have its same color, while all the trains entering an arrival station must have a matching color or they will crash. Trains can, however, change their color during their course by traversing special tiles, or by touching or merging with other trains. While in the game there are seven colors, in the following we will only use four: red, blue, purple and brown. If two trains touch (resp. merge), their color will be modified (resp. the color of the resulting train will be chosen) according to the following rules. Let  $A$  and  $B$  be the color of the two trains and let  $C$  be their new color (resp. the color of the resulting train). If  $A$  and  $B$  are the same, then we also have  $C = A = B$ . If one of  $A$  and  $B$  is red and the other is blue, then  $C$  will be purple. In all the remaining cases  $C$  will be brown (as a consequence, if at least one of  $A$  and  $B$  is brown, then  $C$  will be brown as well).

<sup>3</sup> We will only use departure stations hosting a single train and arrival stations with a capacity of one.

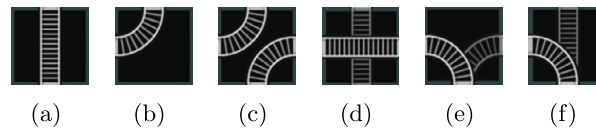


Fig. 2. Different kinds of rail pieces. Each of the pieces can be rotated by 90, 180 or 270 degrees.

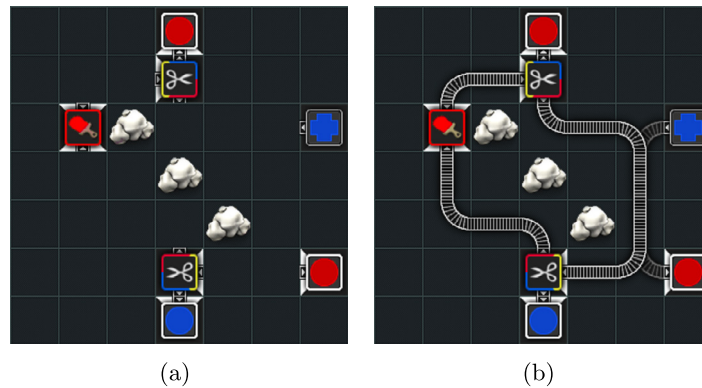


Fig. 3. A level of Trainyard (left) and a possible solution (right).

*Rails and switches.* The different kinds of rail pieces the player can place are shown in Fig. 2. The behavior of some of them is straightforward, while others deserve more attention. The player can rotate the pieces by 90, 180 or 270 degrees before placing them; in the following we will refer to the orientations shown in Fig. 2.

Fig. 2 (d) shows two intersecting rails, these rails can be crossed in both the horizontal and vertical direction but turns are not allowed. Notice also that two trains can cross the tile at the same time without crashing, as we already described. Fig. 2 (e) shows a *switch*. Note that the rail turning to the left is highlighted: this means that if a train comes from the bottom it will continue to the left and the switch will flip to the right; if another train arrives, it will turn to the right and the switch will flip again. If a single train comes from left or right, it will proceed towards the bottom regardless of the current state of the switch but it will still cause the switch to flip. If two trains come from the sides at the same time, they will merge into a single train and the switch will flip. Fig. 2 (f) shows another type of switch, consisting of a straight track and a left turn; here similar rules to the ones we just described apply. When placing the rails, the player is able to choose the initial state of the switches.

Finally, we note that if two trains are on different rails of Fig. 2 (c), they will not touch and hence they will not cause their colors to mix.

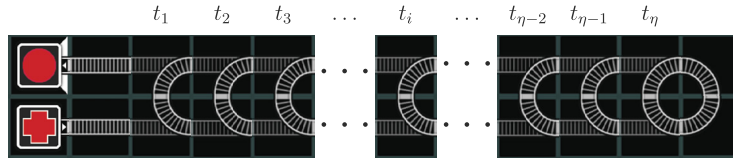
*Special tiles.* Apart from departure and arrival stations, which we already described (see Fig. 1 (a) and (b)), there are also three other special tiles: the *rock*, the *painter* and the *splitter*. A rock is just a tile that causes any entering train to crash, effectively obstructing one available position for the player to place a rail piece (see Fig. 1 (c)).

A painter (shown in Fig. 1 (d)) always has a color  $C$  that is either red or blue and it changes the color of any incoming train to  $C$ . It has two inputs/outputs on opposite sides and can be traversed in both directions (a train entering from a side will exit from the opposite one). Any train entering a painter from one of the two other sides will crash.

The splitter is more involved: it has only one input, marked in yellow, and two outputs on the sides adjacent to the input. One of the outputs (clockwise w.r.t. the input) is marked in blue and the other one is marked in red (see Fig. 1 (e)). Whenever a train enters the splitter from the input, two trains will exit from the two outputs in the next time step (and the original train vanishes). The colors of these two new trains depend on the color  $C$  of the input train: if  $C$  is red, blue, or brown then the two new trains will also have color  $C$ . If  $C$  is purple, then the train exiting from the red-marked output will be red, while the train exiting from the blue-marked output will be blue. Any train trying to enter a splitter from a side different from its input side will crash.

### 3. Our results

In this paper we consider the problem of deciding whether a given level of Trainyard admits a solution (see Fig. 3 for an example instance and one possible solution). More precisely, an instance (or level) of TRAINYARD consists of a rectangular board along with an initial placement of the tiles shown in Fig. 1. We define TRAINYARD as the problem of deciding whether there exists a way to place the rail pieces shown in Fig. 2 on (a subset of) the empty cells so that the simulation will reach



**Fig. 4.** An instance of TRAINYARD of size  $2 \times (\eta + 3)$  and a corresponding solution that requires exponential time to simulate (w.r.t.  $\eta$ ). It is easy to see that the train leaving the only departure station traverses tile  $t_1$  twice. This implies that tile  $t_2$  is traversed exactly 4 times, since the switch in  $t_2$  leads to  $t_1$  only half of the times. Similarly,  $t_3$  is traversed 8 times and, in general,  $t_i$  is traversed  $2^i$  times. As a consequence, the simulation requires at least  $\sum_{i=1}^{\eta} 2^i = \Omega(2^{\eta})$  time steps.

a state where: (i) there are no trains left in the departure stations, (ii) there are no moving trains, (iii) no train crash has happened, and (iv) all the arrival stations received a number of trains matching their capacity.

In the following section we will design a polynomial time reduction from a variant of the boolean satisfiability problem to TRAINYARD, hence proving the following:

**Theorem 1.** TRAINYARD is NP-hard.

We also provide an implementation of our reduction which can be found at <http://trainyard.isnphard.com>.

We note that it is not currently known if TRAINYARD lies in NP, i.e., we do not know whether, given a level and a corresponding design for the rails, it is possible to check – in polynomial time – whether the solution is indeed correct. The trivial simulation strategy fails since it is possible to create solutions that require an exponential number of time steps for the simulation to stop (see, e.g., Fig. 4). On the other hand, TRAINYARD is clearly in PSPACE as the simulation algorithm only needs to keep track of the current state of the board. This requires a polynomial amount of space since, due to train merges, the number of moving trains cannot be larger (up to constant factors) than the size of the board itself. We regard the problem of establishing whether  $\text{TRAINYARD} \in \text{NP}$  as an interesting – and fun – challenge.

#### 4. Our reduction

We prove the NP-hardness of TRAINYARD by showing a polynomial reduction from (the decision version of) *Minimum Monotone Boolean Satisfiability Problem* (MIN-MON-SAT for short). In MIN-MON-SAT we are given (i) a CNF formula  $\phi$  of  $n$  variables  $x_1, \dots, x_n$  and  $m$  clauses  $C_1, \dots, C_m$  such that each clause contains only positive literals, and (ii) an integer  $k$ . The goal is to decide whether there exists a truth assignment for the variables that satisfies  $\phi$  and sets at most  $k$  variables to true. This problem is easily shown to be NP-hard as there is a straightforward reduction from the decision version of *Minimum Dominating Set*: for each vertex  $u$  of the input graph we add a variable  $x_u$  and a clause  $\bigvee_{v \in N[v]} x_v$ , where  $N[v]$  denotes the closed neighborhood of  $v$ . Clearly, there exists a dominating set of size  $k$  iff the formula can be satisfied by setting at most  $k$  variables to true.

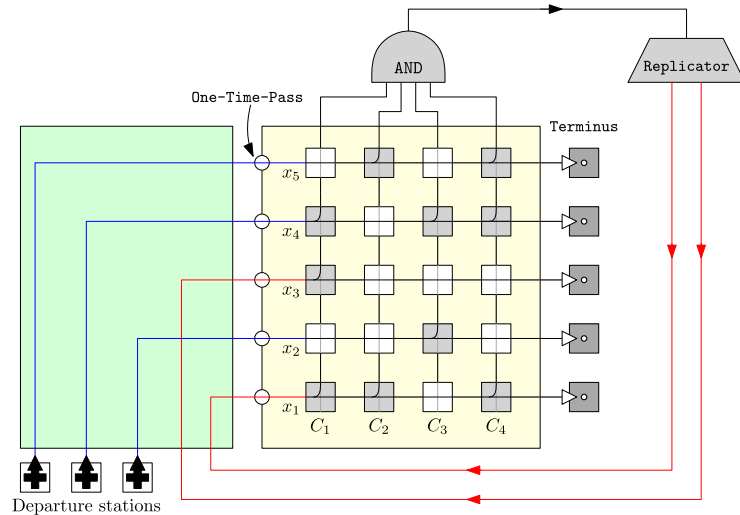
In the following subsections we first give a high-level picture of our reduction, and then we move to the description of the gadgets we use.

##### 4.1. Overview

The overview of our reduction is shown in Fig. 5. In the bottom left there are  $k$  departure stations – one per variable that can be set to true – each containing a single train. The clause area, shown in yellow, encodes the formula  $\phi$  using a  $n \times m$  matrix of gadgets: the gadget on the  $i$ -th row<sup>4</sup> and  $j$ -th column will be one of two different types that we call Cross-Satisfy and Cross-Ignore. More precisely, we use a Cross-Satisfy gadget whenever  $x_i$  is contained in  $C_j$  and a Cross-Ignore gadget otherwise.

To describe how our reduction works, let us look at the case where the formula is satisfiable using  $k$  true variables. Consider, e.g., the instance of Fig. 5 and the satisfying assignment  $x_2 = x_4 = x_5 = \text{true}$ ,  $x_1 = x_3 = \text{false}$ . Here the rails are designed so that the  $k$  trains exiting from the departure stations will traverse the rows of the matrix corresponding to the variables set to true. When a train enters a Cross-Ignore or Cross-Satisfy gadget from the left it will proceed to the right, thus entering the next gadget on the same row. After a train finishes traversing a row (i.e., when it exits from the right side of the last gadget on the row), it is collected by a dedicated Terminus gadget containing an arrival station. Moreover, when a train enters a Cross-Satisfy gadget from the left, it is possible to create a copy of it by suitably placing the rails. If this copy is created, the new train will necessarily exit from the top of the gadget. Intuitively, this means satisfying the clause corresponding to the column using the variable corresponding to the row. Since we started with a

<sup>4</sup> We are counting rows from bottom to top.



**Fig. 5.** A high-level picture of our reduction for an instance of MIN-MON-SAT having 5 variables, 4 clauses, and  $k = 3$ . The corresponding formula is  $(x_1 \vee x_3 \vee x_4) \wedge (x_1 \vee x_5) \wedge (x_2 \vee x_4) \wedge (x_1 \vee x_4 \vee x_5)$ . Cross-Ignore gadgets are shown in white while Cross-Satisfy gadgets are in gray. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

satisfying assignment, it is possible to duplicate (exactly) a train for each column. These trains will move from bottom to top: each time a train enters a Cross-Ignore or Cross-Satisfy gadget from the bottom it will only be allowed to exit from the top. Eventually, all these trains will exit the clause area thus entering the AND gadget: this gadget allows a train to exit from the top iff all  $m$  trains are entering from the bottom, i.e., iff all the clauses have been satisfied.

Notice that, at this point, we still have to bring a train to the Terminus gadgets of the rows corresponding to false variables, e.g.,  $x_1$  and  $x_3$ . Moreover, due to their actual implementation, the gadgets on these rows also need to be traversed by a train going in the left-to-right direction. In order to successfully complete the level, we make  $n - k$  copies of the single train exiting from the AND gadget using a Replicator gadget, and we feed them into such rows. To guarantee that two or more trains can not enter the same row of the matrix, we make use of One-Time-Pass gadgets which are placed at the beginning of each row.

Consider now the scenario in which the  $k$  departing trains traverse a set of rows whose corresponding variables do not satisfy the formula, e.g.,  $x_2, x_3$  and  $x_4$ . In this case it will not be possible to both (i) satisfy the Terminus gadgets of these rows and (ii) allow a train for each column to reach the AND gadget. As a consequence, if the formula is not satisfiable, every possible assignment will necessarily result in a loss.

It is to be noted that, although the player can place the rails in any empty tile of the board, our construction is such that the layout of the rails is essentially forced, except for the green area – which encodes the truth assignment – and for the Cross-Satisfy gadgets (where rails can be placed in two different ways, as we will discuss in the following).

## 4.2. Handling parity issues

Consider two train stations with one train each that are placed next to each other, according to the mechanics of the game, the two exiting trains will never be able to merge. This can be easily seen by considering the *parity* of the trains: if a train occupies the  $i$ -th row and the  $j$ -th column of the board, then its parity is  $(i + j) \bmod 2$ . As the two stations are adjacent, the initial parity of the two trains will differ. Now, according to the game rules, each train moves by exactly one cell per time step, hence flipping its parity. We refer to the initial parity of a train as its *phase*.

It is now easy to verify that two trains can merge iff they have the same phase (also notice that trains exiting from a splitter will have the same phase of the entering train). In order to avoid being distracted by parity issues, in the following we will assume that all trains have the same phase. This can be easily achieved by restricting the placement of the departure stations to tiles whose sum of coordinates is even (i.e., in a checkerboard pattern).

## 4.3. Description of the gadgets

Here we describe how the gadgets used in our reduction can be implemented, and we argue on their correctness.

### 4.3.1. Rails and lanes

In our reduction we will need to move the trains from one gadget to another. However, as we cannot directly place rails in our instance, we need to force the player to build the correct railways between gadgets. This is easily done by creating a



Fig. 6. A lane and the corresponding design of rails.



Fig. 7. Implementation of the *Terminus* gadget.



Fig. 8. Implementation of the *One-Way* gadget.

*lane* of rocks, leaving a single empty tile in-between so that there is only one way for the player to design the rails without causing a train to crash. An example of a lane where rails have already been placed is shown in Fig. 6.

#### 4.3.2. *Terminus* gadget

The *Terminus* gadget is shown in Fig. 7 and it consists of an arrival station preceded by a painter. This allows a train of any color to reach the arrival station.

#### 4.3.3. *One-Way* gadget

The *One-Way* gadget has one input and one output on the opposite side. As the name suggests, it can only be traversed in the input-output direction. Any attempt to traverse the gadget in the opposite direction will cause a train to crash and the player to lose the game. Its implementation is shown in Fig. 8 and it consists of a single splitter with suitable spacing. Notice that there is only one way to design the rails for this gadget that does not cause an incoming train to crash. This gadget will be useful as a component in our other gadgets.

#### 4.3.4. *One-time-pass* gadget

This gadget is similar to the *One-Way* gadget but it has the additional constraint that it must only be traversed exactly once in the whole level. Its implementation is a straightforward modification of the *One-Way* gadget and it is shown in Fig. 9.

#### 4.3.5. *Cross-ignore* gadget

The *Cross-Ignore* gadget is used in the clause area of the matrix each time a variable  $x_i$  does not appear in a clause  $C_j$ . The only ways of placing the rails in this gadget without losing the level ensures that a train can exit from the right (resp. top) only if a train is entering from the left (resp. bottom). The gadget, along with such a design of the rails, is shown in Fig. 10. Notice that the train coming from the left will always be colored red while the one coming from the bottom will be colored blue. To show the correctness of the gadget we only need to argue on rail piece placed on the tile adjacent to



Fig. 9. Implementation of the One-Time-Pass gadget.

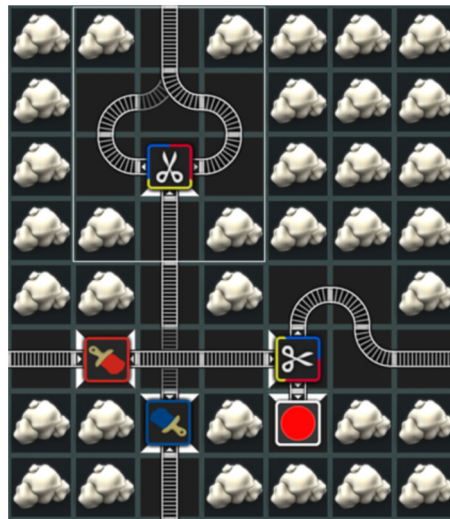


Fig. 10. Implementation of the Cross-Ignore gadget. The One-Way subgadget is highlighted. (For interpretation of the colors in this figure, the reader is referred to the web version of this article.)

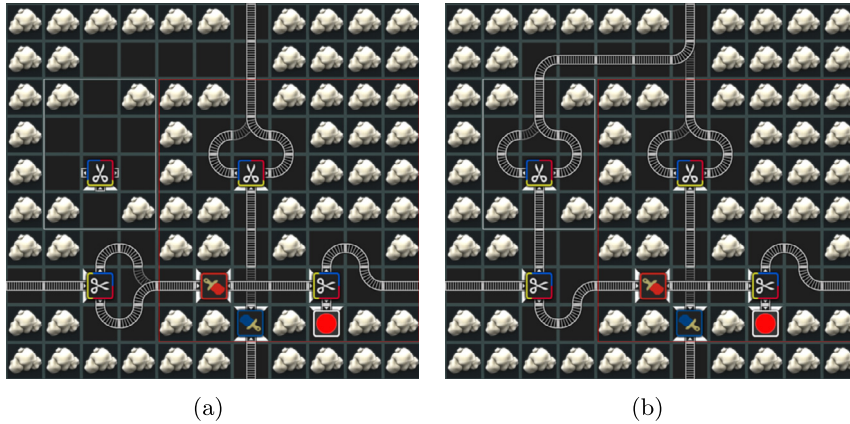
the two painters, since the rest of the rail design is forced. If the rails in that tile cross, as in figure, then it is clear that the gadget works as expected when only one train arrives. If two trains arrive – one from the left and one from the bottom – then either they touch or they do not. If they do not touch, then the red train is split so that one copy can continue to the right while the other goes into the arrival station. If they touch, then they will both become purple. This does not cause any problems since the train going to the right will be split into one blue and one red train: the red train goes into the arrival station and the blue train exits from the right of the gadget.

If the rails on the tile are placed in any way that causes the left train to go up, then the arrival station will always remain empty (as the blue train coming from the bottom painter cannot be sent there), thus losing the level. Otherwise, if the rails merge and continue to the right, then the level will be lost unless *both* trains come at the same time. In that case the arrival station will receive a train and another train will exit from the right, but no train will exit from the top. This case, however, does not cause any problems since it corresponds to “forgetting” that  $C_j$  has already been satisfied by some variable in  $x_1, \dots, x_{i-1}$ , which is never helpful.

Finally, notice that no train can come from the right (due to the final splitter) or from the top (due to the One-Way subgadget).

#### 4.3.6. Cross-satisfy gadget

The Cross-Satisfy gadget is used in the matrix in the clause area each time a variable  $x_i$  appears in a clause  $C_j$ . It works similarly to the Cross-Ignore gadget but the player also has the option to place the rails so that a train is able to exit from the top when the left train reaches the gadget. This encodes the fact that the clause  $C_j$  has been satisfied using variable  $x_i$ . The implementation is shown in Fig. 11 where a One-Way and a Cross-Ignore subgadgets are highlighted in white and red, respectively. Apart from the Cross-Ignore subgadget that we already discussed, the only sensible rail designs that does not cause the player to lose the level are those shown in Fig. 11 (a) and (b). The first one acts exactly as



**Fig. 11.** Implementation of the *Cross-Satisfy* gadget with two possible rail designs. The *One-Way* and *Cross-Ignore* subgadgets are highlighted in white and red, respectively. (For interpretation of the colors in this figure, the reader is referred to the web version of this article.)

a *Cross-Ignore* gadget: the train entering from the left is split and rejoined. In the second design, the train is also split but now a copy continues towards the top while the other copy serves as an input for the *Cross-Ignore* gadget. The two rails going to the top are then joined together in a single rail which is the output of the gadget (here the two *One-Way* gadgets ensure that no train can go back into the gadget). Notice that, when the left and the bottom train both enter the gadget, the second design can cause two trains to exit from the top and one to reach the *Cross-Ignore* subgadget. However, doing this never helps in completing the level and the first design can be chosen instead.

#### 4.3.7. AND gadget

The *AND* gadget takes a number of trains as inputs from the bottom. If all these trains eventually reach the gadget, then the rails can be designed so that a single train will exit from the top. Otherwise, i.e., if at least one train is missing, there is no way of placing the rails to allow a train to exit the gadget. Here we describe how a *AND* gadget with two inputs is implemented. In order to extend the construction to an arbitrary number of inputs trains it suffices to chain together multiple copies of this gadget.

The implementation is shown in Fig. 12 where two distinct areas can be seen. The bottom one is a *buffer* area: here there is some empty space where rails can be placed so that the incoming trains (which are colored in red and blue by the painters) can be merged into a single purple train. This purple train can then proceed to the upper area. In this area there is only one possible rail design that will not result in a train crash, namely the one shown in Fig. 12. It is easy to see that if a single purple train reaches the upper area, this correctly causes two trains (one red and one blue) to reach the arrival stations and one to exit from the top of the gadget. Any other number of trains or any single non-purple train will result in a crash.

#### 4.3.8. Replicator gadget

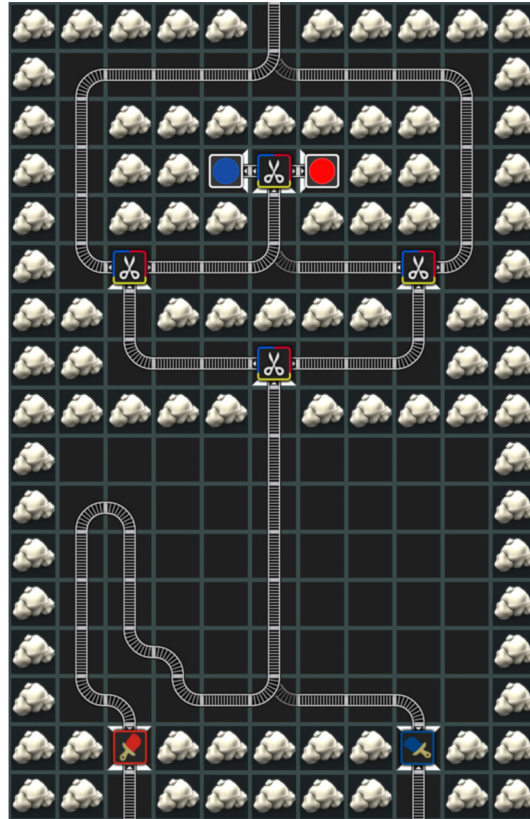
This gadget takes a train from the top as an input and creates a number of copies of it, so that  $n - k$  trains will exit from the bottom. Intuitively, in a yes instance, these trains should enter the rows corresponding to the negated variables. The implementation of a *Replicator* with three outputs is shown in Fig. 13. The construction can be adapted in a straightforward way in order to create as many copies of the incoming train as necessary.

#### 4.4. Final remarks

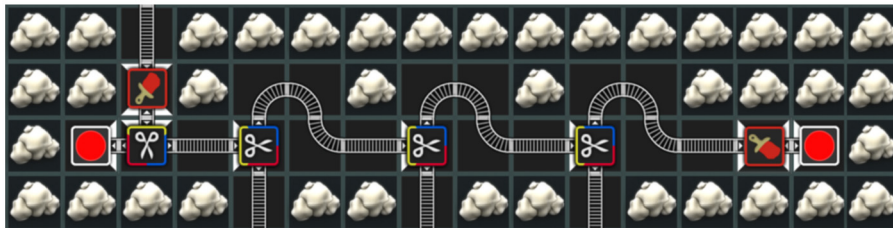
Combining together all the above gadgets, as indicated by Fig. 5, allows us to build the instance of *TRAINYARD* corresponding to the instance of *MIN-MON-SAT*. We can now sketch the proof of Theorem 1, whose correctness follows from the correct operation of the single gadgets.

**Sketch of the proof of Theorem 1.** If there is a truth assignment for *MIN-MON-SAT*, then the  $k$  trains exiting from the departure stations can be fed into the rows of the clause matrix corresponding to variables set to true. For each clause  $C_j$  let  $r_j$  be the index of the first variable that satisfies  $C_j$ , i.e.,  $r_j = \min\{1 \leq i \leq n : x_i = \text{true} \wedge x_i \in C_j\}$ . By construction, the  $j$ -th column of the matrix will have a *Cross-Satisfy* gadget on the  $r_j$ -th row. This means that we can design the rails in the *Cross-Satisfy* gadgets of coordinates  $(r_j, j)$  so that a train will exit from the top. The rails for all the other gadgets in the matrix can be designed so that incoming trains simply cross to the opposite side. This allows exactly one train per column to reach the *AND* gadget, and  $n - k$  trains to exit the *Replicator* gadget. These trains are fed into the  $n - k$  remaining rows to win the level.





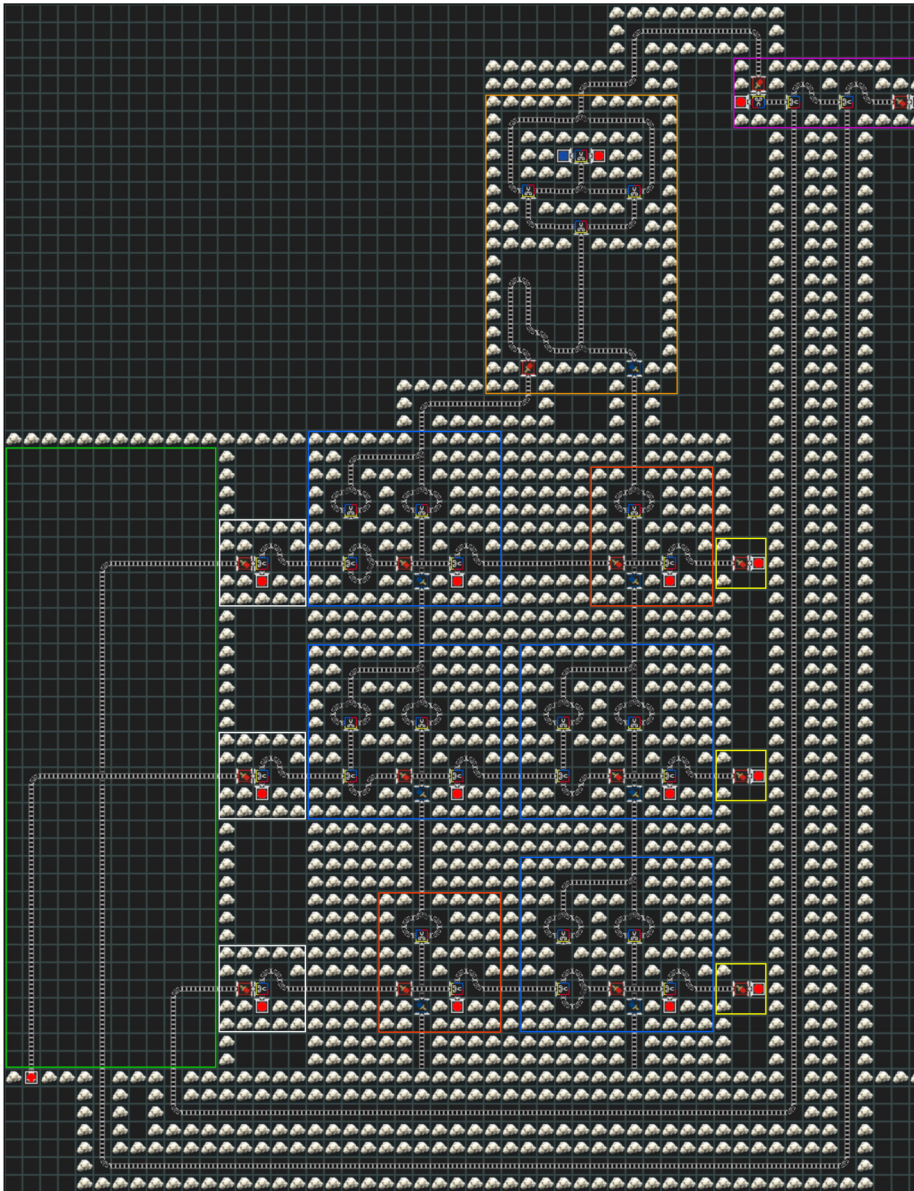
**Fig. 12.** Implementation of the AND gadget. (For interpretation of the colors in this figure, the reader is referred to the web version of this article.)



**Fig. 13.** Implementation of a Replicator gadget with three outputs.

On the other hand, if we have a solution for the instance of TRAINYARD, then this means that exactly one train is entering in each row of the matrix (since they are all preceded by a One-Time-Pass gadget). As there are only  $k$  departure stations, and the only way to make more trains reach the green area is by using the Replicator gadget, this implies that at least  $k$  trains reached the AND gadget (i.e., exactly one train per column). This, in turn, implies the existence of at least one Cross-Satisfy gadget per column where a train is entering from the left. Hence, this Cross-Satisfy gadget must necessarily be placed on a row traversed by a train coming from a departure station. This means that we can find a satisfying truth assignment for the instance of MIN-MON-SAT by setting to true the variables corresponding to the rows where the  $k$  departing trains are entering.  $\square$

A solved instance of TRAINYARD corresponding to the formula  $(x_1 \vee x_2) \wedge (x_2 \vee x_3)$  can be seen in Fig. 14. Moreover, at the web page <http://trainyard.isnphard.com> it is possible to generate instances of TRAINYARD corresponding to arbitrary (small) MIN-MON-SAT formulas and even simulate the possible solutions.



**Fig. 14.** A solved instance of TRAINYARD corresponding to the formula  $(x_1 \vee x_2) \wedge (x_2 \vee x_3)$ . The various gadgets are shown in different colors: One-Time-Pass in white, Terminus in yellow, Cross-Ignore in red, Cross-Satisfy in blue, AND in orange, and Replicator in purple. The perimeter of the green area is also highlighted. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

## Acknowledgements

The authors wish to thank Erik D. Demaine and Rupak Majumdar for insightful and pleasant discussions and email exchanges.

## References

- [1] Greg Aloupis, Erik D. Demaine, Alan Guo, Giovanni Viglietta, Classic Nintendo games are (computationally) hard, *Theoret. Comput. Sci.* 586 (2015) 135–160, <http://dx.doi.org/10.1016/j.tcs.2015.02.037>.
- [2] Casual Games Association, Casual games sector report: Towards the global games market in 2017, <http://www.casualconnect.org/education.html>. (Accessed 22 February 2016).
- [3] Anne Condon, Joan Feigenbaum, Carsten Lund, Peter W. Shor, Random debaters and the hardness of approximating stochastic functions, *SIAM J. Comput.* 26 (2) (1997) 369–400, <http://dx.doi.org/10.1137/S0097539793260738>.
- [4] Joseph Culberson, Sokoban is pspace-complete, in: *Proceedings of the 1st International Conference on Fun with Algorithms (FUN'98)*, vol. 4, 1998, pp. 65–76.

- [5] Erik D. Demaine, Joshua Lockhart, Jayson Lynch, The computational complexity of Portal and other 3d video games, CoRR abs/1611.10319 (2016), <http://arxiv.org/abs/1611.10319>.
- [6] David Eppstein, Computational complexity of games and puzzles, <https://www.ics.uci.edu/~eppstein/cgt/hard.html>. (Accessed 22 February 2016).
- [7] Gary William Flake, Eric B. Baum, Rush hour is PSPACE-complete, or “Why you should generously tip parking lot attendants”, Theoret. Comput. Sci. 270 (1–2) (2002) 895–911, [http://dx.doi.org/10.1016/S0304-3975\(01\)00173-6](http://dx.doi.org/10.1016/S0304-3975(01)00173-6).
- [8] Luciano Gualà, Stefano Leucci, Emanuele Natale Bejeweled, Candy Crush and other match-three games are (NP-)hard, in: Proceedings of the 2014 IEEE Conference on Computational Intelligence and Games (CIG’14), 2014, pp. 1–8.
- [9] Luciano Gualà, Stefano Leucci, Emanuele Natale, Roberto Tauraso, Large Peg-Army maneuvers, in: Proceedings of the 8th International Conference on Fun with Algorithms (FUN’16), 2016, pp. 18:1–18:15.
- [10] Robert A. Hearn, Erik D. Demaine, Games, Puzzles, and Computation, CRC Press, 2009.
- [11] Graham Kendall, Andrew J. Parkes, Kristian Spoerer, A survey of NP-complete puzzles, ICGA J. 31 (1) (2008) 13–34.
- [12] Rahul Mehta, 2048 is (PSPACE) hard, but sometimes easy, CoRR abs/1408.6315 (2014).
- [13] Neeldhara Misra, Two dots is NP-complete, in: Proceedings of the 8th International Conference on Fun with Algorithms (FUN’16), 2016, pp. 24:1–24:12.
- [14] Daniel Ratner, Manfred K. Warmuth, NxN puzzle and related relocation problem, J. Symbolic Comput. 10 (2) (1990) 111–138, [http://dx.doi.org/10.1016/S0747-7171\(08\)80001-6](http://dx.doi.org/10.1016/S0747-7171(08)80001-6).
- [15] Matt Rix, The story so far, <http://struct.ca/2010/the-story-so-far/>. (Accessed 22 February 2016).
- [16] Matt Rix, Trains on paper, <http://struct.ca/2010/trains-on-paper/>. (Accessed 22 February 2016).
- [17] Matt Rix, The week that was, <http://struct.ca/2010/the-week-that-was/>. (Accessed 22 February 2016).
- [18] Ryuhei Uehara, Shigeki Iwata, Generalized Hi-Q is NP-complete, IEICE Trans. 1976–1990 73 (2) (1990) 270–273.
- [19] Giovanni Viglietta, Gaming is a hard job, but someone has to do it!, Theory Comput. Syst. 54 (4) (2014) 595–621, <http://dx.doi.org/10.1007/s00224-013-9497-5>.