Trainyard is NP-Hard

Matteo Almanza[1], Stefano Leucci[2], and Alessandro Panconesi[3]

1 **Rome, Italy**
`almanza.1597415@studenti.uniroma1.it`
2 **Dipartimento di Informatica, "Sapienza" Università di Roma, Italy**
`leucci@di.uniroma1.it`
3 **Dipartimento di Informatica, "Sapienza" Università di Roma, Italy**
`ale@di.uniroma1.it`

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 this features are somehow related to their computational complexity [5]. 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, 7, 10]. 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.[1]

1998 ACM Subject Classification F.2.m Miscellaneous

Keywords and phrases Complexity of Games, Trainyard

Digital Object Identifier 10.4230/LIPIcs.FUN.2016.2

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 outperfoming 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.*" [13].

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 in the United

[1] `http://trainyard.isnphard.com`

States [12, 14]. 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 of smart-phones boosted the diffusion of casual games and it is estimated that 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 [5]:

> *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 [11], Rush Hour [6], Sokoban [4], Super Mario Bros and other classical Nintendo games [1], Bejeweled, Candy Crush and similar match-three games [7], 2048 [10], just to cite a few. We refer the reader to [9] for a 2008 survey, although other puzzles have been proved to be NP-hard ever since, and to [8] for a general framework for showing NP- and PSPACE-hardness puzzles. It is also known that games exhibiting certain mechanics are NP-hard [15]. Here we show that Trainyard is also NP-hard.

The paper is organized as follows: next section briefly describes the game rules; in Section 3 we define our problem and state our main theorem, and in Section 4 we describe the 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 kind of tiles, the most important being *departure stations* and *arrival stations*: the first ones host a number of trains while the latter are initially empty and have a maximum capacity, i.e, a number of trains they are able to hold (see Figure 1 (a) and (b)).[2] The player can use empty cells to build different types of tracks. More precisely, the player can place any rail piece on each of the empty cells, where rail pieces can be either straight tracks, 90 degree turns, crossings, or switches (see Figure 2). When the player is satisfied with his design, he can check his 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 user's rails, and arrival stations that are not full will receive incoming trains if they are coming from the right 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 out of the board, it crashes and the level is lost. Trains will also crash 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 in an arrival station that is at full capacity. If the simulation reaches a state where all the trains have reached their arrival stations and each of these stations is at full capacity, without ever encountering a crash, then the player wins.

[2] We will only use departure stations hosting a single train and arrival stations with a capacity of one.

(a) (b) (c) (d) (e)

Figure 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.

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 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 due to special tiles and by touching or merging with other trains. In our paper we only care of four colors: 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 coincide, 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 A or B are brown, then C will be brown as well).

Rails and switches

The different kinds of rail pieces the user can place are shown in Figure 2. The behavior of some of them is straightforward, while others deserve more attention. The user can rotate the pieces by 90, 180 or 270 degrees before placing them; in the following we will refer to the orientation shown in figure.

Figure 2 (d) shows two intersecting rails, these rails can be crossed in both the horizontal and vertical direction but turns are not allowed. Also note that two trains can cross the tile at the same time without crashing, as we already described. Figure 2 (e) shows a *switch*. Notice that the rail turning to the left is highlighted, this means 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 this 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. Figure 2 (e) 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 the Figure 2 (c), they will not touch and hence they won't cause their colors to mix.

Special tiles

Apart from the departure and arrival stations, that we already described (see Figure 1 (a) and (b)), there are also three other special tiles: the *rock*, the *painter* and the *splitter*. A

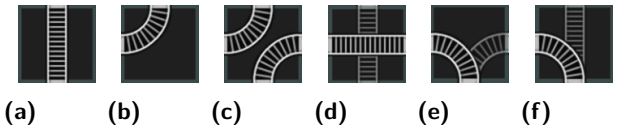


Figure 2 Different kinds of rails. Each of the pieces can be rotated by 90, 180 or 270 degrees.

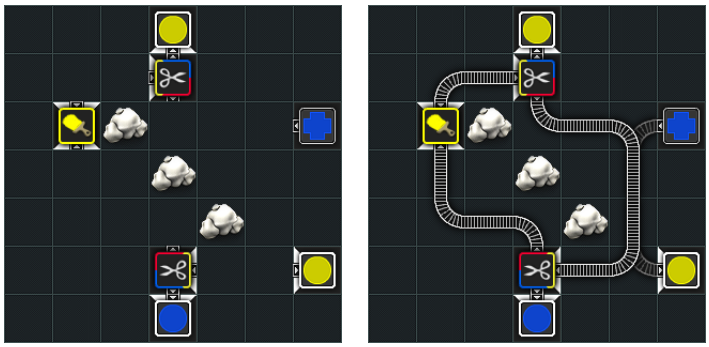


Figure 3 A level of Trainyard (left) and a possible solution (right).

rock is just a tile that causes any entering train to crash, effectively removing one position available for the player to place a rail piece (see Figure 1 (c)).

A painter (shown in Figure 1 (d)) always has a color C that is either red or blue, and will change 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 trying to enter a painter from one of the other two sides will crash.

The splitter is more involved: it has only one input, marked in yellow, and two outputs in 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 Figure 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 be colored C. If C is purple, then the train exiting from the red-marked output will be red, and other 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. More precisely, we define Trainyard as the following decision problem: given a rectangular board and an initial placement of the tiles shown in Figure 1, is there a way to place the rails pieces shown in Figure 2 on (a subset of) the empty cells so that the simulation will reach 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 station 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 actual implementation of our reduction which can be found at `http://trainyard.isnphard.com`.

We note that we do not actually know 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, that the solution is indeed correct. The trivial simulation strategy fails as it is possible to create solutions that require an exponential number of time steps for the simulation to stop. 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 polynomial space since, due to the merge rule, the number of moving trains cannot be asymptotically larger than the size of the board itself. We regard the problem of establishing whether Trainyard $\in$ 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 ϕ of n variables $x_1, \ldots, x_n$ and m clauses $C_1, \ldots, C_m$ 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 ϕ 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 an 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 Figure 4. In the bottom left we have 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 ϕ using a $n \times m$ matrix of gadgets: the gadget on the i-th row[3] 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 Figure 4 and the satisfying assignment $x_2 = x_4 = x_5 =$ true, $x_1 = x_3 =$ 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 last the 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

[3] We are counting rows from bottom to top.

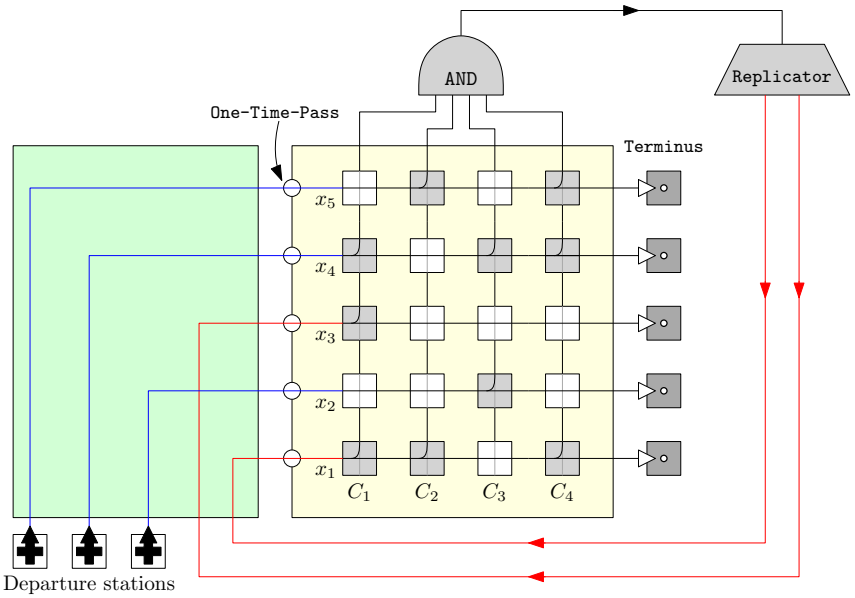


Figure 4 A high-level picture of our reduction for an instance of Min-Mon-SAT having 5 variables, 4 clauses, and $k = 3$. The corresponding formula is $(x_1 \vee x_3 \vee x_4) \wedge (x_1 \vee x_5) \wedge (x_2 \vee x_4) \wedge (x_1 \vee x_4 \vee x_5)$. `Cross-Ignore` gadgets are shown in white while `Cross-Satisfy` gadgets are in gray.

the column using the variable corresponding to the row. Since we started with a satisfying assignment, it is possible to duplicate 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 and they will enter the `AND` gadget: this gadget allows a train to exit form 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-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 a suitable `One-Time-Pass` gadget which is placed at the beginning of each row.

Now consider the case where 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 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

Figure 5 A lane and the corresponding design of rails.

Figure 6 Implementation of the `Terminus` gadget.

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 mechanics of the game, 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*.

Actually, it is possible to show 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 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 *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 Figure 5.

4.3.2 Terminus Gadget

The `Terminus` gadget is just an arrival station preceded by a painter to ensure that the incoming train will be of the correct color, as it is shown in Figure 6.

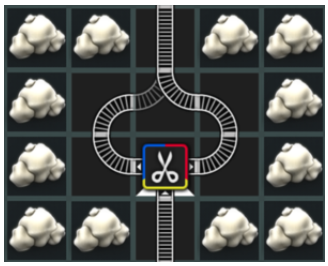


Figure 7 Implementation of the `One-Way` gadget.

Figure 8 Implementation of the `One-Time-Pass` gadget.

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 hence the player to lose the level. Its implementation is shown in Figure 7 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 its shown in Figure 8.

4.3.5 Cross-Ignore Gadget

The `Cross-Ignore` gadget is used in the matrix in the clause area 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 Figure 9. Notice that the train coming from the left will always be colored red while the one coming form 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 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 if only one train arrives the gadget works as expected. If two trains arrive, one

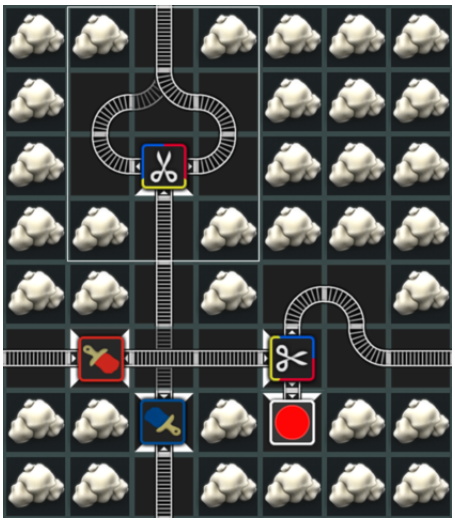


Figure 9 Implementation of the `Cross-Ignore` gadget. The `One-Way` subgadget is highlighted.

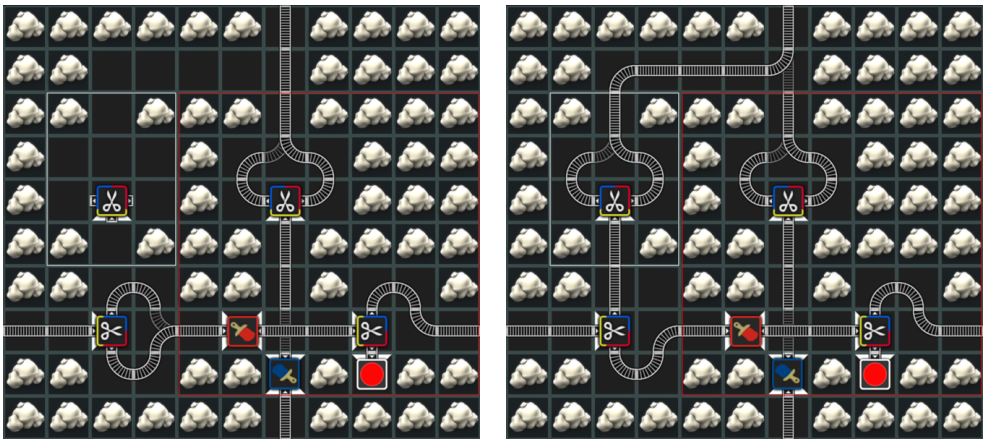
from the left and one from the bottom, then either they touch or they do not. If they do not touch, then the red train is split so that one copy can continue to the right while the other goes into the arrival station. If they touch, then they will both become purple. This does not cause any problems 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 top of the gadget.

If the rails on the tile are placed in any way that causes that the left train to go up, then the arrival station will always remain empty (as the blue train 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 $x_1, \ldots, x_{i-1}$, which is never convenient.

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 Figure 10 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 do not make the player lose the level, are those shown in Figure 10 (a) and (b). The first one acts exactly as a `Cross-Ignore` gadget: the train entering from the left is just split and rejoined. In the second design, the train is also split but now a copy continues towards the top while the other 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



(a) (b)

Figure 10 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.

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 train 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 reach the top. On the converse, if at least one train is missing, then there is no way of placing the rails to make a train exit the gadget. Here we describe how such a 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 Figure 11 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 read 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 Figure 11. 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 Figure 12. The construction can be adapted in a straightforward way in order to create as many copies of the incoming train as necessary.

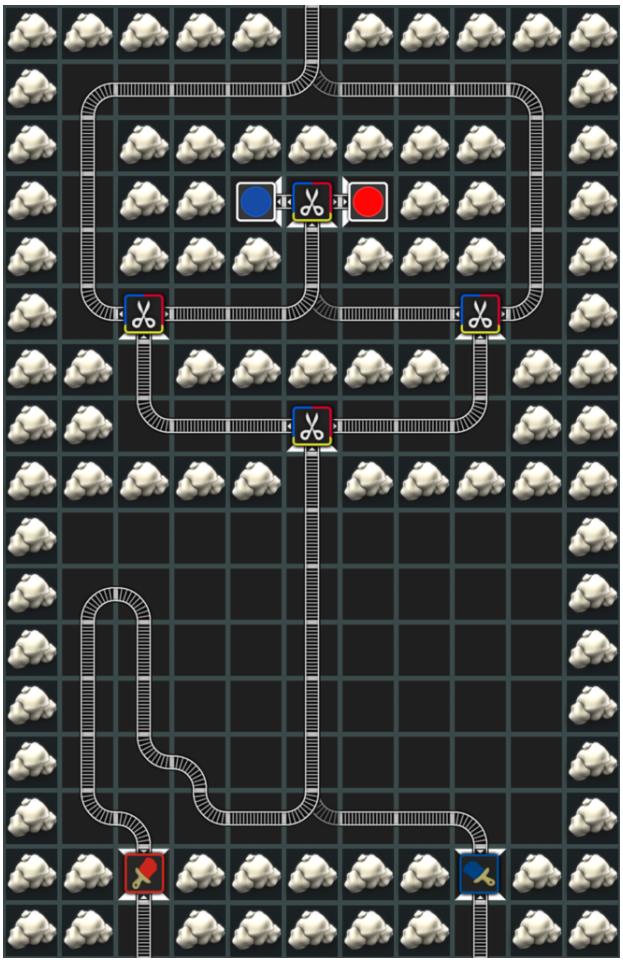


Figure 11 Implementation of the `AND` gadget.

4.4 Final Remarks

Combining together all the above gadgets, as indicated by Figure 4, 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 just cross to the opposite sides. This allows exactly one train per column to reach the `AND` gadget, and $n - k$ trains to exit the `Replicator`. These trains are fed into the $n - k$ remaining rows to win the level.

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

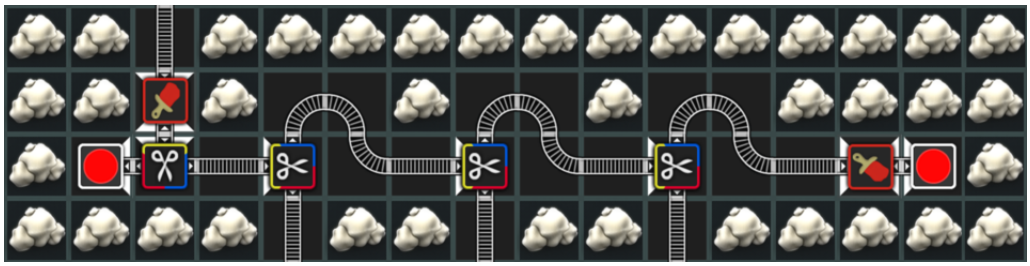


Figure 12 Implementation of a `Replicator` gadget with three outputs.

`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 (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. ◀

A solved instance of Trainyard corresponding to the formula $(x_1 \vee x_2) \wedge (x_2 \vee x_3)$ can be seen in Figure 13. Moreover at `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.

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, and Giovanni Viglietta. Classic nintendo games are (computationally) hard. *Theoretical Computer Science*, 586:135–160, 2015. `doi: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: 2016-02-22.

3 Anne Condon, Joan Feigenbaum, Carsten Lund, and Peter W. Shor. Random debaters and the hardness of approximating stochastic functions. *SIAM Journal on Computing*, 26(2):369–400, 1997. `doi:10.1137/S0097539793260738`.

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

5 David Eppstein. Computational complexity of games and puzzles. `https://www.ics.uci.edu/~eppstein/cgt/hard.html`. Accessed: 2016-02-22.

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

7 Luciano Gualà, Stefano Leucci, and 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*, pages 1–8, 2014. `doi:10.1109/CIG.2014.6932866`.

8 Robert A. Hearn and Erik D. Demaine. *Games, puzzles, and computation*. CRC Press, 2009.

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

10 Rahul Mehta. 2048 is (PSPACE) hard, but sometimes easy. *CoRR*, abs/1408.6315, 2014.

11 Daniel Ratner and Manfred K. Warmuth. Nxn puzzle and related relocation problem. *Journal of Symbolic Computation*, 10(2):111–138, 1990. doi:10.1016/S0747-7171(08)80001-6.

12 Matt Rix. The story so far. http://struct.ca/2010/the-story-so-far/. Accessed: 2016-02-22.

13 Matt Rix. Trains on paper. http://struct.ca/2010/trains-on-paper/. Accessed: 2016-02-22.

14 Matt Rix. The week that was. http://struct.ca/2010/the-week-that-was/. Accessed: 2016-02-22.

15 Giovanni Viglietta. Gaming is a hard job, but someone has to do it! *Theory Computing Systems*, 54(4):595–621, 2014. doi:10.1007/s00224-013-9497-5.

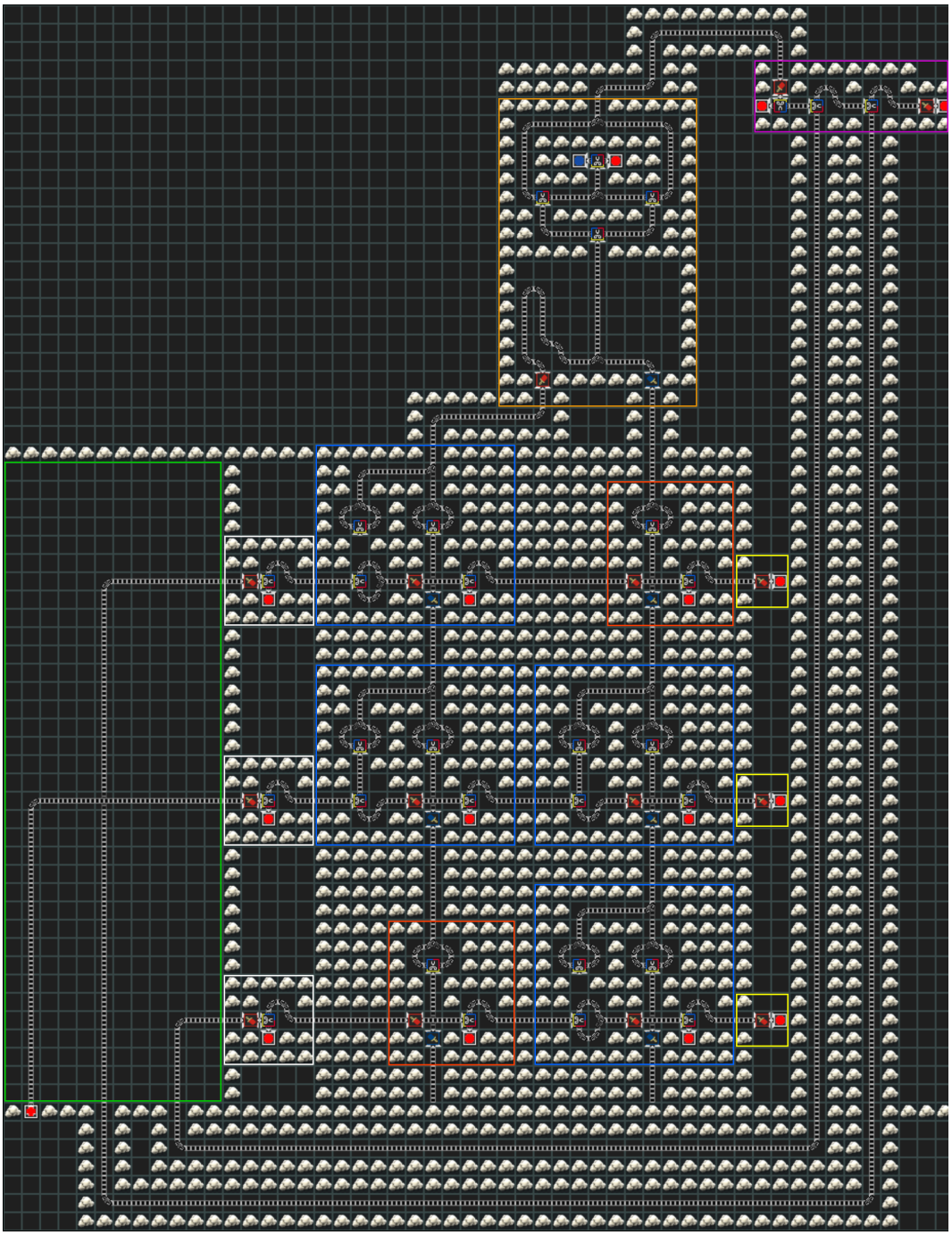


Figure 13 A solved instance of TRAINYARD corresponding to the formula $(x_1 \vee x_2) \wedge (x_2 \vee x_3)$. The green area is shown in green and the various gadgets are highlighted in different colors: `One-Time-Pass` in white, `Terminus` in yellow, `Cross-Ignore` in red, `Cross-Satisfy` in blue, `AND` in orange, and `Replicator` in purple.