Coordinating “7 Billion Humans” Is Hard

Alessandro Panconesi
Sapienza University of Rome, Italy

Pietro Maria Posta
Sapienza University of Rome, Italy

Mirko Giacchini
Sapienza University of Rome, Italy

Abstract

In the video game “7 Billion Humans”, the player is requested to direct a group of workers to various destinations by writing a program that is executed simultaneously on each worker. While the game is quite rich and, indeed, it is considered one of the best games for beginners to learn the basics of programming, we show that even extremely simple versions are already NP-Hard or PSPACE-Hard.

2012 ACM Subject Classification Theory of computation → Complexity classes

Keywords and phrases video games, computational complexity, NP, PSPACE

Digital Object Identifier 10.4230/LIPIcs.FUN.2024.26

1 Introduction

In a world where robots have occupied every possible job position, humans are finally free to dedicate themselves to their favourite pastimes. However, in this utopian world, the work ethic of yesteryear reigns supreme and the only thing humans desire is good-paying jobs. To appease them, the robots construct one colossal building, so colossal to be visible from outer space, and hire all 7 billion humans living on Earth as well-paid white-collar workers. Now, robots are faced with the challenge of coordinating the human workforce to keep them all constantly entertained.

It is in such a world that “7 Billion Humans” takes place. Released in 2018 as the successor of “Human Resource Machine”, “7 Billion Humans” is a puzzle video game praised by tech reviewers as one of the best games to learn the basics of programming [16, 17, 18]. The player, who takes on the role of the robots in the story, must coordinate a group of workers by specifying their actions by means of an ad-hoc programming language. While the programming language in the game is quite rich, also containing “if” statements and “go-to” commands, in this paper, we will focus on the core mechanic of the game: moving the workers. In particular, the goal is to move the workers into an accepted configuration by writing a program that is executed simultaneously by each one of them. While moving, the workers will have to navigate through walls, desks, plants, and other objects that block their movement, as well as holes where workers can fall through. This extremely limited set of commands and objects constitutes the core of the game since they appear in essentially all the game levels. Even under these limitations, we show that the player (and hence the robots) will have a hard time coordinating the humans.

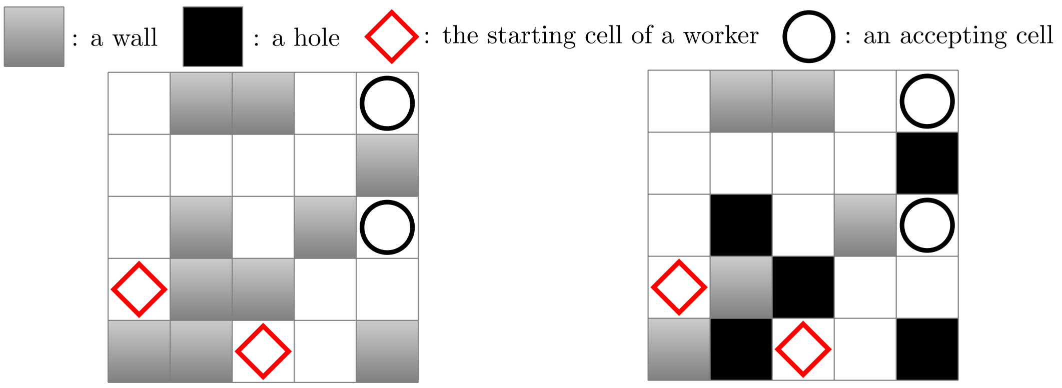
Our work falls within the rich area of video-game computational complexity. In recent years, several extremely popular video games have been proven to be NP-Hard or PSPACE-Hard, such as Super Mario Bros. and other Nintendo games [3, 6], Portal and several other 3D games [5], Trainyard [1, 2], Candy Crush [9], and many others [10, 14].

Let us now describe the game “7 Billion Humans” and our contributions.

1.1 Our Contributions

A level of “7 Billion Humans” consists of a grid of cells containing *workers* and *objects*. The player must write a program, which is executed by *every* worker simultaneously, in order to satisfy the requests of the level designer. We consider only the most basic kind of request: the workers must be moved from their starting configuration into an accepted configuration. More precisely, some cells of the grid are *accepting cells* and to solve a level, all the workers must be standing on an accepting cell after executing the program. There are many commands at the disposal of the user to write the program, but we make use only of the most basic one: `step {direction}`, which is used to move the workers (all at the same time) by one cell in a given direction, which can be one of `up`, `down`, `left`, `right`, `up-left`, `up-right`, `down-left`, or `down-right`. A cell can either be empty or contain an object.[1] The simplest type of objects are the *walls* and, as one might expect, stepping into a wall results in a non-movement.[2]

We show in figure 1a an example of a level. We abbreviate the command to step in one direction with `u`, `d`, `l`, `r`, `(ul)`, `(ur)`, `(dl)`, `(dr)`, and a sequence of steps in the same direction using exponentiation (e.g., $\mathtt{r}^4$ instead of `rrrr`). For a generic finite alphabet Σ, we denote with Σ^* the set of all finite strings consisting of symbols of Σ. A program is therefore represented as a string over the alphabet $\{\mathtt{u}, \mathtt{d}, \mathtt{l}, \mathtt{r}, \mathtt{(ul)}, \mathtt{(ur)}, \mathtt{(dl)}, \mathtt{(dr)}\}$. For simplicity, when a program $\pi \in \{\mathtt{u}, \mathtt{d}, \mathtt{l}, \mathtt{r}, \mathtt{(ul)}, \mathtt{(ur)}, \mathtt{(dl)}, \mathtt{(dr)}\}^*$ solves a level, we also say that the level *accepts* the string π.



(a) This level can be solved, for example, by the program $\mathtt{u}^2\mathtt{r}^3\mathtt{uru}$ or by the program $\mathtt{ru}^2\mathtt{r}^3\mathtt{ur}$. Instead, `ruru` does not solve the level because only one worker reaches an accepting cell.

(b) This level can be solved by the program $\mathtt{ru}^2\mathtt{r}^3\mathtt{ur}$. Instead, the program $\mathtt{u}^2\mathtt{r}^3\mathtt{uru}$ does not work anymore because one of the two workers would get stuck in a hole.

Figure 1 Two examples of game levels. The level on the left contains only walls and empty cells, while the one on the right also contains holes. We assume that the grids are surrounded by walls.

The decision problem that we consider is: given a level of “7 Billion Humans”, say if the level is solvable or not. Since we included only the essential elements of the game, we call this problem `7BH-Essential`. We will show that even this extreme simplification is already NP-Hard.

[1] We can assume that workers start in empty cells and the accepting cells are all empty.

[2] In the game there are also other obstacles, such as desks and plants: since they all act the same, we will always talk about walls. Other workers also behave as obstacles if hit. However, this will never happen in our reductions.

▶ **Theorem 1.** *It is NP-Hard to check if a given level of "7 Billion Humans" is solvable, even using only* walls, *empty cells, and the* **`step`** *command. That is,* *`7BH-Essential` is NP-Hard.*

Holes are another common object in the game. When a worker steps on a cell containing a *hole*, it gets stuck for the rest of the computation. Since our goal is to have each worker on an accepting cell, even if a single worker steps on a hole, the level is lost. An example of a level with holes is shown in figure 1b. We call `7BH-Holes` the decision problem previously described where we add *holes* in addition to the already mentioned elements. Adding holes might make the problem more difficult, in fact, `7BH-Holes` is PSPACE-Hard.

▶ **Theorem 2.** *It is PSPACE-Complete to check if a given level of "7 Billion Humans" containing only* walls, holes, *empty cells, and using only the* **`step`** *command, is solvable. That is, `7BH-Holes` is PSPACE-Complete.*

The paper is organized as follows. In Section 2 we highlight some connections between ours and other problems. In Sections 3 and 4 we prove our two main theorems and finally, in Section 5, we discuss some final remarks.

2 Relations with other problems

In our reductions, the game level of "7 Billion Humans" will be divided into isolated sub-levels, each containing a single worker. Then, to solve the level, all the sub-levels must be solved simultaneously. Moreover, in our reductions, we will prevent diagonal movements: the only admissible directions are `right`, `left`, `up`, and `down`. This special structure of the level can be used to draw some relations with other problems.

2.1 Simultaneous Maze Solving

Each sub-level can be interpreted as a grid maze: the worker must find a path from its starting position to one of the accepting cells,[3] avoiding the holes and navigating through the walls. Our results, then, entail that solving multiple mazes simultaneously is NP-Hard (Theorem 1), or PSPACE-Hard if the mazes can contain holes (Theorem 2). The only other work on the topic, to the best of our knowledge, is the one of Funke et al. [7], which, however, studies very special mazes that are always solvable simultaneously.

2.2 Intersection Non-Emptiness

Each sub-level can also be interpreted as a deterministic finite automaton (DFA for short). In particular, a sub-level $w \times h$ naturally translates into a DFA with at most $w \cdot h$ states (corresponding to the cells), and with the transition function on the alphabet $\{\mathtt{u}, \mathtt{d}, \mathtt{l}, \mathtt{r}\}$ that simulates the behavior of the cells. Solving all the sub-levels is equivalent to finding a string that is accepted by a set of DFAs: this is a fundamental problem in automata theory known as Intersection Non-Emptiness Problem [4, 12, 15] and first shown to be PSPACE-Complete by Kozen [11]. The structure of our DFAs is very special: the undirected transition graph, excluding self-loops, is a subgraph of the $w \times h$ grid graph. Therefore, our Theorem 2 entails that Intersection Non-Emptiness is PSPACE-Complete even with this class of automata.

[3] Note that passing upon an accepting cell is not enough: each worker must be standing on an accepting cell at the end of the sequence of moves.

To the best of our knowledge, given the strong restrictions that we have to make on the DFAs, our results are not derivable from existing work. As an example, in the original PSPACE-Hardness proof of Kozen [11], the standard construction of the DFAs contains vertices having in-degree $(|Q| + |\Sigma| + 1)^3$, where Q and Σ are the set of states and input symbols of a Turing Machine, therefore, the in-degree is at least 27 and possibly much larger. Instead, our DFAs have an in-degree of at most 8, considering self-loops. Moreover, in such proof, the automata use several one-way transitions; instead, the transitions of our DFAs are reversible (except for states associated with holes).

3 NP-Hardness of `7BH-Essential`

In this section we prove Theorem 1. In particular, we show a polynomial-time reduction from Positive 1-in-3-SAT, notoriously known to be NP-Hard (see, e.g., [8, page 259, problem LO4]), to `7BH-Essential`.

▶ **Definition 3** (Positive 1-in-3-SAT). *The input of Positive 1-in-3-SAT consists of n boolean variables, $x_1, x_2, \dots, x_n$, and a set of m clauses, each containing exactly three distinct positive variables (i.e., there are no negated literals). The goal is to find a truth assignment to the variables such that each clause contains exactly one true variable.*

Fixed an instance of Positive 1-in-3-SAT, we make use of three types of gadgets: (i) the *diagonal* gadget, to prevent diagonal movements, (ii) the *assignment gadget* that, intuitively, allows assigning truth values to the variables, and (iii) one *clause gadget* for each clause, to ensure that the truth assignment satisfies the original 1-in-3 formula. Each gadget will be built with multiple independent sub-levels that must be solved simultaneously. The final game level is obtained by stacking the sub-levels together and isolating them via walls.

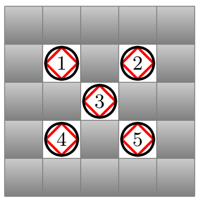


Figure 2 The *diagonal* gadget, used to prevent diagonal movements, consists of five sub-levels: for $i \in \{1, 2, 3, 4, 5\}$, the i-th sub-level contains only the worker and the accepting cell labeled with i.

The diagonal gadget is reported in figure 2. It consists of five sub-levels that we draw together for brevity. In particular, for $i \in \{1, 2, 3, 4, 5\}$, the i-th sub-level contains only the one worker and the one accepting cell labeled with i. Suppose the program contains a diagonal movement (i.e., `(ul)`, `(ur)`, `(dl)`, or `(dr)`), then, worker 3 would "overlap"[4] with another worker and it would become impossible to solve all the five sub-levels of the gadget (indeed, once two workers are overlapped in different sub-levels having the same walls, it is impossible to separate them).

3.1 Assignment Gadget

This gadget consists of four sub-levels, reported together in figure 3. For $i \in \{1, 2, 3, 4\}$, the i-th sub-level contains only the i-th worker and the accepting cells labeled with i.

[4] the workers are in different sub-levels, so they "overlap" if we imagine the sub-levels on top of each other

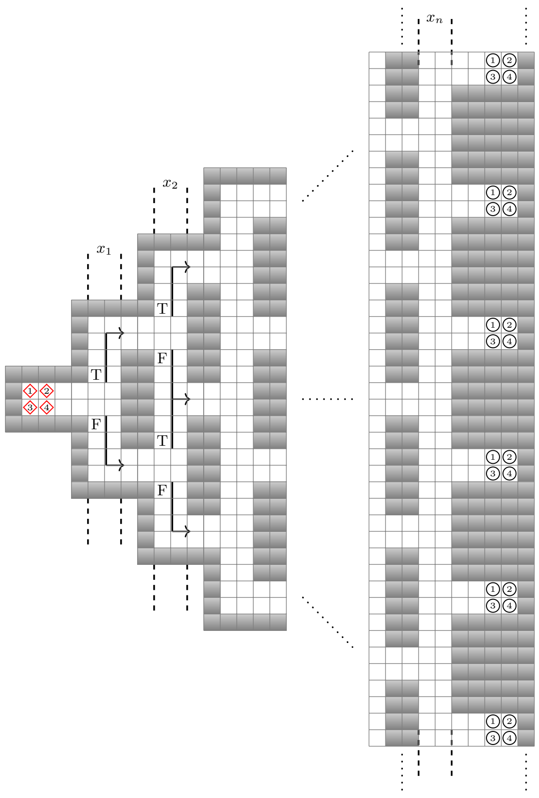


Figure 3 Assignment Gadget. It consists of four sub-levels, drawn together for brevity. In particular, the i-th sub-level, for $i \in \{1, 2, 3, 4\}$, contains only the worker with label i on the left, and only the accepting cells with label i on the right. The workers select the truth value of the variables by moving `up` or `down`.

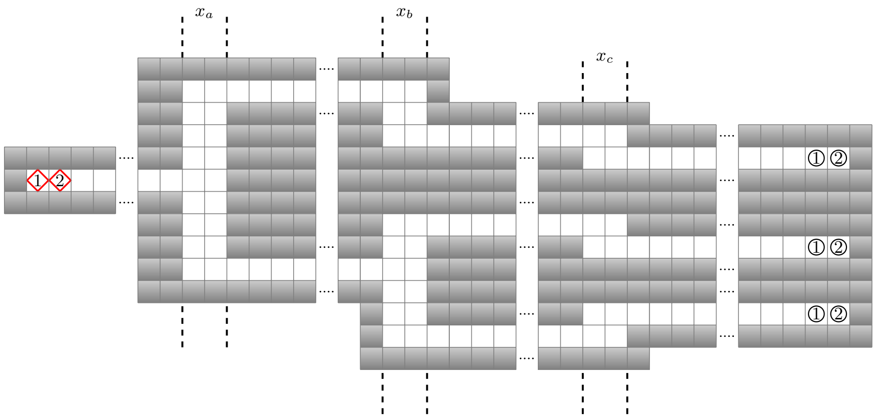
By using four sub-levels, the gadget ensures that the workers cannot hit the walls, indeed, if that happened, two workers would overlap and at least one sub-level would become unsolvable (e.g., using the program $\mathtt{r}^5$, workers 1 and 2 overlap, making it impossible for them to simultaneously stand on an accepting cell). This property will be used in the clause gadgets.

Since the accepting cells are at the right end of the sub-levels, the workers must pass through all the variables, and select their truth values by moving `up` or `down`. Specifically, when the workers are in correspondence with the variable x_i (that is, when the workers are in columns $4i + 1$ and $4i + 2$, assuming that the columns are numbered from left to right starting with 0 on the left wall), moving `up` will set x_i to true in all clauses while moving `down` will set it to false. Intuitively, we can think that the workers will move via a program of the form $\mathtt{r}^4\sigma_1^4\mathtt{r}^4 \ldots \sigma_n^4\mathtt{r}^4$ with $\sigma_i \in \{\mathtt{u}, \mathtt{d}\}$, and σ_i determines the truth value of the i-th variable ($\mathtt{u}$ = True, $\mathtt{d}$ = False).

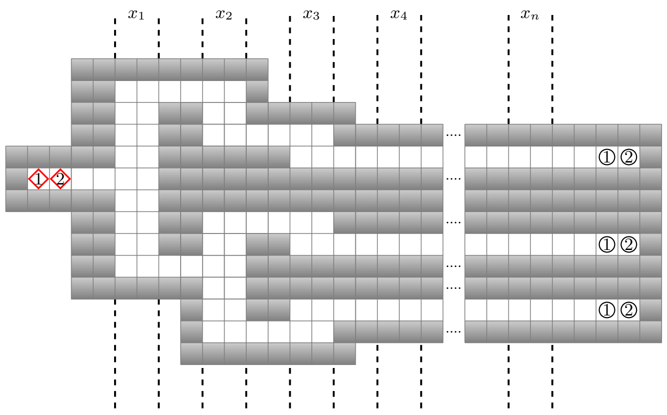
▶ Remark. Note that the workers have more freedom of movement than what we would like. For example, they can move `left`, and in correspondence with a variable, they can move `up` and `down` multiple times before going `right`. However, our clause gadgets are such that this extra freedom does not permit cheating. More precisely, if the level can be solved, then it can also be solved by a program of the form $\mathtt{r}^4\sigma_1^4\mathtt{r}^4 \ldots \sigma_n^4\mathtt{r}^4$ with $\sigma_i \in \{\mathtt{u}, \mathtt{d}\}$.

3.2 Clause Gadgets

For each clause $C = (x_a, x_b, x_c)$, with $a < b < c$, we create a clause gadget consisting of two sub-levels. In figure 4a, we report the sub-levels for the generic clause C. For the reader's convenience, we also show, in figure 4b, the gadget for the clause (x_1, x_2, x_3). For $i \in \{1, 2\}$, the i-th sub-level contains only the i-th worker and the accepting cells labeled with i.



(a) Gadget for the generic clause (x_a, x_b, x_c), with $1 \leq a < b < c \leq n$.



(b) Gadget for the clause (x_1, x_2, x_3). We can handle all the clauses by stretching opportune sections of the gadget (see figure 4a).

Figure 4 Figure (a) shows the Clause Gadget for the generic clause (x_a, x_b, x_c). It consists of two sub-levels drawn together for brevity. The i-th sub-level, $i \in \{1, 2\}$, contains only the worker with label i on the left and the accepting cells with label i on the right. The columns associated with the variables are aligned with those of the assignment gadget (figure 3). Solely for the sake of clarity, we also show in figure (b) the gadget for the particular clause (x_1, x_2, x_3).

Let us first argue that the flexibility of the *assignment gadget* cannot be exploited to generate invalid truth assignments. Observe that the two workers of the clause gadget (of the two distinct sub-levels) must always remain side by side, indeed, if they were to split, the assignment gadget would become unsolvable since a worker would have hit a wall. Given that the two workers must remain side by side, the workers in the clause gadget cannot hit a wall horizontally (otherwise they would overlap, making the level unsolvable). This means that using the `left` command is useless: in fact, it can only be used to go back to a variable and possibly change its truth value in *all* the clauses. Moreover, moving `up` and `down` multiple times when selecting a truth value does not lead to benefits either. Indeed, assuming that the user moves `left` or `right` only when none of the workers would hit a wall, then only the last `up` or `down` movement is used to decide the truth value of the variable in *all* the clauses. In other words, if the final game level is solvable, then it is also solvable by a program of the form $\mathtt{r}^4\sigma_1^4\mathtt{r}^4 \dots \sigma_n^4\mathtt{r}^4$ with $\sigma_i \in \{\mathtt{u}, \mathtt{d}\}$.

Now, observe that the workers of the clause gadget for $C = (x_a, x_b, x_c)$ arrive at an accepting cell if and only if they move `up` exactly once in correspondence with x_a, x_b, or x_c. Note that the vertical movement in correspondence with the other variables is ignored. Since the `up` movement is associated with the "True" truth value, the clause gadget behaves exactly like a clause of the Positive 1-in-3-SAT formula.

3.3 Conclusion of the proof

Proof of Theorem 1. Suppose the Positive 1-in-3-SAT instance is solvable, and let $\alpha_i \in \{\text{True}, \text{False}\}$ for $i \in \{1, \dots, n\}$ be the truth assignment to the variables that solves the instance. Then, letting $\sigma_i = \begin{cases} \mathtt{u}, & \text{if } \alpha_i = \text{True} \\ \mathtt{d}, & \text{if } \alpha_i = \text{False} \end{cases}$ the program $\mathtt{r}^4\sigma_1^4\mathtt{r}^4 \dots \sigma_n^4\mathtt{r}^4$ solves the corresponding `7BH-Essential` instance. Conversely, if the `7BH-Essential` instance is solvable, we can assume without loss of generality that it is solved by a program of the form $\mathtt{r}^4\sigma_1^4\mathtt{r}^4 \dots \sigma_n^4\mathtt{r}^4$ with $\sigma_i \in \{\mathtt{u}, \mathtt{d}\}$. Let $\alpha_i = \begin{cases} \text{True}, & \text{if } \sigma_i = \mathtt{u} \\ \text{False}, & \text{if } \sigma_i = \mathtt{d} \end{cases}$ then, as we argued, α satisfies all the 1-in-3-SAT clauses.

Moreover, note that each sub-level of the assignment gadget contains $O(n^2)$ cells, each sub-level of a clause gadget contains $O(n)$ cells, and the five sub-levels of the diagonal gadget have a constant number of cells. Therefore, the complete game level obtained by stacking all the sub-levels contains $O(n^2 + nm)$ cells and can be constructed in polynomial time. Our reduction is complete. ◀

4 PSPACE-Completeness of `7BH-Holes`

In this section, we prove Theorem 2. For a positive integer z, we use the notation $[z] = \{1, 2, \dots, z\}$. Let us start by showing that `7BH-Holes` can be solved in polynomial space.

▶ **Observation 4.** *`7BH-Holes` ∈ PSPACE*

Proof. Consider a game level $n \times m$ containing k workers. Since the workers are the only non-static elements of the level, each cell can be in at most one of two states: either it contains a worker, or it does not. Then, there are at most $\binom{nm}{k} \leq 2^{nm}$ possible configurations for the level. Then, consider the non-deterministic Turing Machine M that, given in input a level of `7BH-Holes`, maintains the current configuration of the level and a counter of the number of steps performed so far. If all the workers are standing on an accepting cell, then M accepts; if instead the counter exceeds 2^{nm}, then M rejects. In all other cases, M guesses non-deterministically the next step in $\{\mathtt{l}, \mathtt{r}, \mathtt{u}, \mathtt{d}, (\mathtt{ul}), (\mathtt{ur}), (\mathtt{dl}), (\mathtt{dr})\}$, updating the configuration and increasing the counter accordingly. It is evident that M solves `7BH-Holes` because if the level is solvable, there is a solution using at most 2^{nm} instructions. Moreover, the space required in any computation branch is $O(nm + \log(2^{nm})) = O(nm)$. Therefore, we showed that `7BH-Holes` ∈ NPSPACE, then, by Savitch's theorem [13], `7BH-Holes` ∈ PSPACE. ◀

Now it remains to show that `7BH-Holes` is PSPACE-Hard. We do so by exhibiting a polynomial-time reduction from the intersection non-emptiness problem for finite automata.

4.1 Intersection Non-Emptiness Problem

Recall that a deterministic finite automaton (DFA for short) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where Q is the set of states, Σ is the alphabet, $\delta : Q \times \Sigma \to Q$ is the transition function, $q_0 \in Q$ is the starting state and $F \subseteq Q$ is the set of accepting states. The language accepted by the DFA A, called $L(A)$, is the set of strings $x \in \Sigma^*$ such that, starting from q_0 and applying δ repeatedly, A ends up in an accepting state.

The following is a classic decision problem in automata theory and it was proved to be PSPACE-Complete by Kozen [11]:

▶ **Definition 5** (Intersection Non-Emptiness Problem). *Given in input a set of k DFAs $\{A_1, A_2, \ldots, A_k\}$, with $A_i = (Q_i, \Sigma, \delta_i, q_0^i, F_i)$ for $i \in [k]$, say if $\cap_{i=1}^k L(A_i) \neq \varnothing$*

To simplify the exposition, let us assume that all the DFAs have the same set of states Q and the same starting state q_0. This is without loss of generality because we can rename the states and add fictitious extra states, without changing the languages of the automata.

4.2 Reduction Overview

Consider an instance $I = \{A_1, A_2, \ldots, A_k\}$ of the Intersection Non-Emptiness Problem, with $A_i = (Q, \Sigma, \delta_i, q_0, F_i)$ for each $i \in [k]$. Without loss of generality, from now on we assume that $|Q| = n$, $Q = \{q_0, q_1, \ldots, q_{n-1}\}$, and $|\Sigma| = m$, $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_m\}$. Note that our reduction must be polynomial in k, n, and m.

We represent the computation of the DFAs using a string. Specifically, let $\Gamma = \Sigma \cup Q \cup \{\#\}$, where # is a new symbol, and consider a string of the following form:

$$R_1 \# R_2 \# \ldots R_r \# \in \Gamma^*$$

where:

$$\begin{aligned} R_j &= \sigma^{(j)} q^{(1,j)} q^{(2,j)} \ldots q^{(k,j)} && \text{for } j \in [r], \\ \sigma^{(j)} &\in \Sigma \text{ and } q^{(i,j)} \in Q && \text{for } i \in [k], j \in [r] \end{aligned}$$

Call $\mathcal{R}$ the set of all such strings. Intuitively, R_j contains the j-th input symbol and the state of the DFAs after processing the first $j-1$ input symbols.

We say that a string $R_1 \# R_2 \# \ldots R_r \# \in \mathcal{R}$ is *accepting* if it describes a valid accepting computation for all the automata, that is, if for all $i \in [k]$ the following holds:

- $q^{(i,1)} = q_0$
- $q^{(i,j+1)} = \delta_i(q^{(i,j)}, \sigma^{(j)})$, for all $j \in [r-1]$
- $q^{(i,r)} \in F_i$

The main idea of our reduction is to define (i) an encoding of the alphabet Γ with strings in $\{\mathtt{l}, \mathtt{r}, \mathtt{u}, \mathtt{d}\}^*$, and (ii) a game level $\mathcal{G}$ of `7BH-Holes`, such that, a program solves $\mathcal{G}$ if and only if it is an encoding of some accepting string in $\mathcal{R}$. This will be enough to conclude our reduction. Indeed, finding an accepting string in $\mathcal{R}$ is clearly equivalent to finding a string accepted by all the DFAs. More formally:

▶ **Observation 6.** *Given an instance $\{A_1, A_2, \ldots, A_k\}$ of the Intersection Non-Emptiness Problem, $\cap_{i=1}^k L(A_i) \neq \varnothing \iff \exists x \in \mathcal{R}$ accepting*

Proof. If $\sigma_1\sigma_2 \ldots \sigma_r \in \cap_{i=1}^k L(A_i)$, then the string $R_1 \# R_2 \# \ldots R_{r+1} \#$ where (i) $\sigma^{(i)} = \sigma_i$ for $i \in [r]$ and $\sigma^{(r+1)}$ is any symbol in Σ, and (ii) the states are set according to the computation, is an accepting string. Conversely, if $R_1 \# R_2 \# \ldots R_r \# \in \mathcal{R}$ is accepting, then $\sigma^{(1)}\sigma^{(2)} \ldots \sigma^{(r-1)} \in \cap_{i=1}^k L(A_i)$ (if $r = 1$, the empty string is accepted by all DFAs). ◀

4.3 The Encoding

We first associate an integer value to each element in $Q \cup \Sigma = \{q_0, \dots, q_{n-1}, \sigma_1, \dots, \sigma_m\}$, specifically, we define $\texttt{num} : Q \cup \Sigma \to \mathbb{N}$ as:

$$\texttt{num}(q_i) = 9 \cdot (k+2) \cdot (i+1) \qquad \forall i \in \{0, 1, \dots, n-1\}$$
$$\texttt{num}(\sigma_i) = 9 \cdot (k+2) \cdot (n+2) \cdot (i+1) \qquad \forall i \in [m]$$

Let us now define $\mathcal{C}$, the set of *clockwise* strings[5], as:

$$\mathcal{C} = \{\texttt{r}^{x_1}\texttt{d}^{x_2}\texttt{l}^{x_3}\texttt{u}^{x_4} \mid x_1, x_2, x_3, x_4 \geq 6\}$$

Our encoding will associate to each element of Γ a subset of clockwise strings, specifically, let $\texttt{enc} : \Gamma \to \mathcal{P}(\mathcal{C})$, where $\mathcal{P}(\mathcal{C})$ is the powerset of set $\mathcal{C}$, be defined as:

$$\texttt{enc}(\gamma) = \left\{\texttt{r}^{\texttt{num}(\gamma)}\texttt{d}^{\texttt{num}(\gamma)}\texttt{l}^{x_3}\texttt{u}^{x_4} \in \mathcal{C} \mid x_3, x_4 \geq 6\right\} \qquad \forall \gamma \in Q \cup \Sigma \qquad (1)$$

$$\texttt{enc}(\#) = \{\texttt{r}^{w_\#}\texttt{d}^{x_\#}\texttt{l}^{y_\#}\texttt{u}^{z_\#}\} \qquad \text{where:} \qquad (2)$$
$$w_\# = 9 \cdot (k+2) \cdot (n+2) \cdot (m+2)$$
$$x_\# = 18 \cdot (k+2) \cdot (n+2) \cdot (m+2)$$
$$y_\# = w_\# + 4k + 3$$
$$z_\# = x_\# + 3k + 3$$

Note that the sets are disjoint, therefore it is possible to decode a clockwise string. In particular, if $x \in \texttt{enc}(\gamma)$, with a slight abuse of notation, we say that $\texttt{enc}^{-1}(x) = \gamma$. Our goal now is to build the game level $\mathcal{G}$, such that, if $\gamma_1\gamma_2 \dots \gamma_t \in \mathcal{R}$ is accepting, then, the program $x_1 x_2 \dots x_t$ must solve the level $\mathcal{G}$, where $x_i \in \texttt{enc}(\gamma_i)$. Conversely, if a program solves $\mathcal{G}$, then it must be a concatenation of clockwise strings $x_1 x_2 \dots x_t$ such that they can be decoded into $\texttt{enc}^{-1}(x_i) = \gamma_i$, and $\gamma_1\gamma_2 \dots \gamma_t \in \mathcal{R}$ is accepting.

▶ Remark. At this stage, the numbers used in the encoding might appear arbitrary and obscure. We will point out where these numbers are used as we move forward in the proof.

4.4 The Game Level

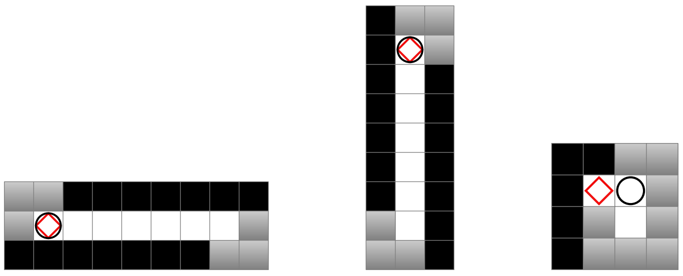
We build several independent sub-levels, each containing a single worker. In particular, we build $2k+4$ sub-levels: $\mathbf{S} = \{CW_1, CW_2, CW_3, \text{enforce}\#\} \cup \{M_i^{even}, M_i^{odd}\}_{i \in [k]}$. The final game level $\mathcal{G}$ is created by stacking the sub-levels together and isolating them via holes. Formally, $\mathcal{G} = \texttt{stack}(\mathbf{S})$.

The three sub-levels CW_1, CW_2, CW_3, reported in figure 5, are solved by all and only the programs that are concatenations of clockwise strings in $\mathcal{C}$ (note that CW_1 and CW_2 also prevent diagonal movements).

4.4.1 The Enforce# Sub-Level

Let us first introduce the *counter gadget*, in figure 6. This gadget is such that, if the worker is standing on the cell labeled with 0, then any clockwise string in $\mathcal{C}$ brings the worker to cell 1. Multiple counter gadgets can be concatenated together, and intuitively, these gadgets can be used to *skip* clockwise strings that we do not need to process.

[5] named after the fact that $\texttt{right}, \texttt{down}, \texttt{left}, \texttt{up}$ is a clockwise movement



(a) CW_1. **(b)** CW_2. **(c)** CW_3.

Figure 5 Sub-levels CW_1, CW_2, CW_3, together they ensure that the program is a concatenation of clockwise strings in $\mathcal{C}$. Note that CW_3 is needed to ensure that the very first movement is `right`.

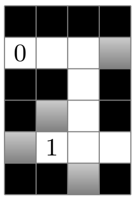


Figure 6 Counter gadget. A worker starting from cell 0 that processes any clockwise string in $\mathcal{C}$ ends up in cell 1.

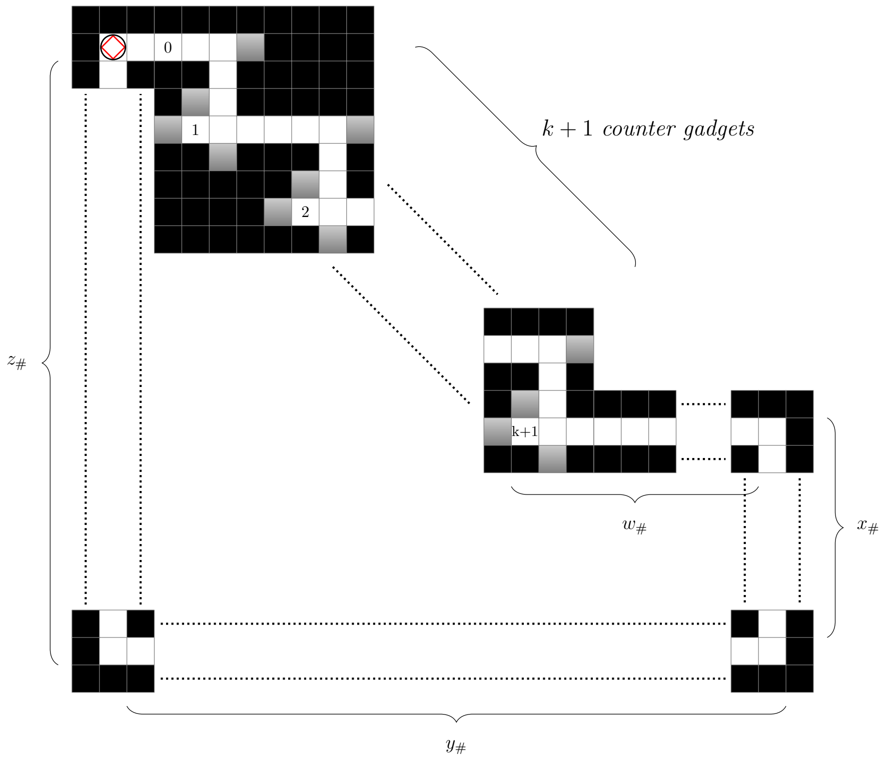


Figure 7 enforce# sub-level. The values $w_\#, x_\#, y_\#, z_\#$ are defined in Equation (2). This gadget ensures that accepting programs must be a concatenation of a multiple of $k+2$ clockwise strings each ending with the encoding of #.

The enforce# sub-level, in figure 7, skips the first $k+1$ clockwise strings using $k+1$ counter gadgets, then, it forces the next clockwise string to be $\mathtt{r}^{w_\#}\mathtt{d}^{x_\#}\mathtt{l}^{y_\#}\mathtt{u}^{z_\#}$, which is the only encoding of $\# \in \Gamma$, as defined in Equation (2). Indeed, if the $(k+2)$th clockwise string was not the encoding of #, the worker would end up in a hole. After processing the whole encoding of #, the worker will be in its starting position again (which is also the only accepting one), ready to possibly process more clockwise strings.

Therefore, to be more formal, the enforce# sub-level together with $\{CW_1, CW_2, CW_3\}$, ensures that if π is a solving program, then it must be of the form $\overline{R_1}\overline{\#}\overline{R_2}\overline{\#}\dots\overline{R_r}\overline{\#}$, where $\overline{\#}$ is the only encoding of #, and each $\overline{R_j}$ is a concatenation of $k+1$ clockwise strings. Note also that all the programs of such form solve these four sub-levels.

▶ Remark. The enforce# sub-level can be built if it holds (i) $y_\# = 2 + 4(k+1) + w_\# - 3 = w_\# + 4k + 3$, and (ii) $z_\# = 3(k+1) + x_\#$. Note that both these relations are satisfied by our encoding, as reported in Equation (2).

4.4.2 The Automata Sub-Levels

For each DFA A_i, $i \in [k]$, we introduce two sub-levels: M_i^{odd} and M_i^{even}. Consider a program of the form $\overline{R_1}\overline{\#}\dots\overline{R_r}\overline{\#}$, where $\overline{\#} \in \mathtt{enc}(\#)$, and each $\overline{R_j}$ is a concatenation of $k+1$ clockwise strings. Intuitively, M_i^{odd} will ensure, for each odd j, that $\overline{R_{j+1}}$ follows from $\overline{R_j}$ according to the computation of A_i. M_i^{even} will guarantee the same, but for even j's. These sub-levels will also guarantee that the last state is an accepting one. Therefore, adding all the sub-levels $\{M_i^{odd}, M_i^{even}\}_{i\in[k]}$ will ensure that an accepting program must describe an accepting computation for all the DFAs.

Before showing the construction of M_i^{odd} and M_i^{even}, we introduce three new gadgets.

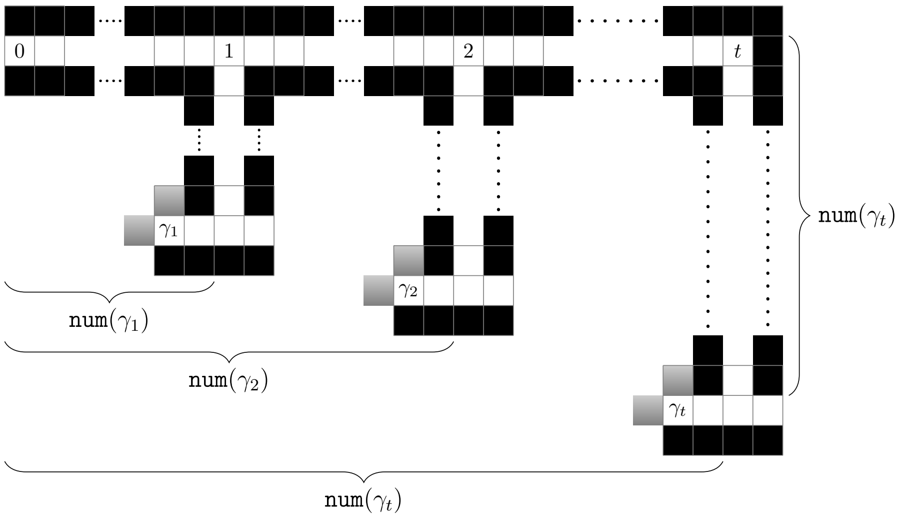


Figure 8 The *S-selector gadget* for $S = \{\gamma_1, \gamma_2, \dots, \gamma_t\} \subseteq Q \cup \Sigma$. A clockwise string reaches the cell labeled with γ_ℓ if it moves `right` and `down` exactly $\mathtt{num}(\gamma_\ell)$ times.

The *S-selector gadget*, in figure 8, is parametrized by a set $S = \{\gamma_1, \gamma_2, \dots, \gamma_t\} \subseteq Q \cup \Sigma$ such that $\mathtt{num}(\gamma_j) < \mathtt{num}(\gamma_{j+1})$. If the worker is standing on the cell labeled with 0 of the selector gadget, then the next clockwise string must be any encoding of any element $\gamma_\ell \in S$, which will lead the worker to the cell labeled with γ_ℓ. Indeed, from our encoding in Equation (1), to verify that a clockwise string $\mathtt{r}^{x_1}\mathtt{d}^{x_2}\mathtt{l}^{x_3}\mathtt{u}^{x_4} \in \mathcal{C}$ is an encoding of a certain element

$\gamma_\ell \in Q \cup \Sigma$, it suffices to check that $x_1 = x_2 = \texttt{num}(\gamma_\ell)$, which is what the gadget does. It is also easy to check that the worker will fall into a hole if the clockwise string is not an encoding of any element of S. This gadget will be useful for choosing the next input symbol and performing different checks depending on the current state of the automaton.

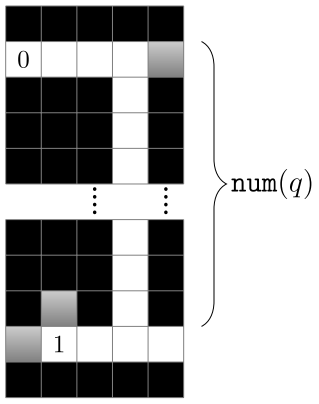


Figure 9 The *q-forcer gadget* for $q \in Q$. Starting from cell 0, the worker reaches cell 1 with a clockwise string if and only if the number of $\texttt{d}$ symbols is equal to $\texttt{num}(q)$.

For a state $q \in Q$, the *q-forcer gadget*, in figure 9, forces the next clockwise string to have a number of `down` steps equal $\texttt{num}(q)$. Therefore, if we know that $c \in \mathcal{C}$ is an encoding of some state, then, by using the q-forcer gadget we impose the constraint that c must be an encoding of $q \in Q$. This gadget will be useful to force the string to respect the transition function.

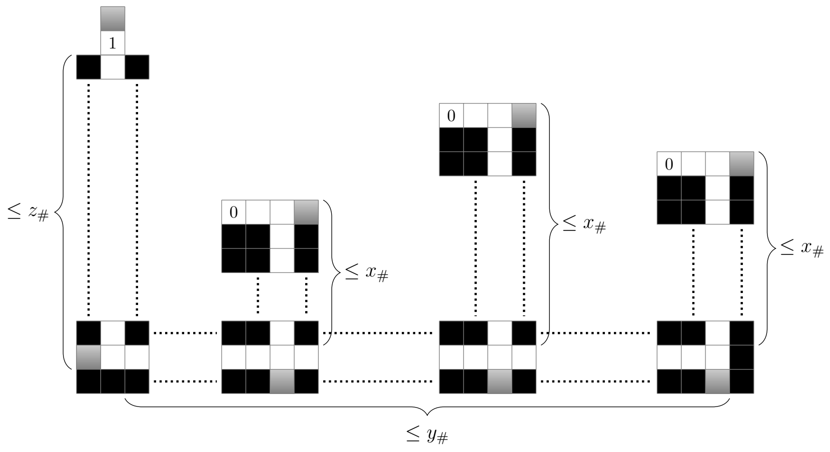


Figure 10 The *go-back gadget*. Starting from any cell 0, the clockwise string $\texttt{r}^{w_\#}\texttt{d}^{x_\#}\texttt{l}^{y_\#}\texttt{u}^{z_\#}$, as described in Equation (2), brings the worker back to the cell 1. Note that more cells 0 can be added as needed, provided that the highlighted constraints are respected.

The last gadget we need is the *go-back gadget*, in figure 10. Suppose the worker is in one of the cells labeled with 0, then, by processing the only encoding of $\# \in \Gamma$, it will go back to cell 1. This gadget will be useful to "go back" to a selector gadget and analyze the next chunk of the program.

Note that these three gadgets and the counter gadget can be concatenated together. The only precaution to take is that when concatenating a selector gadget to a counter gadget, the cell 0 of the selector (figure 8) must coincide with the cell 1 of the counter (figure 6). Similarly, when concatenating a *go-back gadget* to a selector gadget, the cell 1 of the *go-back* must coincide with the cell 0 of the selector.

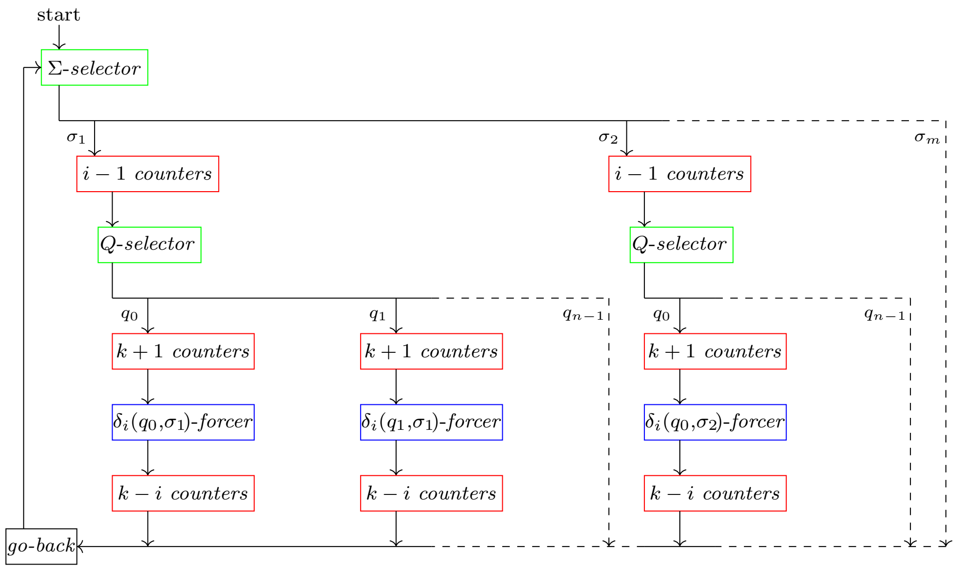


Figure 11 M_i^{odd} sub-level associated to the DFA A_i, $i \in [k]$. The starting position of the single worker is at the beginning of the Σ-selector (i.e., the cell 0 of the Σ-selector, taking figure 8 as a reference). The first cell of the Σ-selector is also an accepting cell. Moreover, each branch of the Q-selectors corresponding to an accepting state $q_{acc} \in F_i$, contains an accepting cell after the first $k + 1 - i$ counters (i.e., the accepting cell is the cell labeled with 1, taking figure 6 as a reference, in the $(k + 1 - i)$th of the $k + 1$ counters).

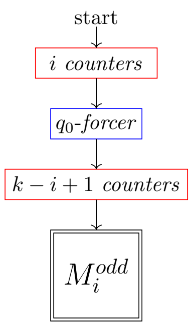


Figure 12 M_i^{even} sub-level associated to the DFA A_i, $i \in [k]$. The starting position of the single worker is at the beginning of the first counter. The accepting cells are the same described for the M_i^{odd} sub-level.

Fixed $i \in [k]$, we report, in figure 11, the sub-level M_i^{odd}, and, in figure 12 the sub-level M_i^{even}. Both are represented in a schematic way. Both sub-levels use the go-back gadget, which requires processing exactly the encoding of #: this is guaranteed by the enforce# sub-level.

We now argue that the sub-levels $\{CW_1, CW_2, CW_3, \text{enforce}\#, M_i^{odd}, M_i^{even}\}$ are solved by all and only the programs describing a valid accepting computation of DFA A_i.

We know, from $\{CW_1, CW_2, CW_3, \text{enforce}\#\}$, that a solving program π must of the form: $\overline{R_1}\overline{\#}\overline{R_2}\overline{\#}\dots\overline{R_r}\overline{\#}$, where $\overline{\#} \in \texttt{enc}(\#)$, and $\overline{R_j} = c_1^j c_2^j \dots c_{k+1}^j$ is a concatenation of $k+1$ clockwise strings. Let us first show that π can always be decoded:

- The Σ-selector of M_i^{odd} (resp. M_i^{even}) ensures that the first symbol of each $\overline{R_j}$, for odd j (resp. even j), is the encoding of a symbol in Σ. Therefore, for all $j \in [r]$, it must be $c_1^j \in \texttt{enc}(\sigma)$, for some $\sigma \in \Sigma$.
- Similarly, the Q-selectors of M_i^{odd} (resp. M_i^{even}) ensure that the $(i+1)$th symbol of each $\overline{R_j}$, for odd j (resp. even j), is the encoding of a state. Therefore, for all $j \in [r]$, it must be $c_{i+1}^j \in \texttt{enc}(q)$, for some $q \in Q$.

Moreover, π describes a valid computation:

- the first forcer of M_i^{even} ensures that the computation starts from the starting state q_0: $c_{i+1}^1 \in \texttt{enc}(q_0)$
- Suppose that the computation is valid up to $\overline{R_j}$, that is, the state $q = \texttt{enc}^{-1}(c_{i+1}^j)$ is correctly reached from q_0. Let us assume that j is odd. Then, M_i^{odd} will follow the branch σ in the Σ-selector, for some $\sigma \in \Sigma$, and the branch q in the Q-selector. Then, the $\delta_i(q,\sigma)$-forcer in M_i^{odd} forces c_{i+1}^{j+1} to be in $\texttt{enc}(\delta_i(q,\sigma))$, therefore, the state reached in $\overline{R_{j+1}}$ is correct too. If instead j is even, M_i^{even} ensures that the state reached in $\overline{R_{j+1}}$ is correct.

It is left to show that the computation described by π is accepting for A_i. Suppose r is odd. Then, at the end of the computation, the worker of M_i^{even} will be at the beginning of the Σ-selector (which is an accepting cell), and the worker of M_i^{odd}, which is "shifted forward" by $k+2$ clockwise strings, will be at the end of the $(k+1-i)$th counter right after the Q-selector, but note that there is an accepting cell in such position only if the worker is in a branch corresponding to a state $q_{acc} \in F_i$, therefore $c_{i+1}^r \in \texttt{enc}(q_{acc})$ and the computation is accepting. If r is even, the situation is analogous: the worker of M_i^{odd} is at the beginning of the Σ-selector, but the one of M_i^{even} is at the end of the $(k+1-i)$th counter after a Q-selector, and therefore it must be in an accepting branch.

Moreover, one can easily see that any encoding of an accepting computation solves all the sub-levels. Indeed, the Σ-selector allows the user to choose the input string, and then, since the computation is accepting, one of M_i^{even} and M_i^{odd} will end up in the $(k+1-i)$th counter of an accepting branch and the other will stay at the beginning of the Σ-selector. Therefore, all the sub-levels would be solved.

▶ Remark. Our encoding allows the construction of M_i^{odd} and M_i^{even}. First, from our encoding, we have that for $j \in \{0, 1, \dots, n-2\}$, $\texttt{num}(q_{j+1}) - \texttt{num}(q_j) = 9(k+2)$, while the space actually needed between two branches of the Q-selector is: $4(k+1)+5+4(k-i)+4+6 < 8(k+1)+11 \leq 9(k+2)$ where the last 6 takes into account the overhead of the selector. Similarly, for $j \in [m-1]$, $\texttt{num}(\sigma_{j+1}) - \texttt{num}(\sigma_j) = 9(k+2)(n+2)$, while the space required between two branches of the Σ-selector is at most $4(i-1) + 9(k+2)(n+1) + 6 < 9(k+2)(n+2)$. To conclude, we need only to check that the constraints for the go-back gadget are satisfied. The width of the whole M_i^{odd} sub-level is at most $9(k+2)(n+2)(m+2) = w_\# < y_\#$. The height of the sub-level is instead at most $\texttt{num}(\sigma_m)+4(i-1)+\texttt{num}(q_{n-1})+4(2k+1-i)+\texttt{num}(q_{n-1})+3 \leq 18(k+2)(n+2)(m+1)+8k < 18(k+2)(n+2)(m+2) = x_\# < z_\#$. Therefore all the gadgets can be concatenated together.

4.5 Conclusion of the proof

Putting all the pieces together, we can prove the theorem.

Proof of Theorem 2. The problem is in PSPACE by Observation 4. Consider the following reduction from the Intersection Non-Emptiness Problem: an instance $I = \{A_1, A_2, \dots, A_k\}$ is associated with the level $\mathcal{G} = \texttt{stack}(\{CW_1, CW2, CW_3, \text{enforce}\#\} \cup \{M_i^{odd}, M_i^{even}\}_{i \in [k]})$.

Suppose I can be solved, then, by Observation 6, there exists an accepting string $x_1 x_2 \dots x_t \in \mathcal{R}$. Consider any encoding $\pi = y_1 y_2 \dots y_t$ such that $y_j \in \texttt{enc}(x_j)$. Such program can be rewritten in the form $\overline{R_1}\overline{\#} \dots \overline{R_r}\overline{\#}$. Therefore, as we argued, π solves $\{CW_1, CW_2, CW_3, \text{enforce}\#\}$, and, given that I is solved, for each $i \in [k]$, π describes an accepting computation for A_i, then, it also solves M_i^{odd} and M_i^{even}. Therefore, $\mathcal{G}$ is solved: all its workers will be standing on an accepting cell at the end of the program.

Suppose now that there exists a program $\pi \in \{\texttt{l}, \texttt{r}, \texttt{u}, \texttt{d}\}^*$ solving $\mathcal{G}$. As we argued, such a program can be decoded into a string $R_1\# \dots R_r\# \in \mathcal{R}$. Given that, for each $i \in [k]$, M_i^{odd} and M_i^{even} are solved, it means that the string represents an accepting computation for A_i. That is, letting $R_j = \sigma^{(j)} q^{(1,j)} \dots q^{(k,j)}$, it holds: (i) $q^{(i,1)} = q_0$, (ii) $q^{(i,j+1)} = \delta_i(q^{(i,j)}, \sigma^{(j)})$ for $j \in [r-1]$, and (iii) $q^{(i,r)} \in F_i$. Therefore, the string in $\mathcal{R}$ is accepting. Using Observation 6, it follows that I is solvable.

Finally, observe that the number of cells in each sub-level is at most $O(y_\# \cdot z_\#) = O(k^2 n^2 m^2)$, and since there are $2k + 4$ sub-levels, $\mathcal{G}$ has at most $O(k^3 n^2 m^2)$ cells, therefore the reduction can be carried out in polynomial time. ◀

5 Conclusions

We analyzed the computational complexity of the video game "7 Billion Humans". The game involves controlling multiple workers simultaneously to direct them to some destination cells. When each cell is either empty or contains a wall, the problem of deciding if a level is solvable is NP-Hard, while adding holes makes the problem PSPACE-Complete. We also observed that the simple structure of our reductions entails hardness results for the problem of simultaneous maze solving and for the intersection non-emptiness problem.

While `7BH-Essential`, the problem where levels only contain walls and empty cells, is NP-Hard and clearly in PSPACE, it is not known whether it lies in NP. We leave this as an interesting open problem.

References

1 Matteo Almanza, Stefano Leucci, and Alessandro Panconesi. Trainyard is np-hard. *Theoretical Computer Science*, 2018. FUN with Algorithms.

2 Matteo Almanza, Stefano Leucci, and Alessandro Panconesi. Tracks from hell — when finding a proof may be easier than checking it. *Theoretical Computer Science*, 2020. FUN with Algorithms.

3 Greg Aloupis, Erik D. Demaine, Alan Guo, and Giovanni Viglietta. Classic nintendo games are (computationally) hard. *Theoretical Computer Science*, 2015. FUN with Algorithms.

4 Emmanuel Arrighi, Henning Fernau, Stefan Hoffmann, Markus Holzer, Ismaël Jecker, Mateus de Oliveira Oliveira, and Petra Wolf. On the Complexity of Intersection Non-emptiness for Star-Free Language Classes. In *41st IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2021)*, 2021.

5 Erik D. Demaine, Joshua Lockhart, and Jayson Lynch. The computational complexity of portal and other 3d video games. *FUN with Algorithms*, 2018.

6 Erik D. Demaine, Giovanni Viglietta, and Aaron Williams. Super mario bros. is harder/easier than we thought. *FUN with Algorithms*, 2016.

7 Stefan Funke, André Nusser, and Sabine Storandt. The simultaneous maze solving problem. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

8 Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

9 Luciano Gualà, Stefano Leucci, and Emanuele Natale. Bejeweled, candy crush and other match-three games are (np-)hard. *2014 IEEE Conference on Computational Intelligence and Games*, 2014.

10 Graham Kendall, Andrew Parkes, and Kristian Spoerer. A survey of np-complete puzzles. *ICGA Journal*, 2008.

11 Dexter Kozen. Lower bounds for natural proof systems. In *18th Annual Symposium on Foundations of Computer Science (FOCS)*, 1977.

12 Klaus-Jörn Lange and Peter Rossmanith. The emptiness problem for intersections of regular languages. In *Mathematical Foundations of Computer Science*, 1992.

13 Walter J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2), 1970.

14 Giovanni Viglietta. Gaming is a hard job, but someone has to do it! In *Fun with Algorithms*, 2012.

15 Michael Wehar. Hardness results for intersection non-emptiness. In *Automata, Languages, and Programming (ICALP)*, 2014.

16 Commonsense review. `https://www.commonsense.org/education/reviews/7-billion-humans`. Accessed: 2024-02-26.

17 Hackr review. `https://hackr.io/blog/coding-games`. Accessed: 2024-02-26.

18 Hubspot review. `https://blog.hubspot.com/website/best-coding-games`. Accessed: 2024-02-26.