Recursed Is Not Recursive: A Jarring Result

Erik D. Demaine
Computer Science and Artificial Intelligence Laboratory, MIT, Cambridge, MA, USA
http://erikdemaine.org/
edemaine@mit.edu

Justin Kopinsky
Work done while at MIT, Cambridge, MA, USA
jkopinsky@gmail.com

Jayson Lynch
Computer Science and Artificial Intelligence Laboratory, MIT, Cambridge, MA, USA
jaysonl@mit.edu

Abstract

Recursed is a 2D puzzle platform video game featuring "treasure chests" that, when jumped into, instantiate a room that can later be exited (similar to function calls), optionally generating a "jar" that returns back to that room (similar to continuations). We prove that Recursed is RE-complete and thus undecidable (not recursive) by a reduction from the Post Correspondence Problem. Our reduction is "practical": the reduction from PCP results in fully playable levels that abide by all constraints governing levels (including the 15×20 room size) designed for the main game. Our reduction is also "efficient": a Turing machine can be simulated by a Recursed level whose size is linear in the encoding size of the Turing machine and whose solution length is polynomial in the running time of the Turing machine.

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

Keywords and phrases Computational Complexity, Undecidable, Video Games

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2020.50

Related Version A full version of this paper is available at https://arxiv.org/abs/2002.05131 [10].

Acknowledgements This work was initiated during the 33rd Bellairs Winter Workshop on Computational Geometry, co-organized by Erik Demaine and Godfried Toussaint in March 2018 in Holetown, Barbados. We thank the other participants – in particular, Robert Hearn – for related discussions and providing an inspiring atmosphere. We thank Edison Y. He for his helpful comments on earlier drafts of this paper. Figures were generated using SVG Tiler [7].

1 Introduction

Recursed[1] [15] is an indie puzzle platform video game by lone developer Portponky. The game's main feature is having rooms contained within treasure chests, often recursively, inspired by functional programming; see Section 2 for details.

In this paper, we show that deciding whether a given Recursed level can be solved is RE-complete and thus undecidable (not recursive).[2] We thus positively settle a player's claim that Recursed is NP-hard [5] and another player's conjecture that it is undecidable [11]. Our proof is by a reduction from the Post Correspondence Problem (PCP); refer to Section 3

[1] All products, company names, brand names, trademarks, and sprites are properties of their respective owners. Sprites are used here under Fair Use for the educational purpose of illustrating mathematical theorems.

[2] A brief recap on terminology: RE (Recursively Enumerable) is the class of decision problems whose "yes" instances are accepted by a Turing machine in finite time, but whose "no" instances may be indicated by the machine running for infinite time, while R (Recursive or Decidable) is the class of decision problems whose "yes" and "no" instances are accepted and rejected, respectively, by a Turing machine in finite time. It is known that RE $\subsetneq$ R; for example, the Halting Problem is in the difference RE $\setminus$ R.

for a definition. We use the properties of PCP that the constraints are locally checkable and that the resolution of choices proceeds in only one direction (adding more words/dominoes to the string).

RE-completeness of a video game requires some source of arbitrarily unbounded state in the game. The unbounded state we use in Recursed stems only from the player's ability to generate instances of rooms arbitrarily deeply through normal use of the game's recursive chest mechanics (and by extension, its jar mechanics); see Section 2 for details. Each of the finitely many rooms resulting from our reduction has constant size – even fitting within the 15×20 size of standard Recursed rooms – and contains only a constant number of objects and a constant amount of state. Indeed, the Recursed levels generated by our reduction are "practical": they could, in principle, be solved by a human, provided they knew which PCP dominoes to place at each placement step. Using the custom level feature of Recursed, we have built a fully playable custom level demonstrating the reduction applied to a simple 2-domino PCP instance, which is available for download [9].

Our reduction is also ***efficient***, meaning that it efficiently represents the execution of a Turing machine. The Recursed level size is linear in the number k of dominoes in the PCP instance, and the Recursed solution length is $O(L \log k)$ where L is the number of symbols in a solution to the PCP instance. Using the standard reduction from the Halting Problem to PCP [18], the Recursed level size is linear in the encoding size k of the Turing machine, and the Recursed solution length is $O(TS \log k) = O(T^2 \log k)$ where T is the running time and S is the space used by the Turing machine. As a consequence, deciding whether a Recursed level can be solved in a polynomial number of steps is NP-complete, and deciding whether a Recursed level can be solved in an exponential number of steps is NEXPTIME-complete.

Related Work. The first RE-completeness/undecidability result for a video game was for another indie puzzle game, Braid [12], designed by Jonathan Blow. (The computational complexity of Blow's other puzzle game, The Witness, has also been studied, with NP-, Σ_2-, and PSPACE-completeness results for various aspects of the game [1].) To our knowledge, our result is the second RE-completeness/undecidability result for a (real-world) single-player video game.

The Braid reduction [12] produces a Braid level of finite size. The unbounded state it exploits comes from the game's ability to generate arbitrarily unbounded quantities of enemies and pack them into the same location, allowing the level to increment a counter arbitrarily high. Enemies prevent the player from getting to a location, allowing the player to detect when the counter is zero. In this way, the Braid reduction simulates a counter machine. Because the reduction from Turing machine to counter machine [14] requires an exponential slowdown, the Braid reduction is not efficient: the resulting solution length is exponential in the running time of the Turing machine. Also, because the items in Recursed all help rather than hinder the player's mobility, this type of approach cannot work for Recursed.

For two-player games, there is one undecidability result we are aware of: Magic: The Gathering is RE-hard/undecidable even for two players [4], via an efficient Turing machine simulation. In fact, the players' moves are all forced, so this result is arguably about a zero-player simulation (but only the two-player game is "real-world"). An earlier RE-hardness/undecidability proof [3] simulated a counter machine, and thus was inefficient; it also required more players and a small tweak to the game rules.

Team multiplayer games are often RE-complete/undecidable even when the game's state is finite; the source of unboundedness is the hypothetical game strategies built in the players' heads [13]. Recently, this technique has been applied to prove RE-completeness/undecidability

of real-world team video games, including Team Fortress 2, Super Smash Brothers: Brawl, and Mario Kart [6]. These reductions are naturally very different from Recursed, given the different source of unboundedness.

Roadmap. Section 2 gives an overview of the mechanics of Recursed relevant to our construction. Section 3 presents a sketch of our reduction construction. For full details, please refer to the full paper [10]. Section 4 describes some open questions and conjectures regarding the complexity of subsets of Recursed.

2 Game Rules

This section covers the rules of Recursed insofar as they are needed for our construction in Section 3. For simplicity, we omit those objects and notions which we will not use.[3] See [16] for a video illustration of some core mechanics, including blocks, keys, doors, chests, and jars.

2.1 Basic Player Actions

We will call the player character Rico. By default, Rico can ***run*** horizontally and ***jump*** or ***fall*** vertically. Rico can jump up to a surface at most 3 tiles higher than where they jumped from, but can fall arbitrarily far with no penalty. Rico can ***pick up*** objects they are standing next to (see below for an enumeration). Rico can only carry one object at a time, and cannot pick up further objects until releasing the one held. While holding an object, Rico can ***drop*** it, causing it to fall, or ***throw*** it. Thrown objects travel in a perfectly vertical or horizontal trajectory until hitting a solid tile (or reaching the apex, if thrown upwards), at which point they fall until landing on a floor tile, or falling off the bottom of the screen. Note that other objects do not impede the horizontal trajectory of a thrown object, but, when falling, objects can land on blocks. If Rico is carrying an object, their jump height is lowered to at most 2 blocks.

Typically, walls, floors, and ceilings are comprised of ***tiles***, which are immovable and impassable, by Rico or any object. At any given time, Rico will be situated in a room of size at most 15×20 (though sometimes smaller), which is what is shown to the player. Typically, rooms will have borders consisting of solid tiles (counting towards the size). It is possible for some walls, floor, or ceiling to be missing. If Rico falls off the edge, they bounce back up a few blocks, but we will not have any missing floors in our construction.

A ***level*** is comprised of a collection of rooms (see Section 2.3 for more details). Rico's goal is to reach a purple ***crystal***, of which exactly one exists in some room of each level.

There is one additional special environmental feature called a ***ledge*** (see Figure 1). Ledges are always oriented horizontally. Rico can jump ***upwards*** through a ledge, but cannot by any means traverse back ***downwards*** through a ledge. Thrown or dropped objects ignore ledges entirely in both directions.

2.2 Basic Objects

We depict all of the objects necessary for our construction in Figure 1. A description of each follows.

[3] Inexhaustively including water, acid, Ooblecks, cauldrons, paradoxes, glitches, and cloud walls.

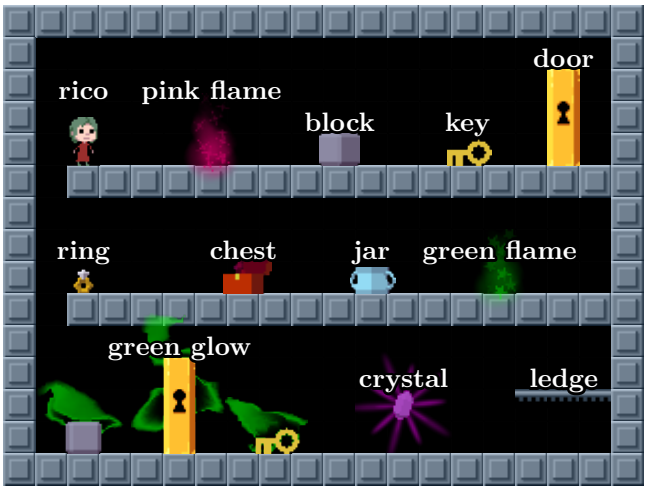


Figure 1 A toy level containing a copy of each object used in our construction.

Blocks. The primary use of blocks is for Rico to stand on and be able to jump higher. In particular, if Rico is standing on a block, they can jump to a height of 4 above the ground (3 if carrying another object), which would be otherwise impossible. Furthermore, blocks can be stacked arbitrarily high, and Rico can "climb" stacks of blocks, so with k blocks, Rico can reach a height of $k + 3$.

Keys and doors. Keys are carryable objects which open ***doors***. Doors are static objects in the level which occupy a space 3 tiles high by 1 tile wide, generally preventing traversal from one side to the other. A key can be carried directly to a door, or thrown at it. In either case, the key and the door both disappear, allowing Rico to traverse the space previously occupied by the door. There is only one type of key and one type of door, so any key in the game can open any door.

Rings. The only direct use of rings in-game is to trigger pre-scripted dialog when thrown against a wall. Therefore, they provide no particular use toward solving a level. Nevertheless, we will make use of rings in our construction as a generic object which specifically does nothing *except* lower Rico's jump height to 2 tiles when held. See Section 3.2.2.

2.3 Chests

Chests are the primary "gimmick" of Recursed. They were designed to emulate function calls (in the programming sense) to some degree. In Recursed, chests do not contain objects, but rather entire rooms. Rico can jump ***into*** chests, thereby entering the room contained therein. The room contained in a chest is an immutable property of the chest itself – a particular chest will always contain a particular room, according to the specification of that chest. Different chests can contain the same room.

When Rico enters a room via a chest, they appear at a pre-specified entry point[4], along with a ***pink flame***, which we refer to interchangeably as an ***exit***. The room will be generated freshly from its specification each time Rico enters a chest containing it *regardless of whether*

[4] The entry point is a property of the room itself; Rico will appear at the same entry point regardless of which chest was used to enter the room.

Rico has previously visited and/or interacted with objects in the room. This follows the function call intuition of each invocation entering the function at the beginning with no local state.

While in a room contained in a chest (which is most of the time), Rico can freely interact with any objects present *including other chests*. Every room except for the initial room Rico begins the level in will necessary have a pink flame exit. If Rico returns to the exit of a room, they can choose to leave by interacting with the pink flame. In doing so, they will hop back out of the chest they initially came in, thus re-entering the "parent" room *in the same state that it was when Rico jumped in the chest.* Note the asymmetry between entering and exiting chests. Again, this emulates the function call behavior of saving the local state of the parent function when returning from a child function. Because any future entries to the chest room will re-generate the room anew, any state that room had when Rico leaves is entirely forgotten.

Rico can of course recursively enter chests (hence the name), thereby saving a "call history" in a stack-like fashion. Rico can even enter a room via a chest contained in that same room, reminiscent of a recursive function calling itself.

One final key property of chests is that when Rico jumps into a chest, or leaves via the pink flame, they can do so while carrying at most one object. Thus, provided that Rico can manage to get access to it, Rico can bring a block, a key, or another chest with them into or out of a chest. To demonstrate the impact of this ability, observe that on the one hand, if Rico carries, say, a block into a chest and then subsequently leaves the chest empty-handed, that block is *lost forever*. On the other hand, if Rico enters a chest empty-handed, but manages to leave while carrying a block, the parent room now has a block that in effect did not previously exist. In particular, Rico can repeat the same sequence of actions any number of times to produce an *unbounded* number of blocks in the parent room.

2.4 Green Glow

Some objects in the game have a ***green glow*** (see Figure 1). These objects violate the "function call" rules of chests described above, in that the state of a green glowing object is saved no matter when or where it is interacted with. The simplest example is a green glowing door, since it cannot be moved, but only open. If a green glowing door in a room is opened by any key, it will *always* be open when Rico revisits that room, even if doing so by entering a chest and thus regenerating (the nonglowing parts of) the room.

Movable objects, including blocks, keys, and chests can also glow green. In this case, if the object moves around the room it begins the level in, then whenever Rico revisits that room the location of the object will be remembered. This is the only property we will make use of in the construction, but we note for posterity that green glowing objects can be moved between rooms and this will be remembered as well. Green glowing chests have even more interesting properties, but we encourage the reader to play the game and discover those for themselves!

2.5 Jars

Jars are similar to chests in that they contain rooms, but unlike chests, they are designed to emulate ***continuations*** (in the functional programming sense), rather than function calls. Jars can never be present in the initial state of a level. Rather, some rooms (other than the initial starting room) will have a ***green flame*** exit in addition to the standard *pink flame* exit (not to be confused with green glow above). The green flame can be located anywhere in the room and is independent of Rico's initial point of entry. There can even be more than

one (though not in our construction). If Rico exits a room via a green flame exit, they will hop back out of the containing chest, just like by the pink flame, except they will now be carrying a newly created ***jar***. Note that Rico cannot be carrying any object when leaving via green flame exits in order to have space to carry the jar.

Rico can carry the jar around just like any other object. If, subsequently, Rico enters a jar, they will re-enter the room containing the green flame the jar was created with, at the location of the green flame, with the room in the *same* state that it was in when the jar was created, *except* that the green flame itself is now gone, so no further jars can be created from the same place. Thus, any doors previously opened or objects previously moved or placed will be just where they were when Rico used the green flame. Importantly, when Rico later exits a room after entering it from a jar, they will reappear in the parent room just as if they had used a chest, and may even be carrying an object, but the jar will be destroyed. Thus, any particular jar can only be entered once.

3 Main Result

▶ **Theorem 3.1** (Recursed is RE-complete)**.** *It is RE-complete to decide whether a player can reach the crystal in a given level.*

Containment is straightforward: the game can obviously be simulated, given an initial state and sequence of player inputs. Thus, with a recursively enumerable Turing machine, one can enumerate every input string frame-by-frame and check whether any such string solves the level.

The hardness reduction is from the Post Correspondence Problem (PCP). Originally shown undecidable by Post in [17], we follow Sipser's description of the problem [18]. Given a set of dominoes $D_1, \ldots, D_k$ each with a string $A_i = a_{i1}a_{i2}\ldots a_{is_i}$ on the top half and a string $B_i = b_{i1}\ldots b_{ir_i}$ on the bottom half, denoted $D_i = \langle A_i \mid B_i \rangle$. We are tasked with laying such dominoes next to each other (copying dominoes as much as necessary) such that the concatenation of the top halves equals the concatenation of the bottom halves. We will enforce that the first domino must be D_0 which will simplify the initial part of the construction. (Forcing the first domino to be of a specified type clearly does not make the problem decidable, since if it did there is a trivial nondeterministic decision algorithm which guesses the first domino and then calls the hypothesized decision oracle).

We will implement a (nondeterministic) algorithm to solve PCP in Recursed. The algorithm is as follows:

1. Nondeterministically choose a domino $D_i = \langle A_i \mid B_i \rangle$ to add to the solution, or stop and skip to 4.
2. Push A_i onto stack S_A and B_i onto stack S_B.
3. Return to 1.
4. Pop S_A and S_B and check whether the popped symbols are equal; REJECT if not.
5. Repeat 4 until one stack empties, then check if the other stack is also empty; if yes, ACCEPT, else REJECT.

3.1 High-Level Overview

Rico initially spawns in a room next to a block and a chests, with another chest and a locked door past a ledge, shown in Figure 2. On the other side of the locked door is the goal crystal, but it is 6 tiles above the ground, one tile too high for Rico to jump to reach unaided[5]. The

[5] Rico can reach a jump height of 3 tiles and is 2 tiles tall, and so can reach a crystal 5 blocks high.

chest next to the door is a Global-Lock chest, which, once unlocked, will provide the key to this very door. All Rico has to do is open the Global-Lock gadget and get the block to the other side of the locked door! Of course, it will not be so easy...

The reduction is demonstrated with a fully playable level [9] for a PCP instance with $D_0 = \langle 01 \mid 0 \rangle$, $D_1 = \langle 0 \mid 10 \rangle$, whose (shortest) solution is of course $D_0 D_1$.

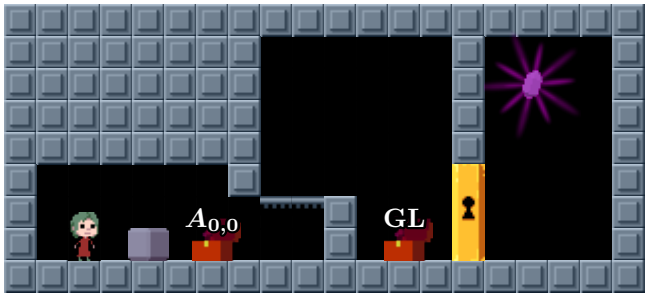


Figure 2 The initial room.

The high level structure of the construction is as follows: We will store S_A in the "call history" of chests Rico has jumped into and we will store S_B in a chain of jars. For each symbol $a_{ij} \in S_A$, there will be one corresponding room in our call history. For each symbol $b_{ij} \in S_B$, there will be one corresponding room in our chain of jars. Rico will need to carry the top level jar around pretty much all the time, unless changing the state of another gadget. The very last jar at the bottom of the chain representing S_B will contain a single block[6] which, if retrieved in the crystal room, Rico can use to jump on and reach the crystal.

The intended solution path which Rico must follow is comprised of two phases: a "Pushing" phase, and a "Checking" phase. During the pushing phase, Rico will push symbols to S_A via the explicit chest stack, and to S_B by building the chain of jars (i.e., the outermost jar Rico is carrying contains machinery corresponding to the top symbol of S_B as well as a second jar which itself contains machinery corresponding to the next symbol of S_B, etc.). During the checking phase, Rico will need to traverse back up the history of chests and prove that each room corresponding to a symbol at the top of S_A matches the symbol corresponding to the room contained in the outermost jar, i.e. at the top of S_B, and "popping" both off their stacks. Rico can reach the crystal only if they reach the starting room (thereby having emptied S_A) when S_B is also exactly empty, at which point Rico will be carrying the initial block rather than a jar.

3.2 Gadgets

In this section, we will enumerate a collection of gadgets which will be used in the overall construction. A gadget is a template for a section of a level with specific properties. We first describe the One-Way, Proof-of-Holding, and One-Time-Traversal gadgets which are simple and useful subcomponents we will use repeatedly. Section 3.2.4 describes the Prove-Verifygadget which has one entrance which can only be traversed if another entrance has previously been traversed. It is the main component in our ability to record state in our construction. Finally, Section 3.3 gives a brief sketch of the full construction with figures depicting the important rooms in the construction. These include gadgets for the choice of domino placement (Figure 7, symbols in the top stack S_A (Figure 8), and symbols in the bottom stack S_B (Figure 9).

[6] One might be forgiven for referring to this as the "blockchain representation" of S_B.

3.2.1 One-Way

A One-Way gadget allows Rico to pass from one side of the gadget to another, but not back in the opposite direction. Our One-Way gadgets have the additional property that Rico is not able to throw an object through one without traversing it themself.

We use two One-Way implementations: the first is comprised of two "stair steps", each two blocks high, followed by a four block drop. Rico can only jump at most 3 blocks, so after jumping down from the ledge, they cannot get back up. The second implementation is a simple ledge: Rico can jump up onto the ledge but then is unable to get back down. Layout constraints govern the choice of implementation.

The latter ledge implementation trivially satisfies the thrown object requirement, since objects always pass through ledges. The stair step implementation requires one extra feature, which is to make sure the floor below the four block cliff is a ledge, so that any object dropped over the edge will fall through the ledge and become inaccessible, either by getting trapped in an unreachable pit, or by falling off the bottom of the screen, depending on placement. Further, we add a multi-block "stalactite" over the ledge to ensure that any item thrown from above the stairs will hit this wall and fall below the ledge.

The ledge One-Way can be seen on the far left sides of Figures 7 and 8. The stair implementation can be seen once in the bottom of Figure 7 and in triplet in the bottom of Figure 8.

3.2.2 Proof-of-Holding

The Proof-of-Holding gadget (H) is a simple gadget which is traversable if and only if Rico is carrying an object. It has the additional important property that the held object must also traverse the gadget, and cannot be left at the entry side of the gadget for later retrieval. See Figure 3a. It makes use of the fact that while carrying something Rico's jump height is lower. If Rico jumps three blocks high, which is unavoidable while not carrying an object, they will get stuck in the enclosed area at the top of the gadget. However, if Rico is carrying an object, they will jump only two blocks high and land on the lower edge, and have space to walk out of the gadget to the right. The pit at the bottom of the gadget prevents Rico from dropping the held object back down and leaving it behind (accessibly), as it will get stuck in the pit.

3.2.3 One-Time-Traversal

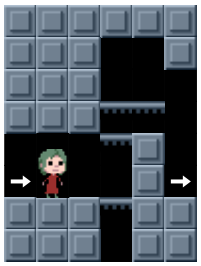
The One-Time-Traversal (1O) is what it says on the tin. Rico can traverse it one time in one direction, after which it cannot be traversed in that direction again. See Figure 3b. It is implemented by forcing Rico to be holding an object (we use a ring so as not to bestow any other abilities) in order to jump up a small step, without irreversibly getting stuck on the ledge 3 blocks up. If Rico is not holding the ring while jumping up the step, getting stuck on the ledge is unavoidable. The gap below the ring's tile is to allow Rico to throw a held object over to the other side of the gadget to be retrieved after traversal (recall that Rico, being two tiles high, cannot fit through that gap).

▶ **Lemma 3.2.** *The One-Time-Traversal gadget can be traversed at most once.*

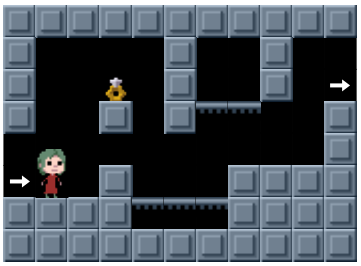
Proof. The reason the gadget is one-time use is because (1) once the ring is removed it is impossible to get it or any other object up to the tile where the ring is initially and (2) the gadget is only possible to traverse from left to right if there is an object present precisely on that tile.

It is easy to see (1) by recalling that Rico cannot jump to a height of 3 blocks while holding an object, nor is there anyway to throw an object upwards with any horizontal velocity, so Rico can neither carry nor throw an object up to the ring's starting tile. Given (1), (2) becomes clear because there is no way Rico can be carrying an object while standing on the lower ledge *except* by grabbing one off the ring's starting tile. ◀

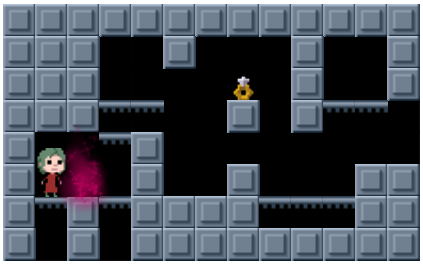
For notational convenience, and because we always want to force Rico to prove that the jar is never dropped, we will always combine One-Time-Traversal with Proof-of-Holding gadgets, to get a gadget which Rico can traverse if and only if they are carrying something and even then at most once. We denote this combined gadget by $1O + H$, or $\twoheadrightarrow$.



(a) Proof-of-Holding gadget.



(b) One-Time-Traversal gadget.



(c) Combined start-of-room construction.

Figure 3 The Proof-of-Holding and One-Time-Traversal gadgets. These exclusively appear together and at the start of most rooms, so we will always use the combined version shown in **(c)** for compactness. Note that Rico cannot jump from the ledge on the left directly to the ring, as they will necessarily bump their head on the "stalactite" block and fall, even while holding an object.

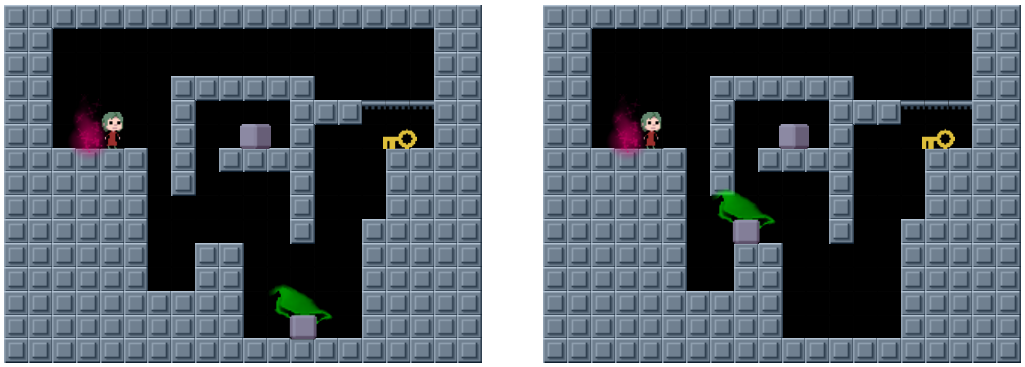
3.2.4 Prove-Verify Gadget

The primary driving gadget behind much of our construction is what we call a Prove-Verify (PV) gadget. The basic idea is that the gadget primarily consists of a single room which contains a green glowing block which can be in one of two states: set or unset. If Rico is able to visit a Prove chest, P, they can put the gadget in the set state. If Rico visits a Verify chest, V, they will be able to retrieve a key from V if and only if the gadget is Set, and in doing so must return it to the Unset state. In this way, retrieving the key from V verifies that P was visited. Note that a Verify chest is always followed by a locked door. Unless otherwise stated, PV gadgets are initially unset. We note that this is a minor variation on the "Self-closing Door" gadget [2] in the framework of [8].

A major use case for PV gadgets is to force Rico to prove that a room is being entered when and from where it is intended to be. To enforce this, for most rooms R, the first element encountered will be a V_R gadget corresponding to that particular room, which will be set

only if Rico is coming into that room immediately after visiting a corresponding P_R gadget in the previous room. Correspondingly, whenever we intend Rico to continue on to R in the intended call stack, we will precede the chest containing R with a P_R gadget followed by a one-way.

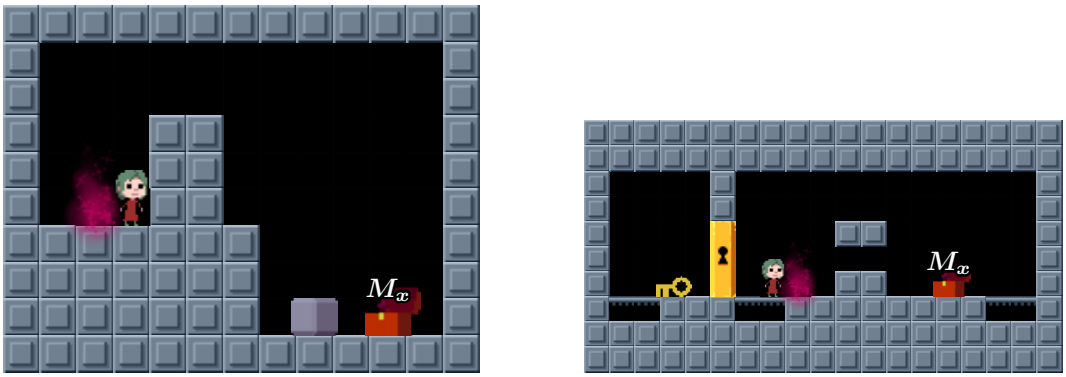
The crux of the gadget is a stateful ***memory room*** M_x shared by a PROVE-VERIFY pair P_x and V_x, shown in Figure 4. If the green block is in the pit, the gadget is Unset, for Rico cannot retrieve the key (or indeed, leave the gadget at all once falling down the cliff next to the entry). If the block is not in the pit, Rico can jump on it to retrieve the other block up on the ledge, and put both of them in the pit which is enough to get up to the key and back around to the exit. Of course, after doing so the green block is in the pit and the gadget is unset again.



(a) Unset state. **(b)** Set state.

Figure 4 The MEM component of the PROVE-VERIFY gadget.

The PROVE gadget simply gives Rico a block to take into the corresponding VERIFY chest with which they can retrieve the green block from the pit (by using both the extra block and the block up on the ledge), shown in Figure 5a. At no point can anything but the block enter the MEM room, due to the 3 block high barrier which Rico cannot carry any object over. Similarly, no object but the key may exit the MEM gadget, and bringing the key out of the MEM chest while in the PROVE chest is clearly not useful due to the same barrier preventing it going anywhere else.



(a) PROVE gadget P_x. **(b)** VERIFY gadget V_x.

Figure 5 The PROVE and VERIFY gadgets.

The VERIFY gadget, shown in Figure 5b, is intended to allow retrieval of a single key if the corresponding MEM room is set. The intended usage is to jump into the MEM room and retrieve a key, then throw that key against the door on the left to allow access to the *other* key, which can then be brought out of the VERIFY chest.

There is some additional complexity in order to prevent cheating, embodied by the following lemma.

▶ **Lemma 3.3.** *No object (other than a block from the corresponding PROVE gadget) can enter the MEM chest.*

Proof. First, note that the MEM chest only appears in the PROVE and VERIFY gadgets. For the PROVE gadget, the 3 tile high barrier ensures that nothing can be brought from the entrance of the gadget to the MEM chest. For the VERIFY gadget, we again use a 3 tile high barrier which nothing can be carried over. However, in order to allow the MEM chest state to interact with the rest of the gadget, we require a gap which the key retrieved from the MEM room can be thrown through, opening the door on the left. The important observation is that no object can be usefully thrown through the gap *except* a key going from right to left hitting and opening the door. Any other object will hit a wall or the door and land inaccessibly in the pit under one of the ledges. Thus, again, no object can enter the MEM chest, and no object can leave the MEM chest except a key, and therefore no object can leave the VERIFY chest except the key behind the door. Finally, we need to ensure that the MEM chest itself cannot exit the PROVE or VERIFY gadgets. Once again, the 3 tile high barriers and the ledges also prevent this.

Note that Rico could bring in a key from outside to open the door with, but all this would achieve is replacing the old key with a new key, so doing so cannot be eminently useful. ◀

If Lemma 3.3 did not hold and Rico could bring in, say, another chest, they could then move the green glowing block from the MEM chest into some other room, and this could subsequently result in Bad Things.

3.2.5 Global-Lock

We will also use a variant of the PROVE-VERIFY gadget semantics which we will call a GLOBAL-LOCK (GL). The GLOBAL-LOCK can be "set" just once, and subsequently used to retrieve a key any number of times. The GLOBAL-LOCK gadget will only be used to allow Rico to transition between the pushing phase and the checking phase, and subsequently verify that the transition was made. The GLOBAL-LOCK is just a room with a green glowing locked door with a key behind it (see Figure 6). As long as the door is locked, the key is irretrievable, but once Rico is given a key to take into even one chest containing the GLOBAL-LOCK room, they can permanently unlock the door and allow all the other chests containing this room to dispense keys.

Figure 6 The GLOBAL-LOCK gadget.

3.3 Construction Sketch

In this section we present figures for the primary structural rooms of the construction (Figures 7, 8, and 9), as well as full details for the A_{ij} rooms (Figure 8) in Section 3.3.1, thus giving a "flavor" of the setup. Full construction details, including those for the DC rooms (Figure 7) and the B_{ij} rooms (Figure 9), are omitted and can be found in our full paper [10].

Figure 7 depicts rooms which allow Rico to select a domino to logically place at each step. Figure 8 depicts a room type representing symbols on stack S_A, and Figure 9 depicts a pair of room types representing symbols on stack S_B. During the checking phase, for each pair of symbols at the front of S_A and S_B Rico will have to perform a 2-way handshake between the machinery in the respective corresponding room A_{ij} and $B^{(J)}_{i'j'}$ to prove that the symbols are the same. If they aren't, Rico will get stuck. On the other hand, if Rico can successfully perform all handshakes and simultaneously empty the call stack and the jar chain, they will be able to deliver the initial block to the start room and reach the goal!

3.3.1 A_{ij} Rooms

Consider the jth symbol in the top half of the ith domino. We uniquely identify that location by A_{ij} and the (nonunique) symbol by $s(A_{ij})$. Similarly, we label the bottom half locations B_{ij} corresponding to symbol $s(B_{ij})$. Each location A_{ij} has a corresponding room, also labeled A_{ij}, shown in Figure 8.

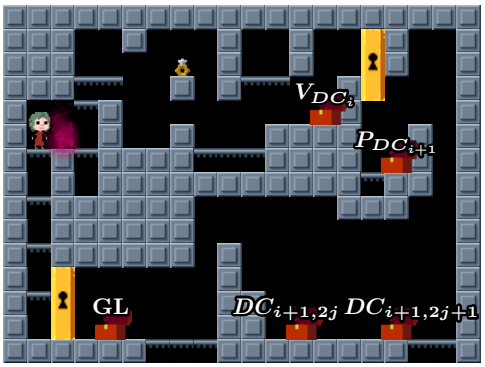


Figure 7 The Domino-Choice rooms, D_{ij}. Rico can traverse these rooms through a series of binary choices to select a domino to logically place at each step.

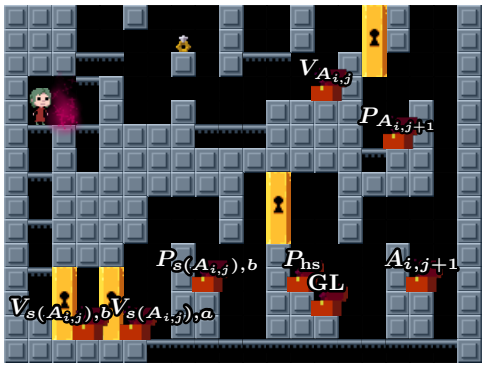


Figure 8 The A_{ij} rooms. When placing domino i, Rico must traverse all of the corresponding A_{ij} rooms. During the checking phase, Rico must traverse the latter halves of these rooms in reverse order, through machinery representing the j^{th} symbol on the domino.

Pushing. During the pushing phase, Rico will do the following. Upon entering, Rico must interact with several elements, each separated from the next by a one-way:

1. Traverse a Proof-of-Holding gadget
2. Traverse a $V_{A_{ij}}$ gadget
3. Traverse a $P_{A_{i,j+1}}$ gadget
4. Enter a chest leading to the next symbol room, $A_{i,j+1}$.

If A_{ij} is the last symbol in the top half string for this domino (i.e. Domino i has exactly j top half symbols), $A_{i,j+1}$ and $P_{A_{i,j+1}}$ will be replaced with B_{i0} and $P_{B_{i0}}$, leading to the B_{ij} rooms for this domino. If the bottom string of Domino i is empty, the replacements will instead be the first Domino-Choice room, DC_{00}, and $P_{DC_{00}}$, respectively. This is as far as Rico will go during the Domino Selection phase.

Popping. Of course, that is not the end of the room, for when Rico jumps back out of the $A_{i,j+1}$ chest they entered in Step 4 above during the checking phase, this is the point where they will need to pop a symbol from S_B and prove that it is equal to a_{ij}. Note that Rico will not be able to usefully go back into the $A_{i,j+1}$ chest they just jumped out of, since there will immediately be an untraversable $V_{A_{i,j+1}}$ gadget.

During the checking phase, Rico must take the following steps, comprising a 2-way handshake to prove that $S(A_{ij})$ and $b_{i'j'}$ are equal to each other. Again, all elements are separated by One-Ways (besides entering and exiting held jars, of course).

1. Traverse the Global-Lock gadget to prove the checking phase has been entered and a "handshake" chest P_{hs}, allowed in either order to save space. The P_{hs} gadget will prove that the handshake has been appropriately initiated (see below).
2. Traverse a $P_{s(A_{ij}),b}$ gadget corresponding to the "bottom half b version" of $s(A_{ij})$.
3. Delve into the jar they should be carrying, hopefully containing $B_{ij}^{(J)}$ with $s(A_{ij}) = s(B_{ij})$.
4. Inside the jar (refer to the full construction [10] for details), traverse a $V_{s(B_{ij}),b}$ gadget, only possible if $s(A_{ij}) = s(B_{ij})$.
5. Traverse a $P_{s(B_{ij}),a}$ gadget corresponding to the "top half a version" of $s(A_{ij})$.
6. Traverse a V_{hs} chest.
7. Exit the jar, thereby destroying the $B_{ij}^{(J)}$ instance, effectively popping S_B.
8. Traverse the $P_{s(A_{ij}),b}$ from Step 2 *again* (see below for an explanation).
9. Traverse a $V_{s(A_{ij}),a}$ gadget, again only possible if $s(A_{ij}) = s(B_{ij})$.
10. Traverse an instance of $V_{s(A_{ij}),b}$ to re-unset it from Step 8.

The P_{hs}-V_{hs} handshake pair is necessary to disallow popping multiple instances of $s(A_{ij})$ off of S_B. Without it, after Step 8, Rico could enter the new top jar, and traverse it successfully, contingent on the symbol it corresponds to being equal to $s(A_{ij})$. However, the V_{hs} gadget prevents this, since it will become unset after having traversed the previous intended jar.

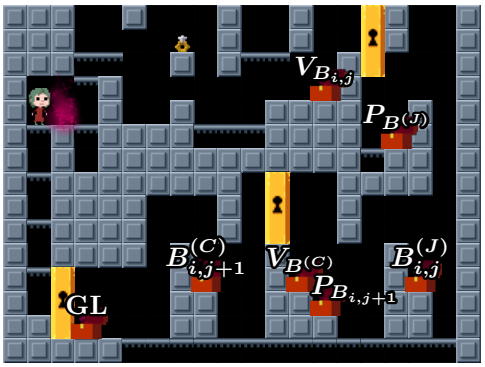
Steps 8 and 10 are necessary because there is no way to prevent Rico from traversing the $P_{s(A_{ij}),b}$ an extra time after exiting the jar, which could allow future unintended traversals of a $V_{s(A_{ij}),b}$ somewhere else. Thus, we must simply assume that Rico will traverse the $P_{s(A_{ij}),b}$ chest again and then force them to unset the $V_{s(A_{ij}),b}$ in Step 10.

As mentioned above, please refer to our full paper [10] for a complete discussion of the DC and B_{ij} rooms and the remaining construction details.

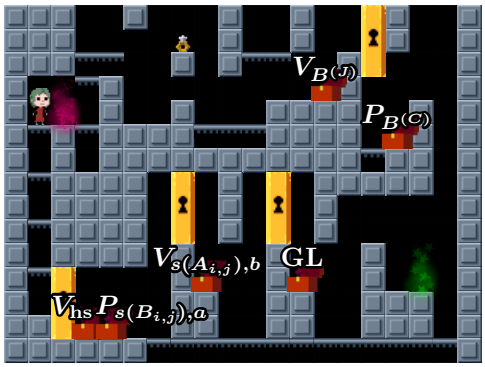
4 Open Problems

Although we achieve a tight result of RE-completeness, we could still ask about the complexity of Recursed with a subset of the puzzle mechanics. We propose two conjectures and two open problem relating to subsets of Recursed mechanics.

▶ **Conjecture 1** (Cauldrons but no Jars). *We conjecture that Recursed with Cauldrons, but no Jars, is still undecidable.*



(a) $B^{(C)}_{ij}$ "call stack" rooms. When placing domino i, traversing room $B^{(C)}_{ij}$ for each bottom symbol j will force Rico to add room $B^{(J)}_{ij}$ (right) to the jar chain.



(b) $B^{(J)}_{ij}$ "jar chain" rooms. Rico will be forced to traverse the second halves of these rooms during the checking phase, again through machinery representing the symbols on the bottom stack.

Figure 9 The B_{ij} rooms.

This conjecture seems very likely because Cauldrons (which are intended to intuitively represent multi-threading) allow Rico to jump between different "worlds" (up to 4, represented visually by background color) which *each have their own chest history.* Thus, it should not be difficult to build a reduction similar to ours which makes use of multiple stacks to simulate PCP or 2-stack Push Down Automata, both of which are undecidable.

▶ **Conjecture 2** (No Cauldrons or Jars). *We conjecture that Recursed without Cauldrons or Jars can be simulated by a Push-Down Automata, and is therefore decidable.*

The main difficulty with proving this conjecture is that during a solution, rooms can contain an unbounded number of objects (blocks, keys, or chests), and such state can not be trivially stored in either the automata head, or on the stack. However, we conjecture that after some bounded point, more objects of a given type cannot help towards a solution, and can therefore be forgotten. However, this seems difficult to prove.

▶ **Open Problem 3** (Jars or Cauldrons but no Green Glow). *What is the complexity of Recursed with Jars or Cauldrons or both, but without green glowing objects?*

Green glowing objects are not required to build some form of two or more stateful stacks with either Jars or Cauldrons, but it seems very difficult to construct reductions without them. It's possible that there simply is not enough interaction amongst the limited set of objects in Recursed for this problem class to be undecidable, but it seems quite difficult to rule out.

▶ **Open Problem 4.** *What is the complexity of Recursed restricted to a polynomial-length "room stack" (analogous to call stack)?*

This problem is naturally in NPSPACE = PSPACE, but is it PSPACE-complete? This question likely needs a different approach, as our reduction is focused on time simulation and not on multiple uses of gadgets.

References

1. Zachary Abel, Jeffrey Bosboom, Erik D. Demaine, Linus Hamilton, Adam Hesterberg, Justin Kopinsky, Jayson Lynch, and Mikhail Rudoy. Who witnesses The Witness? Finding witnesses in The Witness is hard and sometimes impossible. In *Proceedings of the 9th International Conference on Fun with Algorithms (FUN 2018)*, pages 3:1–3:21, La Maddalena, Italy, June 2018.
2. Joshua Ani, Sualeh Asif, Erik D. Demaine, Yevhenii Diomidov, Dylan Hendrickson, Jayson Lynch, Sarah Scheffler, and Adam Suhl. PSPACE-completeness of pulling blocks to reach a goal. In *Abstracts from the 22nd Japan Conference on Discrete and Computational Geometry, Graphs, and Games (JCDCGGG 2019)*, pages 31–32, Tokyo, Japan, September 2019.
3. Alex Churchill. Magic: the Gathering is Turing complete. `http://www.toothycat.net/~hologram/Turing/`, 2012.
4. Alex Churchill, Stella Biderman, and Austin Herrick. *Magic: The Gathering* is Turing complete. arXiv:1904.09828, 2019. `arXiv:1904.09828`.
5. Computational complexity theory Steam curator. `https://store.steampowered.com/curator/31317680-Computational-Complexity-Theory/`, 2017. Steam curator page claiming hardness results for various games.
6. Michael J. Coulombe and Jayson Lynch. Cooperating in video games? impossible! undecidability of team multiplayer games. In Hiro Ito, Stefano Leonardi, Linda Pagli, and Giuseppe Prencipe, editors, *Proceedings of the 9th International Conference on Fun with Algorithms (FUN 2018)*, volume 100 of *LIPIcs*, pages 14:1–14:16, La Maddalena, Italy, June 2018. `doi:10.4230/LIPIcs.FUN.2018.14`.
7. Erik D. Demaine. SVG Tiler. `https://github.com/edemaine/svgtiler`, 2020.
8. Erik D. Demaine, Isaac Grosof, Jayson Lynch, and Mikhail Rudoy. Computational complexity of motion planning of a robot through simple gadgets. In *Proceedings of the 9th International Conference on Fun with Algorithms (FUN 2018)*, volume 100 of *LIPIcs*, pages 18:1–18:21, La Maddalena, Italy, June 2018.
9. Erik D. Demaine and Justin Kopinsky. recursed-xls2lua. `https://github.com/edemaine/recursed-xls2lua`, 2020. Tool to convert xls descriptions of Recursed levels to playable lua files, with examples.
10. Erik D. Demaine, Justin Kopinsky, and Jayson Lynch. Recursed is not recursive: A jarring result. arXiv:2002.05131, 2020. `arXiv:2002.05131`.
11. edderiofer. edderiofer Steam review. `https://steamcommunity.com/id/edderiofer/recommended/497780/`, 2017. User review for Recursed which conjectures undecidability.
12. Linus Hamilton. Braid is undecidable. arXiv:1412.0784, 2014. `arXiv:1412.0784`.
13. Robert A. Hearn and Erik D. Demaine. *Games, Puzzles, and Computation*. A. K. Peters, Ltd., Natick, MA, USA, 2009.
14. Marvin L. Minsky. Recursive unsolvability of Post's problem of "Tag" and other topics in theory of Turing machines. *Annals of Mathematics*, 74(3):437–455, November 1961.
15. Portponky. Recursed. `https://store.steampowered.com/app/497780/Recursed/`, 2016. URL: `https://store.steampowered.com/app/497780/Recursed/`.
16. Portponky. Recursed - fissure / jar mechanic. `https://youtu.be/WumGkuBzvLQ`, 2016.
17. Emil L. Post. A variant of a recursively unsolvable problem. *Bulletin of the American Mathematical Society*, 52(4):264–268, 1946.
18. Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 3rd edition, 2012.