Discrete and Computational Geometry, Graphs, and Games

Celeste is PSPACE-hard

Lily Chung and Erik D. Demaine

MIT Computer Science and Artificial Intelligence Laboratory, 32 Vassar St., Cambridge, MA 02139, USA
e-mail : lkdc@mit.edu (L. Chung); edemaine@mit.edu (E. Demaine)

Abstract We investigate the complexity of the platform video game Celeste. We prove that navigating Celeste is PSPACE-hard in five different ways, corresponding to different subsets of the game mechanics. In particular, we prove the game PSPACE-hard even without player input.

1. INTRODUCTION

Celeste[1] is a 2D platform video game released in 2018 by Maddy Makes Games. It won the Best Independent Game and Games for Impact awards at The Game Awards 2018 [1] and sold over a million copies [2]. In Celeste, the player controls a single character, Madeline, who must navigate various hazards along her journey. We analyze the following natural decision problem about Celeste:

Definition 1.1 (CELESTE)**.** Given a Celeste level, is it possible for Madeline to traverse from a designated start location to a designated end location?

A previous paper [3] attempted to resolve this question by claiming that CELESTE is NP-complete, but failed to correctly prove containment in NP. Specifically, their proof made the incorrect assumption that the sequence of inputs solving a CELESTE instance must be polynomially bounded in size. Their NP-hardness reduction applies the framework from [4] to show hardness with barriers, gates, and buttons.[2] They also showed that adding additional mechanics to Celeste (buttons that close gates instead of opening them) suffices for PSPACE-hardness, using the pressure-plate framework of [5]. By contrast, we show that Celeste's built-in mechanics, excluding gates and buttons, suffice for PSPACE-hardness.

---

[1]https://exok.com/games/celeste/. Celeste and its sprites are the properties of Maddy Makes Games. Sprites are used here under Fair Use for the educational purpose of illustrating mathematical theorems.

[2]Their clause gadget needs some modification to respect Celeste's mechanic that a button opens the Euclidean-nearest gate, but this is easy to do.

jump-through	crumble blocks	spinners	spring	seeker	jellyfish	pufferfish	barrier	move blocks	Kevin blocks	Complexity	Sec
		✓		✓			✓	✓		PSPACE-hard	§3.1
		✓			✓		✓			PSPACE-hard	§3.2
		✓				✓				PSPACE-hard	§3.3
		✓							✓	PSPACE-hard	§3.4
			✓		✓			✓		PSPACE-hard	§4
✓	✓	✓	✓	✓	✓	✓	✓	✓		∈ PSPACE	§2

TABLE 1. Summary of our results about the complexity of CELESTE.

We give five proofs that CELESTE is PSPACE-hard, each using different restricted combinations of existing game mechanics; see Table 1. Four of these proofs involve constructing a polynomial-time reduction to CELESTE from a motion-planning problem through a planar network of doors [6]. We make use of both "open–close–traverse" doors, as introduced in [4, 5, 7] and shown not to need crossovers in [6], and "self-closing doors", as introduced in [6]. In each case we construct a Celeste level corresponding to the motion-planning problem that can be traversed if and only if the motion-planning problem is solvable. In all but one case we additionally show containment in PSPACE, establishing PSPACE-completeness.

We also consider the following problem, where we ignore the player and treat Celeste as an automaton:

Definition 1.2 (ZERO-PLAYER CELESTE). Given a Celeste level and Madeline's starting position, will Madeline ever reach a designated end position if the player makes no inputs?

One of our proofs (Section 4) shows that ZERO-PLAYER CELESTE with a certain combination of game mechanics (springs, jellyfish, and move blocks) is PSPACE-complete by reducing from a zero-player motion-planning problem [8]. Adapting this result gives our fifth proof that CELESTE is PSPACE-hard.

We have tested our constructions in the Celeste game itself; the custom map file and some dynamic illustrations can be found on an accompanying website.[3]

2. DEFINITIONS

We define an idealized version of Celeste which captures its relevant behavior. A Celeste level[4] of size M consists of a subset of $\mathbb{Z}^2$ called the **_tilemap_**, which defines where the solid walls are in the level. It additionally contains a polynomially sized set of entities of various types, each placed at a specific initial location. The locations of entities are not necessarily aligned with the integer tile grid, but we require the tilemap and entities to be confined to an $M \times M$ rectangle. In Celeste, many quantities such as positions,

---

[3]https://github.com/cryslith/celeste-constructions

[4]Chapters in Celeste consist of many levels, each of whose state does not persist across level boundaries. There is no bound on the size of a level; our reductions will each produce a single large level rather than a collection of small ones.

sizes, and velocities, are stored as either 32-bit integers or 32-bit floating point numbers. We will ignore unusual behavior caused by overflow or loss of precision, and instead work with the assumption that Celeste physics are translation-invariant.

The player interacts with Celeste by controlling the main character, Madeline; see Figure 1. Madeline can move left and right on the ground or in the air, jump off of the ground or walls, and climb walls for a limited time. She can also *dash* in any of the eight cardinal and ordinal directions, which gives her a short-lasting boost of speed in that direction. She can dash only once in the air, after which she has to land on the ground before dashing again.[5]

Madeline can also perform a number of more advanced movements, such as super-dashes, hyperdashes, ultras, wallbounces, and crouch-dashes. However, these do not allow her to traverse our gadgets in unexpected ways, and are not necessary for their intended operation. For instance, superdashes, hyperdashes, and ultras can impart more horizontal velocity than usual when jumping from the ground, but breaking most of our gadgets would require Madeline to gain vertical height rather than horizontal distance. Wallbounces can give additional vertical height but require a suitably-placed wall, which we avoid. Crouch-dashes can sometimes be used to bypass obstacles by squeezing between them, but we place our obstacles close enough together to prevent this. It is possible that an undiscovered bug or exploit in the game physics could break our gadgets, but in the following we will assume this is not the case.

FIGURE 1. Madeline, at rest and while dashing

Ordinarily, while falling Madeline can achieve a maximum "glide ratio" of at most 0.563. That is, for every tile she falls vertically, she can drift only 0.563 tiles horizontally[6]. Dashing horizontally can move her up to 9.6 tiles horizontally, while dashing diagonally upwards can move her by at most 8 tiles horizontally and 2.1 tiles upward, after which her glide ratio decays to its baseline. We exploit these limits to require Madeline to expend dashes and utilize entities in order to traverse sections of our constructions.

We make use of the following entities in our constructions.

Jumpthroughs can be passed through from below, but not from above. Madeline activates *crumble blocks* by contacting them from above or from the sides (but not from the bottom), which makes them briefly disappear. These blocks act as one-way "diodes" which Madeline can traverse only in one direction (upwards and downwards respectively).

---

[5]Later in the game, Madeline gains the ability to dash twice before landing on the ground, but we ignore this additional ability as it does not affect our PSPACE containment results. Our hardness constructions could also be modified to be robust to a constant number of dashes.

[6]Detailed information on Celeste's mechanics was obtained from many sources, including reverse-engineering the game code.

Spinners instantly kill Madeline upon her colliding with them, respawning her at her starting position and resetting the entire level to its initial state.[7] This is never beneficial for the player. Spinners do not interact with entities other than Madeline.

Springs in the pictured vertical orientation launch Madeline and jellyfish sideways and slightly upwards. Horizontal springs launch Madeline and jellyfish directly upwards while preserving some of their existing horizontal momentum.

Seekers kill Madeline on contact. They can be pushed around by moving blocks, but otherwise remain still. If they ever come into line-of-sight of Madeline, they chase her and exhibit more complicated behavior; we will avoid defining this behavior by forcing Madeline to either avoid line-of-sight or contact the seeker in our constructions.

Jellyfish can be held by Madeline to float through the air when falling, increasing her horizontal speed and reducing her terminal velocity. This improves her maximum glide ratio to 4.5. Madeline can also throw jellyfish a short distance.

Madeline can interact with ***pufferfish*** by jumping on top of them, which restores her dash and moves the pufferfish downwards a short distance. If she instead approaches the pufferfish from below or to the side, the pufferfish will explode, restoring her dash and bouncing her away. The pufferfish will then respawn at its original position.

Barriers are intangible to Madeline, but act as solid walls for seekers. They also permanently destroy any jellyfish that contact them.

Madeline activates a ***move block*** by touching it from above or from the sides, which causes it to begin moving in the direction displayed as an arrow on the block. The block will keep moving until obstructed by a solid wall, at which point it disappears and respawns at its original position. A move block with a spring attached will also be activated if Madeline or a jellyfish contacts the spring.

Kevin blocks[8] have complex behavior. Each Kevin block maintains a stack of (position, direction) pairs, initially empty. Madeline activates the Kevin block by dashing into it from any side. Upon activation, the Kevin block pushes its current position onto its stack along with the direction Madeline dashed into it from, *provided* that the direction currently on top of the stack (if any) is perpendicular to this new direction. The effect of this condition is that consecutive triples of positions on the stack always form right angles; they are never collinear. In any case, the Kevin block then begins charging quickly in the direction *toward* the side that Madeline hit, until the Kevin block hits a wall. Whenever it is not charging, the Kevin block will retrace its path by

---

[7]A recently discovered bug allows the player to prevent spinner hitboxes from loading by pausing the game periodically, which permits Madeline to bypass these obstacles. This potential cheat can be prevented by duplicating the spinners within a level, essentially because each spinner is randomly assigned to one of three groups when the level is created, and the player can only deactivate one group at a time. (More precisely, each spinner is assigned a point on the unit circle, and the player can only deactivate a set of spinners corresponding to an interval of length $2\pi/3$.) Duplicating each spinner $\Theta(\log n)$ times ensures that, with high probability, no obstacle can be bypassed in this way, where n is the number of spinners in the original level.

[8]Named after Kevin Regamey, Celeste's sound designer.

slowly moving towards the position on top of its stack, removing that position from the stack when it arrives there.

Lemma 2.1. CELESTE *with the listed entities other than Kevin blocks is contained in PSPACE.*

Proof. The states of Madeline and of every entity other than Kevin blocks can be described by a polynomial amount of space, since entities are confined to a polynomial-sized area. Thus an algorithm which guesses Madeline's inputs on every frame and simulates Celeste, until Madeline either reaches the goal location or a state is repeated, requires only polynomial space to function, showing containment in NPSPACE = PSPACE [9]. ∎

3. SINGLE-PLAYER HARDNESS

In order to show PSPACE-hardness, we simulate certain "door gadgets". We make use of both "open–close–traverse" doors, as introduced in [4, 5, 7], and "self-closing doors", as introduced in [6]. Refer to Figure 2. An ***open–close–traverse door*** is a 2-state gadget with three tunnels labeled "open", "close", and "traverse". Traversing the open and close tunnels respectively changes the gadget's state to open or closed; the traverse tunnel can be traversed only in the open state. A ***self-closing door*** is a 2-state gadget with two tunnels labeled "open" and "self-close"; traversing the self-close tunnel is possible only in the open state and changes the gadget's state to closed, preventing it from being traversed again until the open tunnel is traversed and the state is reset to open. A ***symmetric self-closing door*** is similar, except that the open tunnel (more symmetrically called a "self-open" tunnel) cannot be traversed while the door is in the open state. All tunnels are ***directed***: they can be traversed in only one direction. Optionally, the open–close–traverse door or self-closing door (but not the symmetric self-closing door) can be made ***open-optional*** meaning that the two end locations of the open tunnel are identified, making the open tunnel into an open ***port***, so the agent can always freely choose whether to open the door or just skip the traversal.

A key set of results from [6] (Theorems 4.1 and 4.7) is that the "planar motion planning problem" is PSPACE-hard for any one of these types of doors[9]. More precisely, for any gadget, a ***planar network of gadgets*** consists of a finite number of instances of that gadget, each with a specified initial state, and an undirected graph connecting together the ends of tunnels (called "locations"), such that the gadgets and graph can be drawn in the plane without crossings. The ***planar (1-player) motion planning problem*** asks, given a planar network of gadgets, a start location, and a destination location, whether there is a traversal sequence from start to destination. Because it is easy to build "hallways" (paths that Madeline can traverse in any direction) and "branching hallways" (connections where Madeline can freely choose to follow any incident hallway), the graph part of a planar network is easy to represent. Therefore, constructing a gadget that simulates any one door is enough to show PSPACE-hardness of traversing Celeste levels.

The following four constructions all use jumpthroughs, crumble blocks, and spinners, so we omit their mention in the section titles, and instead just list the unique mechanics that each construction uses. The jumpthroughs and crumble blocks are convenient shorthand for one-way diodes, which can instead be replaced by the gadget in Figure 3, so they are not listed in the theorems or Table 1.

---

[9]Except for a specific planar arrangement of the tunnels of an open–close–traverse door; we will avoid that arrangement.

(A) Open–close–traverse door

(B) Self-closing door
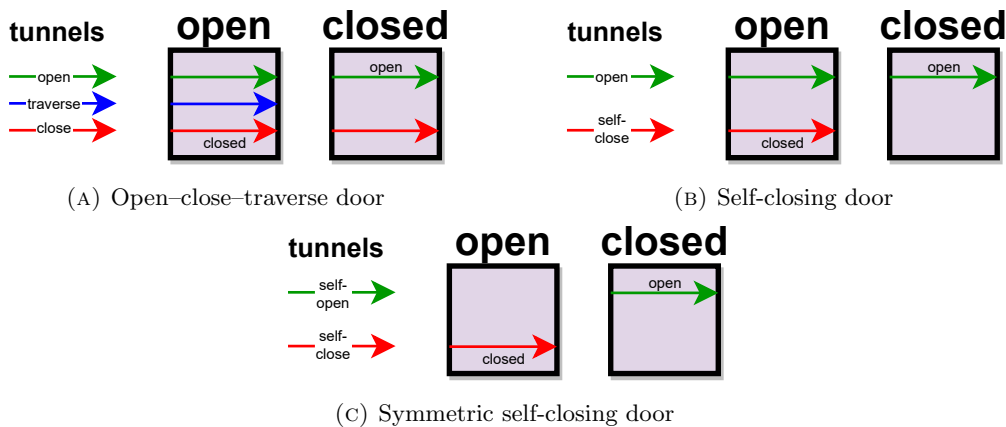

(C) Symmetric self-closing door

FIGURE 2. The three types of door gadgets we use, from [6]. Each diagram consists of a legend of labeled tunnels on the left, and the traversals possible in each of the two states of the gadget ("open" and "closed") on the right. Each traversal that changes the state of the gadget is labeled with the state that it changes to.
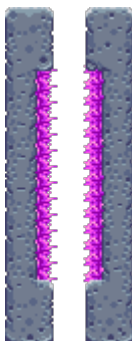

FIGURE 3. A one-way diode gadget, which can be traversed only from top to bottom. Madeline cannot ascend from bottom to top even with a dash.

3.1. SEEKERS, BARRIERS, AND MOVE BLOCKS

Theorem 3.1. CELESTE *with spinners, seekers, barriers, and move blocks is PSPACE-complete.*

Proof. We reduce from planar motion planning with open–close–traverse doors. Figure 4 shows the simulation of the door.

The seeker is constrained by a ring of barriers. Whenever the seeker is in the top-right corner of the ring, Madeline can traverse the middle tunnel by falling down and dashing rightwards to avoid the spinners. However, when the seeker is in the bottom-left corner, she cannot traverse the middle tunnel without hitting it. The door is opened by traversing the left tunnel, which activates two move blocks, pushing the seeker to the
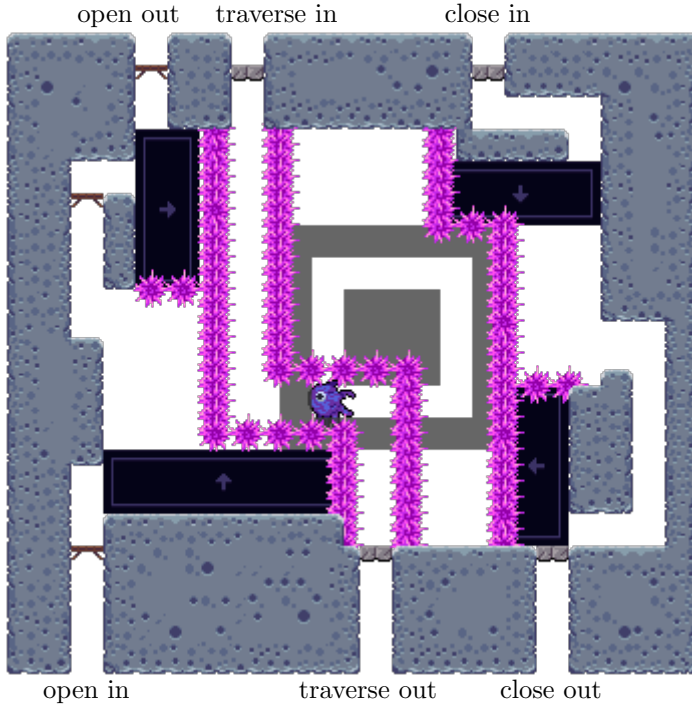
open out   traverse in   close in

open in   traverse out   close out

FIGURE 4. An open–close–traverse door constructed with a seeker, barriers, and move blocks. Currently in the "closed" state.

top-right corner. (In particular, after activating the first move block, Madeline can follow close behind and get past it when the move block hits the solid wall, disappears, and respawns at its original location.) Similarly, traversing the right tunnel activates two other move blocks which return the seeker to the bottom-left corner. Jumpthroughs and crumble blocks ensure that the gadget does not reach an invalid state. For example, once Madeline has triggered one of the move blocks, she cannot exit the gadget (e.g., via the entrance she just used) except via the intended exit. Importantly, it is not possible for Madeline to be in line-of-sight of the seeker without dying; this is necessary to avoid triggering the seeker's complex chasing behavior. ∎

3.2. Jellyfish and Barriers

Theorem 3.2. CELESTE *with spinners, jellyfish, and barriers is PSPACE-complete.*

Proof. We reduce from planar motion planning with open-optional self-closing doors. Figure 5 shows the simulation of the door.

The right-side self-closing tunnel can be traversed from top to bottom only if Madeline has a jellyfish, which she can use to drift around the spinners. However, she cannot traverse it without a jellyfish because her ordinary glide ratio is not enough to get around the corners; the best she can do is use her single dash to get around one of them.

After Madeline traverses the right-side tunnel, the jellyfish is stuck at the bottom with her. The only way to return it to the top of the gadget is to first throw it through the spinners into the left chamber. The door is reopened by entering the left chamber,
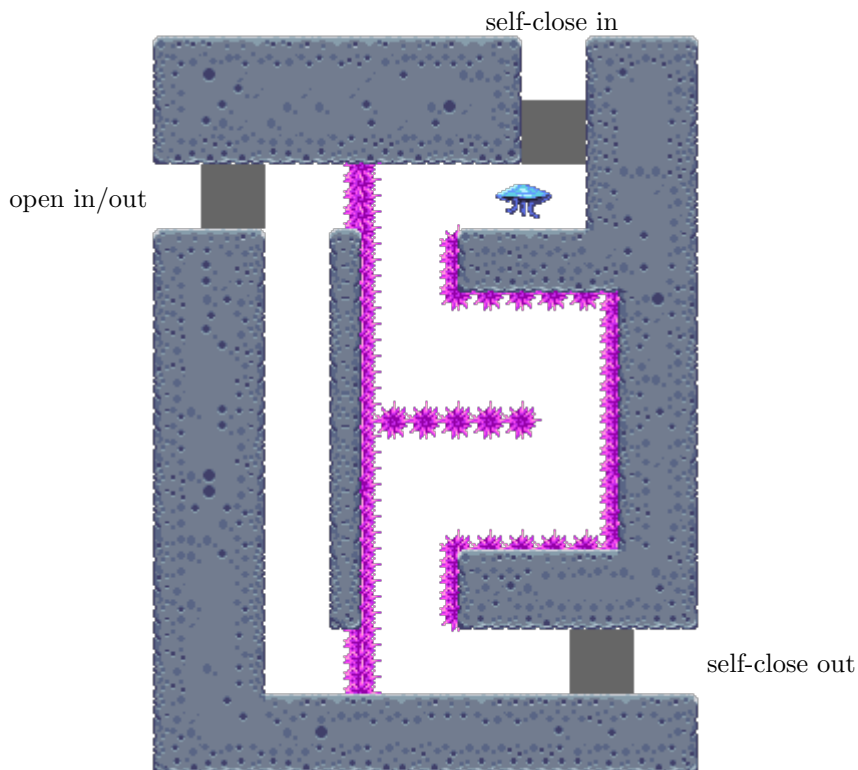
FIGURE 5. An open-optional self-closing door using a jellyfish and barriers, in the "open" state.

retrieving the jellyfish from the bottom, and throwing it back across the spinners at the top. The jellyfish can never exit the gadget because of the barriers blocking the entrances. (Recall that barriers permanently destroy jellyfish, so they also cannot be used to reset the jellyfish's position and re-open the gadget.) ∎

3.3. Pufferfish

Theorem 3.3. Celeste *with spinners and pufferfish is PSPACE-complete.*

Proof. We reduce from planar motion planning with symmetric self-closing doors. Figure 6 shows the simulation of the door.

Initially the bottom tunnel is untraversable, and Madeline can traverse the top tunnel only by dropping in, dashing right, jumping on the pufferfish, and dashing upwards to exit. This moves the pufferfish downwards through the spinners into the central area.

Now the top tunnel cannot be traversed (because Madeline has only a single dash), but Madeline can traverse the bottom tunnel by dashing upwards into the pufferfish's explosion radius. The pufferfish's explosion launches her to the right wall, which she can climb to the exit. The pufferfish then respawns in its original position, resetting the gadget.

self-close in   self-close out
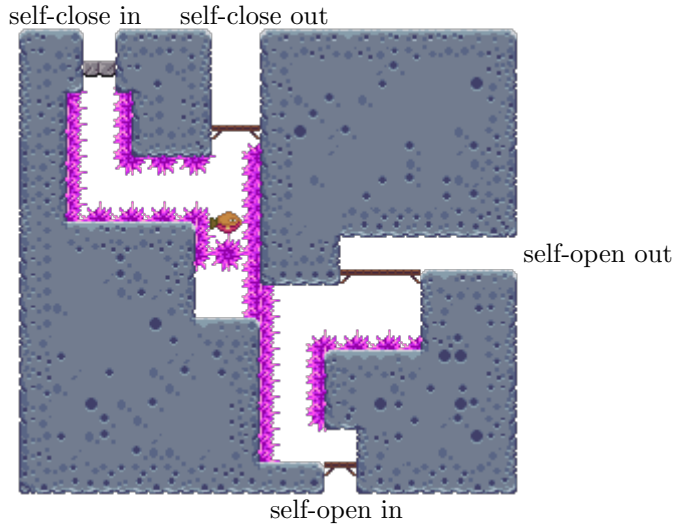

self-open out

self-open in

FIGURE 6. A symmetric self-closing door using a pufferfish. Initially the top tunnel is open and the bottom tunnel closed.

A minor issue with this construction is that because the pufferfish always begins the level at its spawn point, the gadget cannot be initialized with the bottom tunnel open and the top tunnel closed. This is solved by reflecting the gadget about the y axis to obtain a door gadget that initially has the opposite tunnel open. ∎

3.4. Kevin Blocks

In order to show that Celeste is PSPACE-hard with Kevin blocks, we will construct an open-optional self-closing door gadget. One difficulty is that our construction always begins in the closed state, but we will show that this nonetheless suffices for PSPACE-hardness.

Lemma 3.4. *Planar motion planning with initially closed open-optional self-closing doors is PSPACE-complete.*

Proof. We reduce from motion planning with open-optional self-closing doors that may begin in either state.

First, we use a combination of open-optional self-closing doors to create an open-optional self-closing door with two opening ports,[10] as shown in Figure 7. This construction needs one-way diodes, which are easy to implement with a directed open-optional self-closing door by identifying the open port with the entrance to the self-close tunnel. We replace every initially open door in the given instance with one of these doors, giving each one an extra opening port. Finally, we build a new traversal path for the agent that starts from a new start location, visits the extra opening port of each initially open door in sequence, and then proceeds to the original start location of the given instance. We place a one-way diode at the end of the path, so the agent can visit all the extra ports at the beginning but never again. Opening doors to gadgets can only help later traversal,

---

[10]The same construction is used for a different purpose in Theorem 3.3 of [6].

so we can assume that the agent visits all the extra ports, and thus opens all doors that were supposed to be initially open.
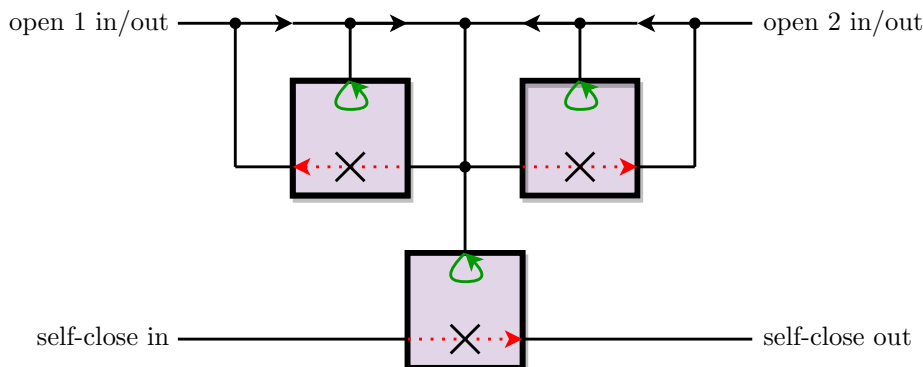

FIGURE 7. Duplicating the opening port of a directed open-optional self-closing door: both of the two upper ports open the bottom self-close tunnel, without it being possible to leak between the two ports. Each box denotes an open-optional self-closing door, where the green loop is the open port and the dotted arrow denotes the self-close tunnel in an initially closed state. The arrows at the top of the diagram (exterior to gadgets) are one-way diodes.

This construction does not preserve planarity. Luckily, the simulation of a directed crossover in Theorem 4.1 of [6] uses only initially closed doors. Thus we can make our construction planar by replacing any crossing wires with these simulated crossovers, while preserving that all doors are initially closed. ∎

Now we can show the following result about Kevin blocks.

Theorem 3.5. Celeste *with spinners and Kevin blocks is PSPACE-hard.*

Proof. We reduce from planar motion planning with initially closed open-optional self-closing doors, which is PSPACE-complete according to Lemma 3.4. Figure 8 shows the simulation of the door.

Initially the right tunnel is untraversable, since the Kevin block is placed too high up for Madeline to dash into it. By entering from the left, Madeline can climb the wall and move the Kevin block into the left chamber. By dashing into it repeatedly from different directions, she can "wind up" the Kevin block, adding arbitrarily many positions to its internal stack (e.g., by moving it clockwise around a rectangle). Once this is done, the block will take arbitrarily long to unwind, depending on how long it was wound up for. The right tunnel can be traversed only when the Kevin block finishes unwinding and returns to its original position, when Madeline can ride on top of it. The constrained space prevents the Kevin block from possibly moving through the right tunnel more than once per winding session, because positions can be added to its stack only when it makes right-angle turns. Therefore, every traversal of the right tunnel corresponds to an immediately previous visit to the left chamber, which is the same condition as a self-closing door.
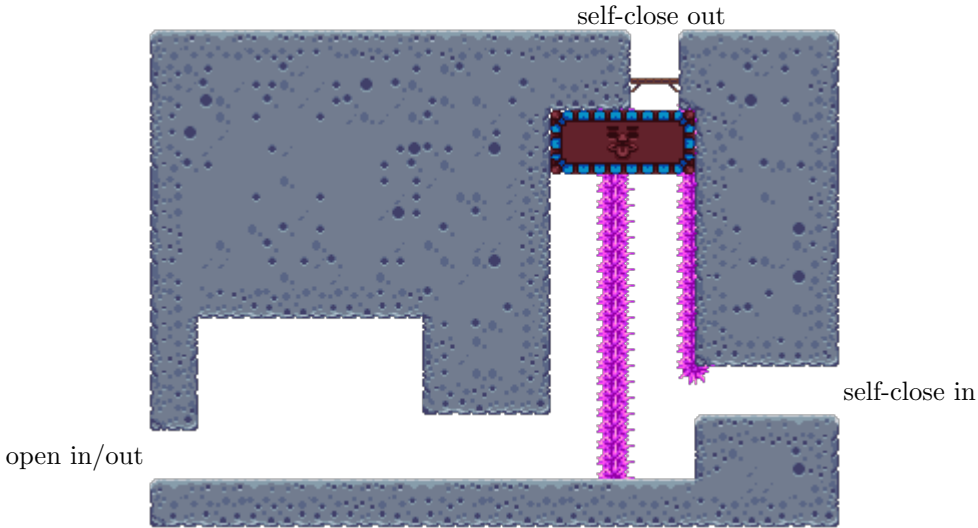
FIGURE 8. An open-optional self-closing door using a Kevin block. Initially in the "closed" state.

It follows that any traversal of a network of Kevin-based doors corresponds to a traversal of the corresponding network of self-closing doors. The only obstacle to showing the converse is in solving certain timing constraints: Whenever Madeline needs to go through an open door, the corresponding Kevin block must still be unwinding; that is, it must not have been wound for too short a time. Fortunately, these constraints are always solvable by the following method. List the doors in reverse order by the time at which the open tunnel is visited (each door will in general appear multiple times on the list). In this order, assign an amount of time to wind each door sufficient to keep the door wound until its self-close tunnel must be traversed. This quantity of time depends only on how much time was assigned for winding previous doors in the list. Because the constraints are always solvable, this shows that any traversal of the initially closed self-closing door network can be transformed into a traversal of the network of Kevin-based doors. ∎

4. ZERO-PLAYER HARDNESS

In this section we prove that predicting the outcome of a Celeste level is PSPACE-complete, ignoring player inputs. We do so by reducing from a zero-player motion-planning problem, in which an agent traverses a network of gadgets entirely deterministically. Specifically, in *zero-player motion planning* [8], the gadgets must be *input/output*, meaning that their locations can be partitioned into entrances (inputs) and exits (outputs) — no location is an entrance for a transition in some state and an exit for another transition in some state — and the connection graph connecting gadget locations must be *branchless*, meaning that it has at most one input location in each connected component. Thus the motion of the agent is fully determined from its starting location and the initial gadget states; the goal is to determine whether the agent ever reaches a given destination location.

The input/output gadget we consider is the ***set-up switch/set-down switch***; refer to Figure 9. This 2-state gadget has two input locations, labeled "set-up" and "set-down", and four output locations, labelled "(up, up)", "(up, down)", "(down, up)", and "(down, down)". If an agent enters at input location set-i, then they are forced to exit at output location (i, s) where s is the state of the gadget before traversal, and then the gadget's state is set to i.
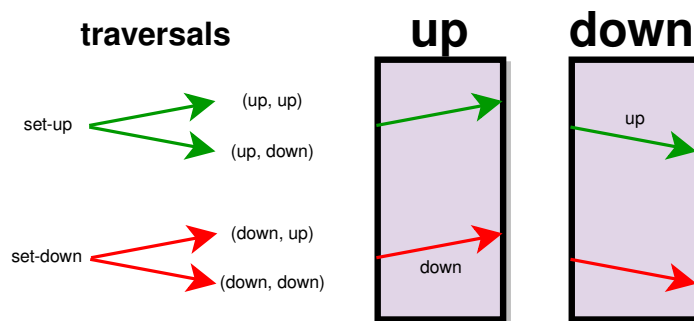

FIGURE 9. The set-up switch/set-down switch gadget. Left: The six locations (two input and two output) along with arrows for all possible traversals between them. Right: The traversals possible from each of the two states, with labels for traversals which modify the resulting state.

Ani et al. [8] gave a partial classification of zero-player gadgets and the resulting complexity of the zero-player motion planning problem. An important result (Corollary 2.8) is that motion planning is PSPACE-complete for any unbounded output-disjoint deterministic 2-state input/output gadget with multiple nontrivial inputs. Examining the set-up switch/set-down switch, we see that it is a 2-state input/output gadget, it is unbounded (can change states arbitrarily many times), output-disjoint (no output can be reached from multiple inputs), and deterministic (each input leads to a unique transition/output in any given state). Neither of the inputs is a trivial tunnel (avoiding interacting with the state at all), so the corollary shows that zero-player motion planning with this gadget is PSPACE-complete.

We reduce this zero-player motion-planning problem to ZERO-PLAYER CELESTE by simulating a set-up switch/set-down switch. Unlike our other simulations, the motion-planning "agent" in this case is not Madeline, but instead a jellyfish, so we do not need to worry about player input interfering with the construction. Also because of this, we do not need spinners, jumpthroughs, or crumble blocks in this construction.

Theorem 4.1. ZERO-PLAYER CELESTE *with jellyfish, springs, and move blocks is PSPACE-complete.*

Proof. We reduce from zero-player motion planning with the set-up switch/set-down switch. Figure 10 shows the simulation of this gadget.

The state of the gadget is determined by which side of the central chamber a jellyfish rests in. A new jellyfish entering either of the input locations hits the spring on the side of the corresponding move block, activating it. The move block pushes the original jellyfish out of the gadget through the corresponding output location. Meanwhile the new jellyfish bounces off the spring on the outer wall and is propelled towards the middle of the gadget,
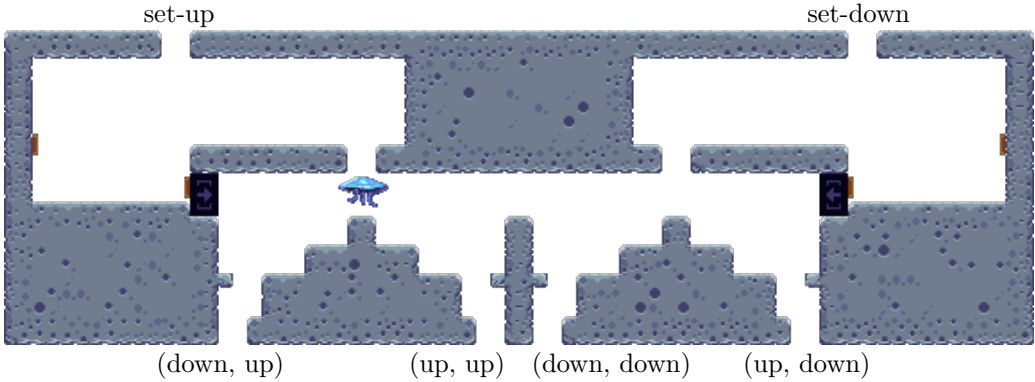
FIGURE 10. A set-up switch/set-down switch constructed with a jelly-fish, springs, and move blocks. Currently in the "up" state.

where it falls through the hole into one of the central locations, setting the state of the gadget.

It remains to show how to combine these gadgets together in a network. Figure 11 shows how to route a jellyfish along predetermined paths to implement "hallway" connections between gadgets, including how to merge the output locations of multiple gadgets into a single input location of another gadget. In our construction, only one jellyfish at a time is outside of a gadget, and that jellyfish represents the agent (though *which* jellyfish represents the agent varies over time).

We also need a final gadget at the destination location, so that Madeline reaches her destination without player inputs if and only if a jellyfish ends up at this gadget. Figure 12 gives one such gadget. ∎

Corollary 4.2. CELESTE *with jellyfish, springs, and move blocks is PSPACE-complete.*

Proof. Using an alternate final gadget, we can instead trap Madeline beneath a move block until a jellyfish enters the gadget, as shown in Figure 13. (Recall that move blocks cannot be triggered from below.) ∎

5. CONCLUSION

We have shown that five variants of Celeste are PSPACE-hard to navigate by reducing from motion-planning problems, as summarized in Table 1. We conclude with a discussion of open questions.

Open Problem 1. What other subsets of Celeste mechanics suffice for PSPACE-hardness?

We can consider strict subsets of the mechanics already shown hard, or consider other incomparable subsets, to fill in Table 1. Alternatively, we could consider other mechanics than those analyzed here; see [10, 11] for longer lists than Section 2.

Of particular note is that Kevin blocks maintain a potentially unbounded stack of locations, which could store more than a polynomial amount of information. However, it is difficult to make use of this information because they are always unwinding towards their initial states when not interacted with.
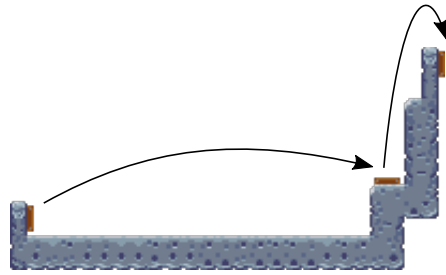
(A) Modular hallway piece. If the third spring is omitted, then jellyfish will fall straight down after traversing the hallway piece, allowing hallways to turn downwards or connect to gadgets.
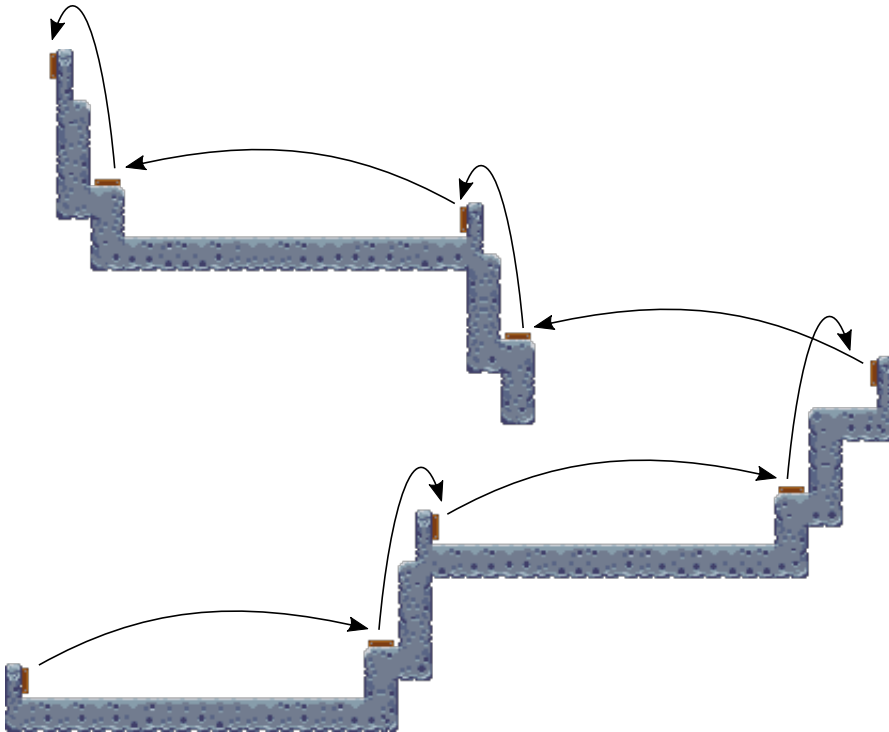
(B) A hallway that turns around, made from four hallway pieces
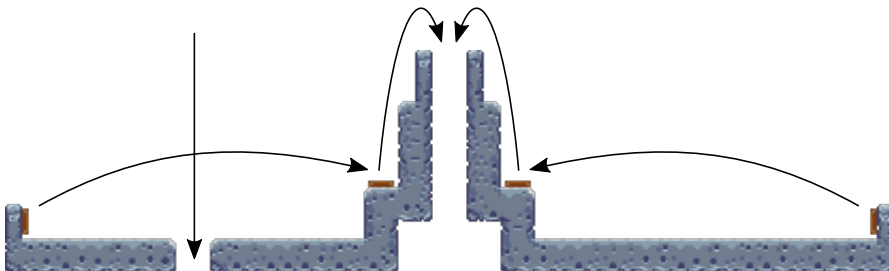
(C) Hallways can merge together or cross over each other.

FIGURE 11. The various types of hallway connections needed to route jellyfish between gadgets.

FIGURE 12. The final gadget. Without player inputs, Madeline reaches her destination (by her car) if and only if a jellyfish enters this gadget.

FIGURE 13. An alternate final gadget. Madeline is able to reach her destination (with player inputs) if and only if a jellyfish enters this gadget.

Open Problem 2. Is CELESTE with Kevin blocks contained in PSPACE?

In our reduction to CELESTE with Kevin blocks (Section 3.4), we constructed a self-closing door gadget which starts out closed. We showed that motion planning over a planar network of initially closed self-closing doors is PSPACE-complete.

Open Problem 3. How does restricting various gadgets to particular initial states affect the complexity of the corresponding motion-planning problems?

---

[11]https://github.com/CelestialCartographers/Ahorn

this excellent tool. We additionally thank the Celeste speedrunning, modding, and Tool-Assisted Speedrunning community for extensively researching Celeste's mechanics.

Finally, we thank Maddy Thorson, Noel Berry, and the rest of the development team for creating Celeste, a difficult and wonderful experience in many ways.

References

[1] Christopher Grant. The Game Awards 2018: Here are all of the winners. *Polygon*, December 2018.

[2] Tom Marks. Inside EXOK Games: The brand new studio that's already sold a million copies. *IGN*, March 2020.

[3] Zeeshan Ahmed, Alapan Chaudhuri, Kunwar Grover, Ashwin Rao, Kushagra Garg, and Pulak Malhotra. Classifying Celeste as NP complete. In *Proceedings of the 9th International Conference on Foundations of Computer Science & Technology*, Chennai, India, November 2022. arXiv:2012.07678.

[4] Greg Aloupis, Erik D. Demaine, Alan Guo, and Giovanni Viglietta. Classic Nintendo games are (computationally) hard. *Theoretical Computer Science*, 586:135–160, 2015.

[5] Giovanni Viglietta. Gaming is a hard job, but someone has to do it! *Theory of Computing Systems*, 54(4):595–621, 2014.

[6] Joshua Ani, Jeffrey Bosboom, Erik D. Demaine, Yevhenii Diomidov, Dylan Hendrickson, and Jayson Lynch. Walking through doors is hard, even without staircases: Proving PSPACE-hardness via planar assemblies of door gadgets. In *Proceedings of the 10th International Conference on Fun with Algorithms (FUN 2020)*, pages 3:1–3:23, La Maddalena, Italy, September 2020.

[7] Giovanni Viglietta. Lemmings is PSPACE-complete. *Theoretical Computer Science*, 586:120–134, 2015.

[8] Joshua Ani, Erik D. Demaine, Dylan H. Hendrickson, and Jayson Lynch. Trains, games, and complexity: 0/1/2-player motion planning through input/output gadgets. In *Proceedings of the 16th International Conference and Workshops on Algorithms and Computation (WALCOM 2022)*, Jember, Indonesia, 2022. arXiv:2005.03192.

[9] Walter J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177–192, 1970.

[10] Fandom: Celeste Wiki. Objects. https://celestegame.fandom.com/wiki/Objects.

[11] Celeste Wiki. Mechanics. https://celeste.ink/wiki/Mechanics.