

Difficulty-Based Generation of Paperfolding Puzzles

Stiliyan Nanovski
Delft University of Technology
Delft, Netherlands
stilijan9@gmail.com

Mrinal Dhume
Delft University of Technology
Delft, Netherlands
m.dhume@tudelft.nl

Rafael Bidarra
Computer Graphics and Visualization
Group
Delft University of Technology
Delft, Netherlands
R.Bidarra@tudelft.nl

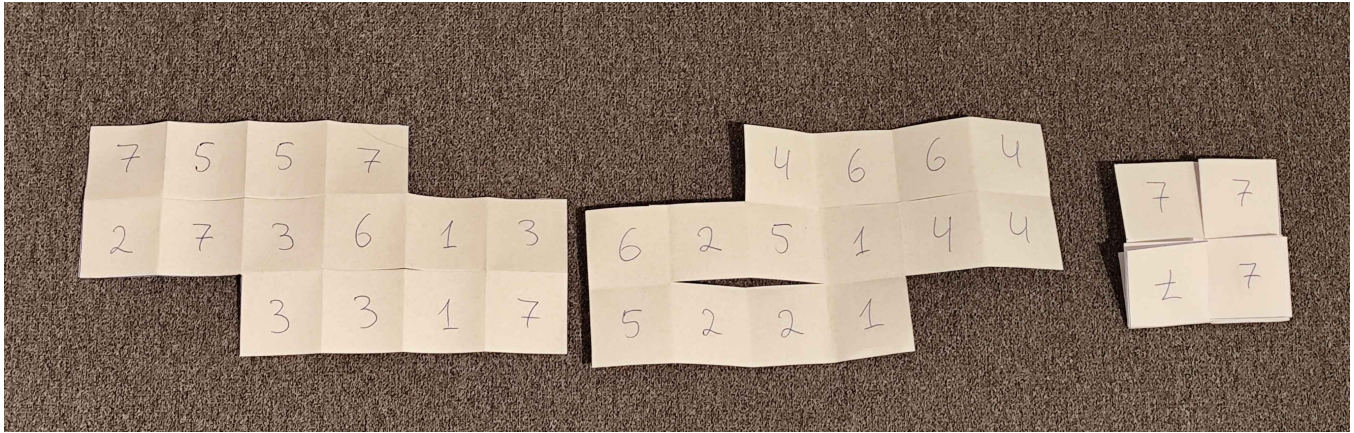


Figure 1: Example puzzle we generated, with both sides of the paper having numbers.

Abstract

Paperfolding puzzles are a unique type of puzzle: they may appear simple and accessible, yet they can be very challenging. Inspired by an existing puzzle [4, 6], based on a square grid and containing slits, we developed a generalized model for such puzzles, with arbitrary shapes and slits. We present that model, for which we implemented a puzzle generator that models paperfolding using constraint solving, and derives the folded states for a given paper shape and size. We also discuss a difficulty metric for each puzzle layout generated, based on the various types of folds it requires. This allows for very customized puzzle generation, for both the target audience, paper shape and type of content per cell grid. Our generator found several new types of folds that are only possible because of the slits, which make the puzzles more varied, interesting and challenging. We conclude with a discussion on some features of the puzzle space for some simple formats.

CCS Concepts

• Computing methodologies → Modeling methodologies.

Keywords

origami, puzzles, constraint solving, PCG, level difficulty

ACM Reference Format:

Stiliyan Nanovski, Mrinal Dhume, and Rafael Bidarra. 2026. Difficulty-Based Generation of Paperfolding Puzzles. In *Foundations of Digital Games (FDG '26)*, August 10–13, 2026, Copenhagen, Denmark. ACM, New York, NY, USA, 10 pages. <https://doi.org/10.1145/3815598.3815696>

1 Introduction

Paperfolding puzzles revolve around finding a way to fold a piece of paper into a certain shape. They have a wide range of variations, from folding intricate 3D objects, as in traditional origami, to creating a specific 2D pattern [7]. These types of puzzles are readily accessible to anyone, as they require only paper and other household items to make and can be understood by both children and adults. They can be used both recreationally and as a way to improve one's problem-solving skills, making them valuable to explore.

We wondered how procedural generation could help create interesting paperfolding puzzles. There are not many instances of procedural generation being used for analogue puzzles [3]; instead, they are usually manually designed. We found inspiration in Chris Harrow's number puzzle [6], as well as in Erik and Martin Demaine's 2017 CSAIL Annual Meeting puzzle [4], whose rules are shown in Figure 2.

However, we propose a more generalized version of those puzzles. The paper grid can be of any size and shape, including non-rectangular shapes, and we fill the grid on the front and back sides of the paper. We also use numbers instead of pictures: the same number can be present on both sides of the paper, which allows for



This work is licensed under a Creative Commons Attribution 4.0 International License. *FDG '26, Copenhagen, Denmark*
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2495-4/26/08
<https://doi.org/10.1145/3815598.3815696>

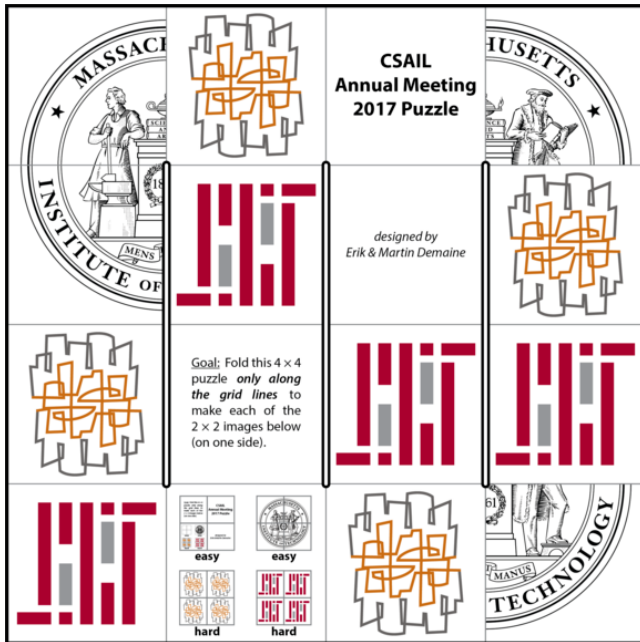


Figure 2: Puzzle by Erik and Martin Demaine [4]. The goal is to fold the paper into a 2×2 paper once for each pattern (Ruleset, MIT seal, CSAIL logo, MIT logo), such that only that pattern is visible on top. The paper has 3 vertical slits shown by the thin lines and can only be folded along the grid lines.

more possible patterns to be created and a wider variety of them, such as that in Figure 1.

What makes this type of puzzle especially interesting is the presence of the slits. Because of them, new types of folds can be performed that would otherwise be impossible, such as the one in Figure 3. This slight modification to a regular piece of paper adds a significant amount of new possible patterns that can be folded into a 2×2 square. As there is significantly less work on origami with slits compared to regular origami, it is worthwhile to explore the possibilities that slits open for making puzzles.

However, finding patterns that can be folded into a 2×2 square is not straightforward. Even when we know that a certain pattern can be folded, finding the sequence of steps for that pattern is also challenging. Since this type of puzzle allows for a wide variety of folds to be performed, they would all need to be described in some way. Moreover, when the steps of the solution are not present, it also makes it hard to define how difficult the puzzle really is, as the path to the solution is often determinant for its difficulty. These challenges make finding a good way to measure the difficulty of our paperfolding puzzles one of our main goals.

Our approach to generating the puzzles is based on deriving all possible folded states that can be reached with the paper, then using them to assign the patterns on the puzzle. This generation is done using a constraint-solving model that can derive the possible states. From those states, we check which can be reached by using a limited set of folds. Finally, we developed a difficulty metric to

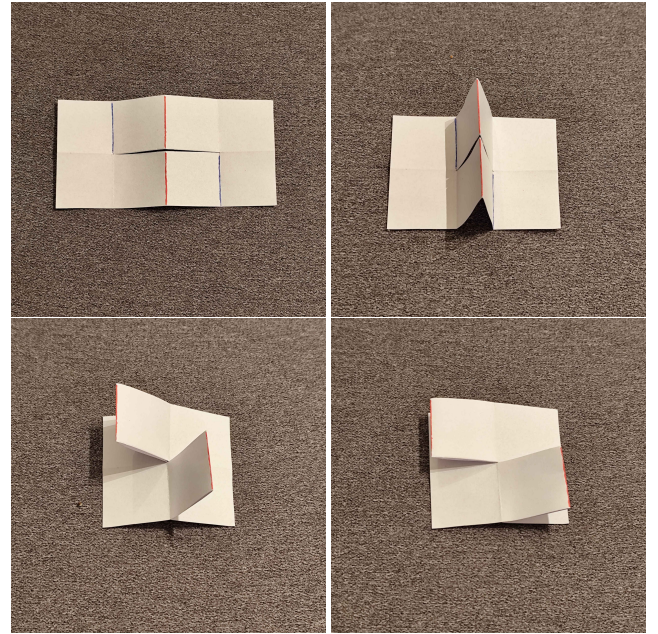


Figure 3: Example fold allowed due to slits, where two segments are folded in opposite directions, and the steps to perform the fold. This can be done because the physical connection that is normally there is not present.

use on the reachable folded states to evaluate a difficulty score for the solution.

2 Related Work

This work builds upon research on two domains, origami and puzzle difficulty assessment, which are here briefly surveyed.

2.1 Origami Research

Our work is closely related to the study of origami, so we first introduce its most commonly used terms. A *crease* is a line segment on which the paper has been folded. The fold performed on the crease can either be a *mountain fold* or a *valley fold*, depending on the direction the paper is folded. All creases on a piece of paper form a *crease pattern*, and a *mountain-valley pattern* (MV pattern) is the assignment of each crease in the crease pattern. A *folded state* is any state in which the origami is folded, according to its MV pattern. If all creases have been completely folded to 180° , then it is also in a *flat-folded state*. Figure 3 shows the MV pattern drawn on the paper, with mountain folds in red and valley folds in blue. The image on the bottom right shows the folded state, in which all four creases have been folded flat. In the rest of the paper, when we refer to a folded state, we always mean a flat-folded state, as those are the states that concern our puzzles. A *folding motion* is the continuous motion through which you go from one folded state to another.

A problem similar in nature to our puzzles is the map folding problem. It asks how many ways a rectangular $N \times M$ piece of paper can be folded into a 1×1 square. The number increases very fast with

the size of the paper; for square inputs $N \times N$ only the first seven values are known. For some specific sizes ($M \leq 2$), polynomial time algorithms exist [1, 9]. However, they use techniques specific to those sizes that cannot be extended to larger grids. In the general case, map folding is NP-hard and algorithms that count the number of ways to fold the map utilise the fact that all lines must be creased, which makes them less applicable to our puzzles. Some variations of the problem can be solved in polynomial time, such as restricting it to only simple folds [1]. However, we do not want to impose such restrictions on our puzzles, as the other unique folds are one of the intriguing factors about them.

It is also possible to generate just the crease patterns without the layer order, as done by Matsukawa et al. [8]. However, their method of using a depth-first search has to go through all possible shapes of origami, while our puzzles require a specific final size (2×2), making it inefficient for our purposes. It would also require additional work to find all ways in which the layers of the origami can be ordered, as a single crease pattern can have multiple ways to fold it and order the layers of the squares.

To the best of our knowledge, combining origami with slits has not been explored for computational origami.

2.2 Procedural Puzzle Generation

Puzzles as a target for procedural generation have been researched for a long time [3]. There are a variety of ways that puzzles can be procedurally generated, such as modelling the puzzles as constraints [10] or working backwards from a solved state [12]. A less common, but still useful approach is Exhaustive Procedural Content Generation (EPCG), which tries to (semi)-exhaustively enumerate the state space of a game's domain [11]. This approach is often better suited for puzzle games, as their states have less elements that do not affect the gameplay, but increase the domain size.

2.3 Puzzle Difficulty

What makes a puzzle difficult varies greatly between different types of puzzles and audiences. Often, a specific instance of a puzzle can be most precisely tested with a player study. However, we would also like to have an automatic method to grade an arbitrary instance. A prominent way of doing this is to choose variables that represent an aspect of the difficulty and combine them in a function for a final score [14]. However, apart from the original puzzles [4, 6], we have not found any other similar types of origami puzzles to base our difficulty metric on. Out of the many already existing metrics, we looked at those that are commonly seen across many puzzle types and can be applied to ours.

One metric that correlates well with difficulty is the difficulty of individual steps used in a solution. If solving a puzzle requires the player to execute a difficult step, then this indicates that the puzzle itself is difficult. This principle has been used in the generation of Sudoku puzzles [2].

Another common metric is the length of the solution, where more steps required correlate to more difficult puzzles. On its own, this metric is not always a good predictor of difficulty, as a puzzle can have many simple steps that are easy to spot, making it easier than a puzzle with a single, very hard-to-spot step. Nevertheless, it can still be useful when combined with other metrics and can be

applied to our origami puzzles. This metric is often used in maze-like puzzles, where it is often correlated with time spent solving as well [13].

3 Methodology

We first introduce the definition of our paperfolding puzzle, and then describe the constraint-based model used to derive its folded states.

3.1 Puzzle Definition

Here we formally define our type of puzzle and its rules:

- A puzzle is composed of a piece of paper of any form based on a square grid, with slits optionally present along any of the grid lines, as long as the paper is still connected. We refer to the squares in the grid as *faces*.
- Each face has a number on both sides, such that each number is present exactly 4 times across the whole paper.
- For each number, the player has to find a way to fold the paper into a 2×2 square, such that only the 4 copies of that number are visible on the top. The orientation of the number does not have to be the same. The 4 sides of the faces with a given number together form a *pattern*.
- The paper can only be folded along the grid lines, i.e. its final crease pattern needs to be along the grid lines.

In this paper, when showcasing both sides of a puzzle, the back side is always shown by flipping the front side horizontally. An example puzzle and the step-by-step solution to one of its patterns are shown in Figure 4.

3.2 Folded State Derivation

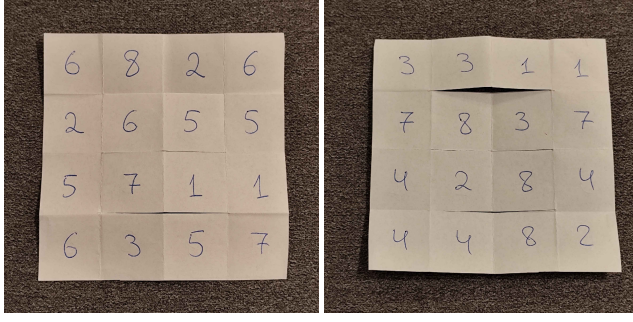
We wrote a constraint-solving model using Minizinc¹ and ran it using Gecode 6.3.0 and Python 3.10.5 to explore and derive the folded states of our puzzles. We do this because the solution to a given pattern is the folded state that shows the faces of the pattern on the top or bottom.

3.2.1 Paper Modelling. To model the flat shape of the paper and its slits, it is sufficient to define the connections between its faces. A *connection* is the physical edge that joins two adjacent faces. We refer to the original unfolded state of the paper, including its connections and slits, as the *shape* of the paper. We formally define the shape as:

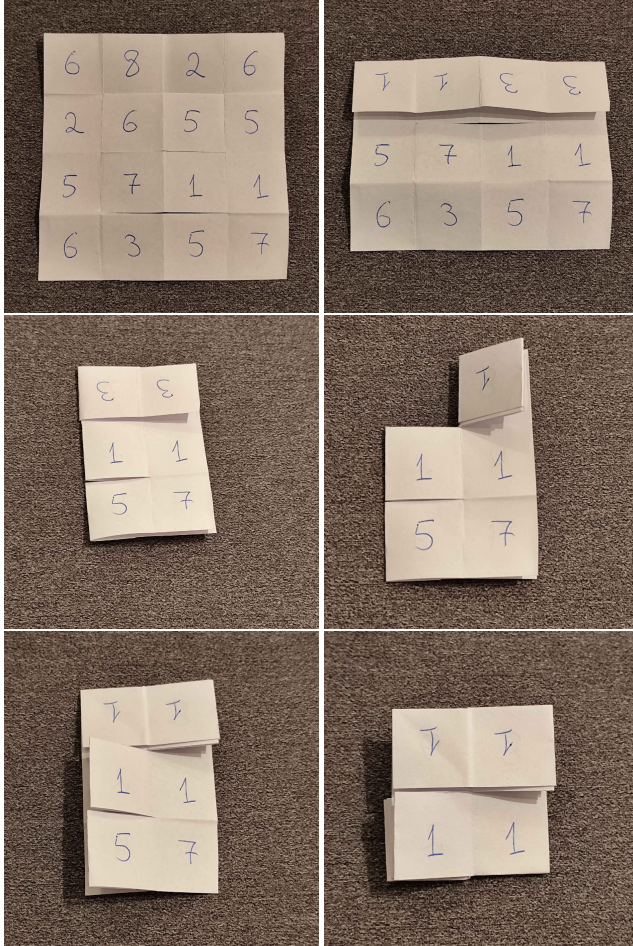
- The set of faces F on the paper.
- The set of connections C between those faces on the paper.
- Each connection is defined as $c_i = (f_a, d, f_b)$, where $d \in \{\text{North, South, West, East}\}$ is a direction, and $f_a, f_b \in F$ are two adjacent faces, such that f_b is positioned relative to f_a in direction d .

Because we model the connections between faces, this model can express any grid-based shape of the paper and any slits, and not just rectangular-shaped papers. This definition serves as a direct input to the Minizinc model.

¹<https://www.minizinc.org/>



(a) Example puzzle. The left figure is the front face. The right figure is the back face, after flipping the paper horizontally.



(b) Solution to the 1's pattern. The folds are performed from left-to-right and top-to-bottom.

Figure 4: An example 4×4 puzzle with slits.

3.2.2 Folded State Constraints. From the shape of the paper, we want to find all folded states that result in a 2×2 square. We model this by looking for a mapping from each face to its physical position

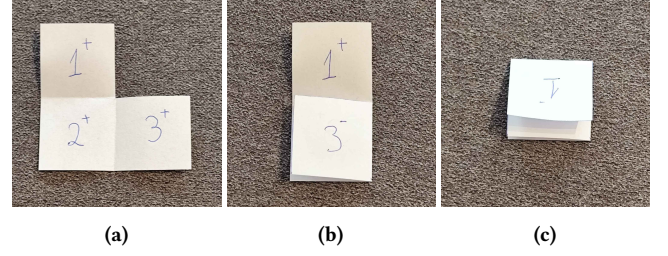


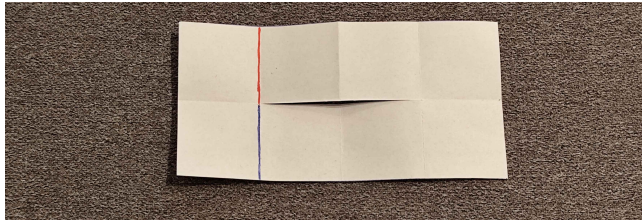
Figure 5: How r_i and s_i change with different folds. (a) All faces have $(r_i, s_i) = (0, +1)$. (b) Face 3 is folded to the left, by folding along the vertical crease. $(r_3, s_3) = (0, -1)$, since the orientation remains unchanged and we see the bottom of the face f_3 . (c) Face 1 is folded down by folding along the horizontal crease. $(r_1, s_1) = (2, -1)$, since the 1 drawn on the paper is upside-down and the back of the face is visible on top.

in the folded state, then ensuring that the topology of the paper is preserved and that there are no self-intersections.

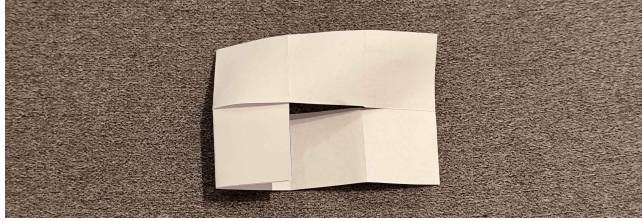
We model each face f_i as 5-tuple $f_i = (x_i, y_i, l_i, r_i, s_i)$. $(x_i, y_i) \in \mathbb{N}^2$ are the coordinates in 2D space where the face is placed. Because all folds are done across a square grid, we represent these coordinates as a pair of natural numbers. The layer $l_i \in \mathbb{N}$ determines how different faces with the same coordinates are stacked, where 1 is the layer of the face on the bottom of the stack. The rotation of a face $r_i \in \{0, 2\}$ is the number of 90° counter-clockwise rotations applied from its original orientation. Because all creases are either horizontal or vertical, faces always end up with either 0 or 2 such rotations. The side $s_i \in \{-1, +1\}$ shows which side of the face is visible from above, with $+1$ being the front (positive) side of the paper and -1 being the back (negative) side of the paper. Figure 5 shows the interaction between r_i and s_i through an example.

We can now define the constraints of the model as follows:

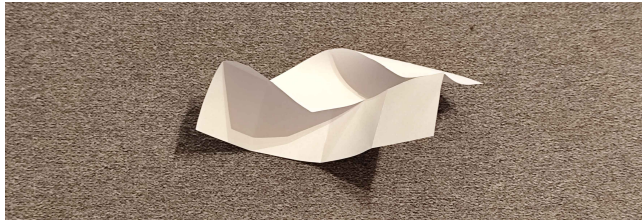
- **Connectivity constraint** - If a connection is a crease, then its two faces must be stacked on top of each other and have the same coordinates. Otherwise, their coordinates must be adjacent.
- **Orientation constraint** - Every connection's two faces must be oriented such that they are still joined by the connection.
- **Stack constraint** - Faces that have the same coordinates must be in different layers.
- **Self-intersection constraint** - The paper may not intersect itself for the folded state to exist. Matsukawa et al. [8] documented three ways in which the paper can self-intersect by looking at how two connections and their faces can be positioned.
- **End shape constraint** - The folded state must result in a 2×2 square. That is, there must be at least one face with coordinates $(1, 1)$, $(1, 2)$, $(2, 1)$ and $(2, 2)$, and these must be the only coordinates that faces can have. This ensures that the folded state results in a 2×2 square. Without this, we can instead generate all folded states, regardless of their end shape. This constraint can be easily adjusted depending on the desired final puzzle shape.



(a) Crease pattern of the impossible folded state.



(b) How the folded state has to look. To make this, we cut and taped back the paper.



(c) Attempt to unfold the paper back into its original state.

Figure 6: Folded state of a 4×2 puzzle shape with a slit in the middle that is topologically different from the unfolded state.

The constraints above are sufficient to describe a state that does not self-intersect. However, we found that some results might have a different topology from the unfolded state, such as the example in Figure 6. To detect and prevent this, we need one more constraint. Because this is too complex to describe in Minizinc, we check for it separately in Python.

For this topology constraint, we use the concept of a *linking number* from knot theory. The linking number between two closed three-dimensional curves is the number of times each curve winds around the other. It is the only invariant that describes how two curves are linked to each other. It can be calculated by looking at the crossings of the two curves on a link diagram and either adding or subtracting 1, depending on the orientation of the curves and which one is on top.

First, we take all cycles of faces in the shape of the paper. Then, for each cycle, we can draw two curves around the loop that do not intersect each other, as shown in Figure 7. Because they do not intersect each other, in the unfolded state, those curves have a linking number of 0. Finally, in a given folded state, we can recalculate the linking number again, and if it is different from 0, then the folded state does not have the original topology. Because our folded state is two-dimensional, but differentiates the layers between faces in

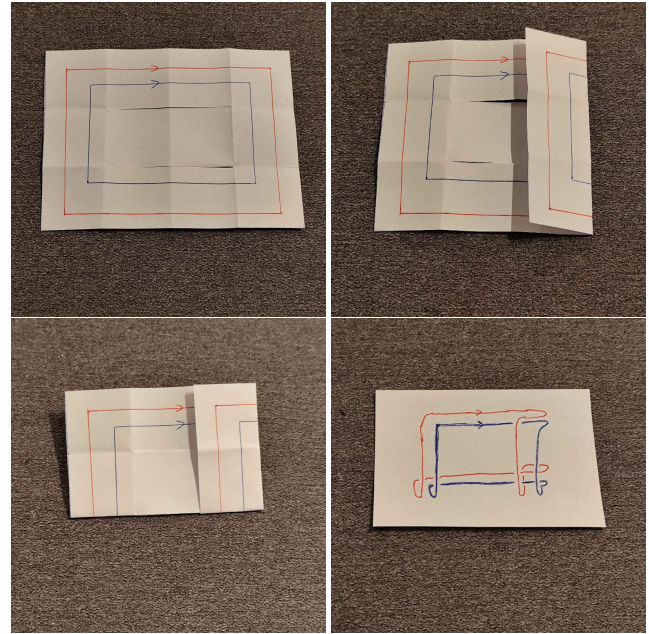


Figure 7: Example of the curves used for the linking number constraint, a folded state and its resulting linking diagram for the large cycle of faces.

the same spot, we can make a link diagram of the two curves to use for calculating the linking number.

In all our experiments, we have always found that these constraints are sufficient for a folded state to exist. However, we do not have a formal proof of it. The proof of a folding motion [5] is not applicable for non-simple polygons, such as our puzzles, which have slits in them.

From the information we have of the folded state, we can infer its MV pattern. For each connection that is a crease, whether it is a Mountain or a Valley fold, it depends on where the faces are facing. If the positive sides of the two faces are facing away from each other, then it is a Mountain fold. Otherwise, it is a Valley fold.

From the list of possible folded states, it is trivial to find what patterns can be solved. From each folded state, we obtain two end patterns, one from the faces visible on top and one from those visible on the bottom. Knowing all solvable patterns, we can easily generate a full puzzle. However, to make it more interesting, its patterns should be well-balanced in terms of their difficulty.

3.3 Difficulty Grading

A given pattern can have multiple folded states that solve it, which have to be taken into account when grading its difficulty. To address this, we first grade each folded state with a difficulty, then grade a pattern based on the folded states that reach it. For both, the difficulty is a score between 0 and 1, with 0 being the easiest and 1 the hardest.

3.3.1 Folded State Grading. To be able to grade a specific folded state more effectively, we first try to find a sequence of folds that

reaches it. We limit ourselves to three types of folds that we identified in many solutions to patterns, while also being more straightforward to define. They are as follows:

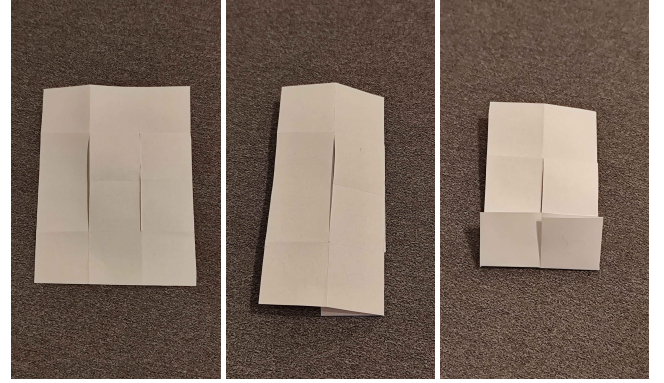
- **Simple folds** - Simple folds, specifically *some-layer simple folds*, take any number of layers and fold them 180 degrees. This is the easiest and most natural type of fold. An example of such a fold can be found in Figure 8a.
- **Pop folds** - This fold is performed in a row of 4 faces, as shown in Figure 8b, where the middle 2 faces are only connected to the faces in the row. In the initial state, the middle connection must already be folded. Then, we can "pop" it on the other side of the "V" shape that is formed to form a "W" shape. This is possible due to the flexibility of the paper. The goal of these folds is to allow more complex folds, such as the one shown in Figure 3, to be possible through smaller steps.
- **Inside-out folds** - This fold is performed on a "U" shape formed out of faces, as shown in Figure 8c. Faces 2, 3, 4 and 5 may have no other connections. In the initial state, connections 2 and 4 must be folded the same way as shown. Then, faces 3 and 4 can be tucked forward through the gap, until they reappear below faces 2 and 5. This fold can also be performed with more faces between 2 and 5, making a wider "U" shape.

To find the steps of a solution for a pattern, we construct a graph G , with the folded states as vertices, and folds as edges. That is, two folded states are joined by an edge if there is a fold from one state to the other. To find the steps to reach a folded state, we just need to find the path from the unfolded state.

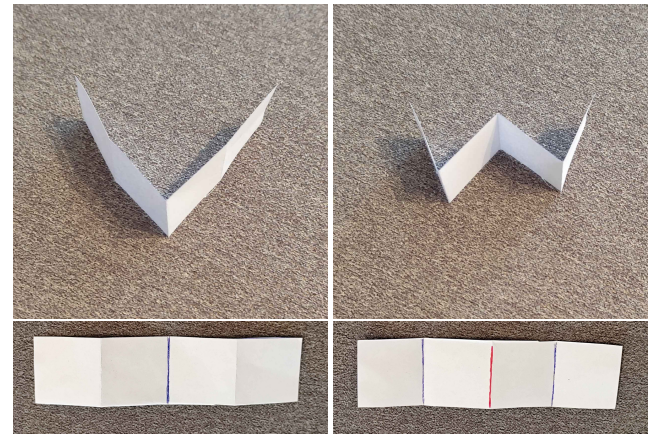
For the folded states that can be reached by a sequence of folds, we grade their difficulty based on the hardest fold they require to be reached. We assume that simple folds are the easiest to perform, followed by pop folds, followed by inside-out folds as the hardest. For example, if a folded state can be reached by using only simple folds, it should have a lower difficulty than a state that can only be reached through an inside fold or through a pop fold. We specify the *level* of difficulty of the folds, with simple, pop and inside-out folds being levels 1, 2 and 3, respectively. Lower level means the fold is easier to perform.

To find the minimum required level of folds to reach a folded state, we use subgraphs of G with edges limited up to a certain level. We define three subgraphs, $G_1 \subseteq G_2 \subseteq G_3 = G$, where G_i has the same vertices as G and only edges of folds up to level i . We can search G_1 using breadth-first search to find which folded states are reachable using only simple folds, and also find the shortest path to them in terms of the number of folds. Then, we repeat this with G_2 and G_3 to find which states can be reached with their respective subsets of folds. Finally, to use this in our difficulty metric, we define the *level* of a folded state l to be the smallest level, such that the folded state can be reached in G_l . For folded states that do not have a path in any of the subgraphs, we assume they require an especially difficult fold to be solved. For convenience, we assign those to be level 4, but do not use them in our difficulty metric, as they are automatically assigned the highest difficulty score 1.

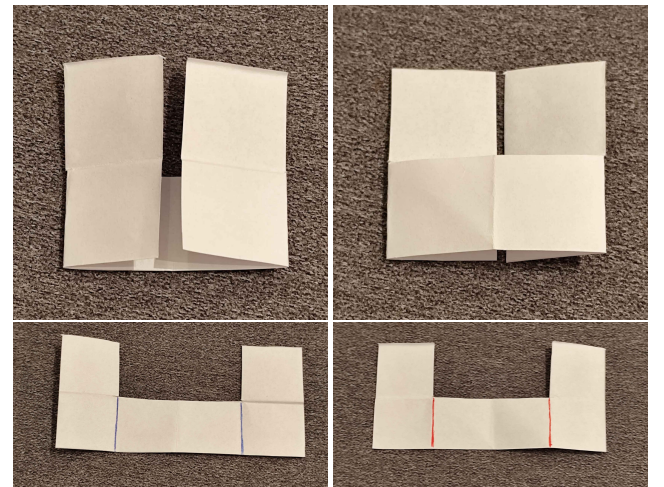
We now define the final difficulty of the folded state, d_{fs} , as follows:



(a) 4×3 paper with two simple folds performed one after another.



(b) Before and after performing a pop fold. Above is how the fold looks. Below is how the MV pattern changes.



(c) Before and after performing an inside-out fold. Above is how the fold looks. Below is how the MV pattern changes.

Figure 8: Three different types of folds that we categorise with increasing levels.

$$d_{fs} = \frac{l - \frac{1}{|p|}}{3}$$

where l is the level and $|p|$ is the length of the shortest path p with folds up to level l . The term $-1/|p|$ correlates longer fold sequences with greater difficulty, while keeping the level as the main term to decide the difficulty. This way, we add more variety to the metric. The division normalises the difficulty to be between 0 and 1, and can be adjusted depending on how many types of folds are used.

3.3.2 Pattern Grading. When grading a pattern, we only take into account the folded states that solve it with the minimal level. If it can be solved with simpler folds, then there is no reason to use more complex ones. The number of solutions available should affect the difficulty of a pattern, as the more distinct solutions there are, the easier it should be to find any one of them. Also, if any of the solutions is significantly easier than the rest, it should make the pattern easier overall.

To take these features into account, we calculate the difficulty of the pattern as a weighted sum of multiple scores. Based on the list of folded states l with the minimal level, we calculate the following scores:

- The minimum difficulty, l_{min} .
- The mean difficulty, l_{mean} .
- The inverse length of the list, $\frac{1}{|l|}$.

The final difficulty of the pattern, d_{sp} , can then be calculated as:

$$d_{sp} = \alpha_1 l_{min} + \alpha_2 l_{mean} + \alpha_3 \frac{1}{|l|}$$

where $\alpha_1, \alpha_2, \alpha_3$ are the weights and add up to 1. For our calculations, the weights $(\alpha_1, \alpha_2, \alpha_3) = (\frac{10}{14}, \frac{3}{14}, \frac{1}{14})$ are used, as we found the most important factor of the difficulty is the easiest solution.

3.4 Complete Puzzle Generation

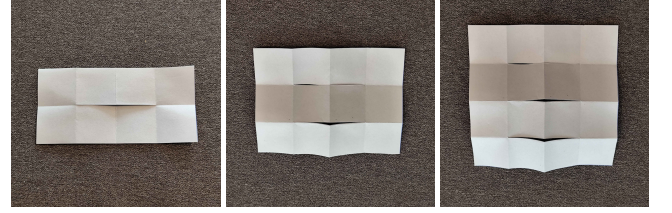
To generate a complete puzzle, we have to find a clique of patterns that do not overlap with each other. We can easily create a graph where two patterns have an edge between them if they do not share any faces, or if they do, they must each use a different side of the face. When searching for a clique, we can specify how many patterns of each difficulty level to include, depending on the desired difficulty of the puzzle. Once the patterns have been decided, we assign them their numbers in ascending order of difficulty, with 1 being the easiest, and the puzzle has been generated.

We developed a software system to combine all of the steps to produce complete puzzles². It takes as input a text file that describes the shape of the paper. Then, we run our Minizinc model to obtain all valid folded states, followed by filtering out those that have a different topology from the unfolded state. Those states then have a difficulty and level assigned, which are then used to grade the patterns that can be solved. Those patterns are obtained from the set of folded states. Finally, a full puzzle is generated by searching for a clique in the graph of patterns. The user can specify the desired number of patterns from each level to be present in the puzzle.

²<https://github.com/steliotop/OrigamiPuzzleGenerator>



(a) The shapes 4×2 , 4×3 and 4×4 without slits.



(b) The shapes 4×2 , 4×3 and 4×4 with slits.

Figure 9: The puzzle shapes for which we ran experiments. From top to bottom, then left to right: 4×2 , 4×3 , 4×4 , 4×2 with slits, 4×3 with slits and 4×4 with slits.

4 Results

Using our software, we derived the folded states for papers of sizes 4×2 , 4×3 , 4×4 , both without slits and with horizontal slits going through all middle lines, as shown in Figure 9. We ran them on a Laptop equipped with an AMD Ryzen 5 5600H processor with 6 CPU cores and 12 logical processors. The Gecode solver was allocated 8 logical processors. For each of them, we obtained how many folded states are reachable and how many there are of each difficulty level. The data for the folded states that result in a 2×2 square is shown in Table 1. We also timed the stages of our system, with those results shown in Table 2. The time performance of the final step of generating a full puzzle has been omitted, as it takes a negligible amount of time.

5 Discussion

Our method is successful in generating complete puzzles. For the shapes 4×3 and 4×4 with slits, there is a sufficient number of patterns in each level to create puzzles with any desired distribution of levels. For the shapes without any slits, the pop and inside-out folds are not applicable, so they do not have any level 2 or 3 patterns.

It is clear from the data that using slits creates a significant amount of newly reachable folded states. Each removed connection adds a new degree of freedom to the constraint model to reach new states, increasing the total number of states by about 2-3 times. Increasing the number of faces also naturally adds significantly more reachable states.

For the shapes with the horizontal slits, the inclusion of the pop and inside-out folds allowed us to find the sequence of folds of many new folded states. Using them together with simple folds, about 2-3 times more folded states can be reached. However, a significant amount of other (level 4) states still cannot be reached, and the data suggests, that that number grows faster with the size of the paper. For larger-sized shapes, the pop and inside-out folds need to

Shape	F	C	# 2×2 FSs	# Reachable 2×2 FSs	# Patterns	# Patterns Excluding Reflections
4×2	8	10	14	14 (14, 0, 0, 0)	12 (12, 0, 0, 0)	4 (4, 0, 0, 0)
4×2 slits	8	8	82	32 (18, 8, 2, 4)	36 (16, 8, 4, 8)	8 (5, 1, 1, 1)
4×3	12	17	192	192 (176, 0, 0, 16)	108 (108, 0, 0, 0)	18 (18, 0, 0, 0)
4×3 slits	12	13	6512	1064 (288, 460, 60, 256)	360 (152, 56, 48, 104)	51 (24, 7, 7, 13)
4×4	16	24	1768	1768 (1352, 0, 0, 416)	460 (448, 0, 0, 12)	37 (35, 0, 0, 2)
4×4 slits	16	18	434912	29844 (3172, 11636, 642, 14394)	1952 (712, 216, 148, 876)	264 (102, 27, 20, 115)

Table 1: Results from deriving the 2×2 folded states (FS) of different shapes of paper. The four numbers in brackets show the data split amongst difficulty levels (1 to 4). In the last column, we count as one those patterns that are reflections or rotations of each other, as well as the same pattern on the other side of the paper.

Shape	Constraint Model	Topology Constraint	Grading FSs	Counting Patterns	Grading Patterns	Total
4×2	0.954s	0.014s	0.270s	0.002s	< 0.001s	1.240s
4×2 slits	0.805s	0.026s	0.559s	0.007s	0.001s	1.398s
4×3	2.059s	4.485s	4.500s	0.039s	0.001s	11.084s
4×3 slits	11.716s	5.904s	25.839s	0.151s	0.003s	43.613s
4×4	68.854s	467.659s	47.024s	0.448s	0.005s	583.990s
4×4 slits	1509.158s	593.109s	779.671s	3.957s	0.033s	2885.928s

Table 2: The time performance of running our system on each of the paper shapes. For each shape, we list the time spent on running the Minizinc constraint model, verifying the topology of the folded states (FS) from the model, grading the difficulty of those FSs, obtaining the solvable 2×2 patterns and grading their difficulty.

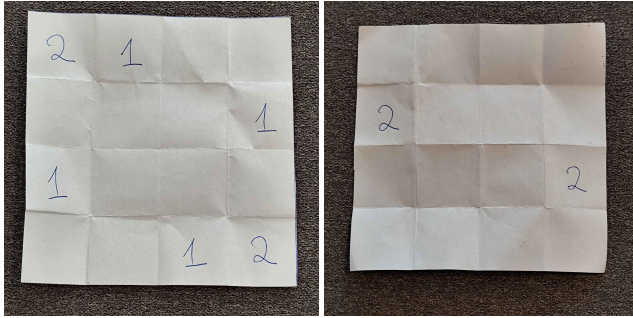


Figure 10: The only two level 4 patterns for the 4×4 shape with no slits, shown as the 1's and 2's pattern, respectively. The left and right figures show the front and back sides of the paper, respectively.

be generalized for longer slits. These folds also only work for slits that are a straight line; for more complicated slits, different types of folds need to be identified and used.

It is interesting to note that out of the shapes without slits, only the 4×4 paper has any level 4 patterns. These two unique patterns are shown in Figure 10. They both require a unique fold as the last step, although once understood, they are not difficult to execute.

Although some level 4 patterns are not too complicated to perform, the majority of them require a fold that is physically difficult to execute. The single level 4 pattern of the 4×2 with slits, can be solved using an inside-out fold, followed by shifting most of the faces to the side, as shown in Figure 11. This and a lot of other level

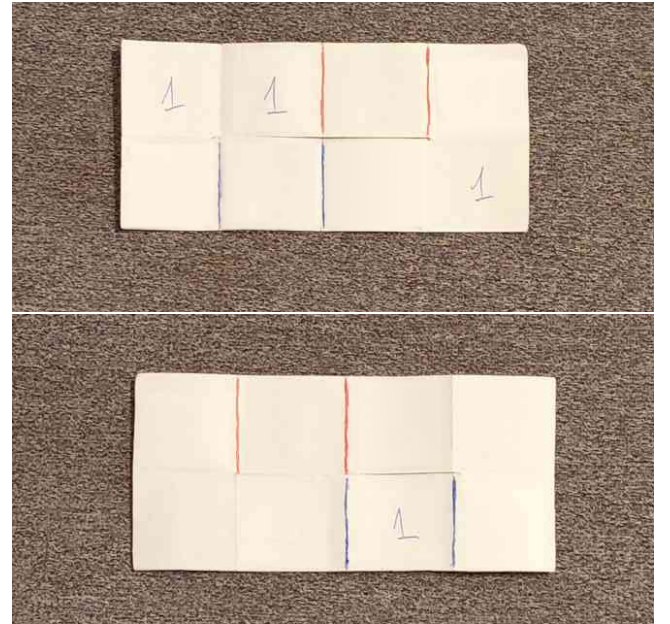


Figure 11: The only level 4 pattern for the 4×2 shape with a slit and the crease pattern that solves it. The top and bottom figures are the front and back sides, respectively.

4 patterns would often feature a face around which the paper wraps around.

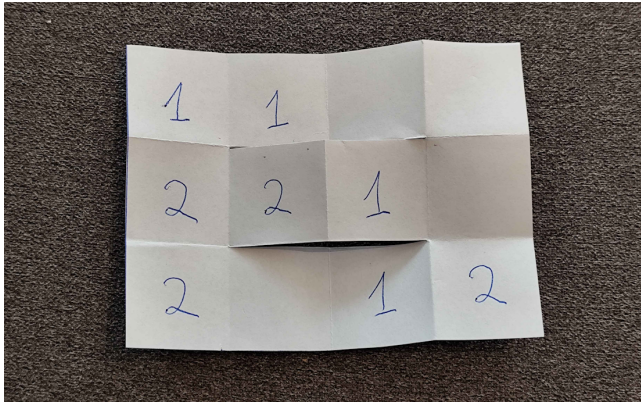


Figure 12: A level 1 and level 2 patterns, of which the level 1 pattern is harder to solve.

We consider that the inside-out fold is already a difficult fold to execute and also hard to discover without prior knowledge of it. For the 7's pattern in Figure 1, we were only able to solve it by knowing the crease pattern and face layers in advance and trying to fold it in one big step. The more complicated folds, including the inside-out fold, would also sometimes start to tear the paper around the ends of the slits when trying to perform the fold.

Our assumption in the difficulty metric that patterns that require a higher level of folds are harder does not always hold. There are patterns that require a pop fold to be solved, but the solution is easy to spot and execute; meanwhile, patterns that are solvable with only simple folds can still be fairly difficult to solve. Examples of such patterns are shown in Figure 12. The number of steps is also not always a good metric for pop folds, as the player would usually perform a single fold that is done in a longer sequence of moves by our graph search.

One of the main limitations of our approach in trying to find the steps to reach a specific folded state is the limited expressiveness of the folds we consider. While the pop and the inside-out fold can be easily generalized for slits of longer lengths, they would not work for slits with more complex shapes, such as a cross-shaped slit. Those slits would require different and more expressive folds, although that would also make the tree search for the sequence of steps take even longer.

The time and memory performance of the model are also a big limiting factor. Our current implementation is unable to enumerate shapes that are bigger than 4×4 , as either they take too long or the solver runs out of RAM before it can finish. We believe this to be a limitation of using a constraint model with a general-purpose solver, over an enumerator that is specifically optimized for our problem. Checking for the topology constraint outside of the model adds additional overhead, whereas if it was a part of the model directly it would significantly reduce the model's runtime, as the output would be smaller. It would also reduce the required memory, as the constraint model has to return all folded states at once to be processed in Python. Although this would improve the performance by some constant factor, the results from Table 2

suggest the required time explodes exponentially and really fast, which a dedicated enumerator would typically not address.

It is interesting to note that the shapes without slits spend proportionally longer on the topology constraint than those with slits, and that to no avail. For example, in the 4×4 shape with slits, grading the folded states takes about 20% longer than applying the topology constraint. However, in the 4×4 shape without slits, applying the topology constraint takes almost 10 times longer than grading the folded states. For the latter, due to there being more connections, there are many more cycles of faces (e.g. every 2×2 square being counted as a loop). But from Table 1, we know that all its folded states are reachable, i.e. none was discarded, which suggests that the constraint model is sufficient for shapes without slits. This makes sense, because if there are no slits or holes in the shape, the shape would be a simple polygon. For simple polygons, we only need to show that the folded state can exist for a folding motion to exist [5], which is what the constraint model shows. Therefore, checking for the topology constraint is unnecessary for simple polygons.

6 Conclusion and Future Work

We successfully defined a generalized model for paperfolding puzzles with slits, and managed to derive their crease patterns. We documented two new interesting types of folds, the pop fold and the inside-out fold, which are detectable in the model. The pop fold allows a significant amount of folded states to be reached through any graph search algorithm, thus finding exact steps on how to reach the folded state. The puzzles created can also be successfully graded with a difficulty score, using a heuristic-based metric. Our system can generate interesting puzzles of varying difficulty, from simple to really difficult, with many puzzles requiring unique and challenging solutions. This, combined with how simple it is to generate a new puzzle, adds a lot of replayability and enjoyment for the player.

For future work, an important part would be a player study to verify the difficulty metric. Our formula and weights are based on empirical data, and the current level classification is likely too inflexible. Including more types of slits in the origami, implementing new folds into the search and further generalizing the existing types would also add more nuance to the metric. A player study would allow the weights to be tuned to real data and, therefore, better reflect the perceived difficulty of the puzzles. A non-exhaustive approach to the puzzle generation would also be valuable, as it could significantly improve the performance and avoid requiring the pre-computation of the possible folded states.

References

- [1] Esther M. Arkin, Michael A. Bender, Erik D. Demaine, Martin L. Demaine, Joseph S. B. Mitchell, Saurabh Sethia, and Steven S. Skiena. 2004. When Can You Fold a Map? *Computational Geometry: Theory and Applications* 29, 1 (September 2004), 23–46. doi:10.1016/J.COMGEO.2004.03.012 Special issue of selected papers from the 10th Annual Fall Workshop on Computational Geometry, 2000..
- [2] T Boothby, L Svec, and T Zhang. 2008. Generating Sudoku puzzles as an inverse problem. *Math. Contest Model* 24 (2008).
- [3] Barbara De Kegel and Mads Haahr. 2019. Procedural puzzle generation: A survey. *IEEE Transactions on Games* 12, 1 (2019), 21–40.
- [4] Erik Demaine and Martin Demaine. 2017. CSAIL Annual Meeting 2017 Puzzle. <https://erikdemaine.org/puzzles/CSAIL2017/>. <https://erikdemaine.org/puzzles/CSAIL2017/> Accessed: 2026-03-30.

- [5] Erik D. Demaine, Satyan L. Devadoss, Joseph S. B. Mitchell, and Joseph O'Rourke. 2004. Continuous foldability of polygonal paper. In *Proceedings of the 16th Canadian Conference on Computational Geometry, CCCG'04, Concordia University, Montréal, Québec, Canada, August 9-11, 2004*. 64–67. <http://www.cccg.ca/proceedings/2004/55.pdf>
- [6] Chris Harrow. 2012. Origami Puzzles. <https://casmusings.wordpress.com/2012/05/03/amazing-number-puzzle/> Accessed: 2026-03-30.
- [7] Tom Hull. [n. d.]. Origami Puzzles. https://origamiusa.org/puzzles_checkerboard. https://origamiusa.org/puzzles_checkerboard Accessed: 2026-03-30.
- [8] Yoshihisa Matsukawa, Yohei Yamamoto, and Jun Mitani. 2017. Enumeration of flat-foldable crease patterns in the square/diagonal grid and their folded shapes. *Journal for Geometry and Graphics* 21, 2 (2017), 169–178.
- [9] Thomas D Morgan et al. 2012. *Map folding*. Ph. D. Dissertation. Massachusetts Institute of Technology.
- [10] Mark J. Nelson and Adam M. Smith. 2016. *ASP with Applications to Mazes and Levels*. Springer International Publishing, Cham, 143–157. doi:10.1007/978-3-319-42716-4_8
- [11] Nathan Sturtevant and Matheus Ota. 2018. Exhaustive and semi-exhaustive procedural content generation. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, Vol. 14. 109–115.
- [12] Joshua Taylor and Ian Parberry. 2011. Procedural generation of sokoban levels. In *Proceedings of the International North American Conference on Intelligent Games and Simulation*. 5–12.
- [13] Elio Valenzuela, Hans Schaa, Nicolas A. Barriga, and Gustavo Patow. 2025. Using search algorithm statistics for assessing maze and puzzle difficulty. *Entertainment Computing* 53 (2025), 100925. doi:10.1016/j.entcom.2025.100925
- [14] Marc van Krevel, Maarten Löffler, and Paul Mutser. 2015. Automated puzzle difficulty estimation. In *2015 IEEE Conference on Computational Intelligence and Games (CIG)*. 415–422. doi:10.1109/CIG.2015.7317913