Using EPCG for Designing a Hexagon Tangram Puzzle

Yazeed Mahmoud[1], **Nathan R. Sturtevant** [1,2]

[1] University of Alberta, Department of Computing Science
[2] Alberta Machine Intelligence Institute
yazeed1@ualberta.ca, nathanst@ualberta.ca

Abstract

Designing and tuning a game is a complex creative process, with a variety of tools have been designed for assisting human designers in this process. But, many existing tools either entirely take the design away from humans, or they only assist in generating content - they cannot be used to create an entire game from scratch. This paper explores how Exhaustive Procedural Content Generation (EPCG) can be integrated as a co-creative partner in the design process. We describe how EPCG was used alongside human designers to create a novel Hexagon Tangram puzzle, from initial conception to puzzle curriculum. EPCG is used to answer design questions at each step of the design, which enable human designers to make more informed design choices.

Introduction

Designing and tuning a game is a complex creative process, which can be time-consuming to do well. While procedural content generation (PCG) is often lauded as a solution to generating more content, it is not necessarily a panacea for design, because working with PCG introduces a new sets of challenges that can be just as expensive to overcome as the original challenges (Rabii and Cook 2023).

Many AI tools have been created to help with the design process. Many of these assist with single levels in an already-defined game (Smith, Whitehead, and Mateas 2010; Liapis, Yannakakis, and Togelius 2013; Sturtevant and Ota 2018; Guzdial, Liao, and Riedl 2018; Sturtevant et al. 2020), or small changes to existing games (Guzdial and Riedl 2016). But, relatively little work has looked at tools for complete game design, and notable examples of complete game design are fully automated (Smith and Mateas 2010; Browne 2014; Cook, Colton, and Gow 2017a,b).

Within this context, this paper explores how Exhaustive Procedural Content Generation (EPCG) (Sturtevant and Ota 2018) can be used as a co-creative assistant for both game and puzzle design. The work is inspired by a reference puzzle shown in Figure 1(a). This broadly available[1] reference puzzle has an implicit goal of placing all pieces onto the board. But, the design is lacking: pieces are different colors, but colors serve no design purpose. There are many possible solutions, but nothing to distinguish them, and there is no curriculum to guide exploration. Finally, besides fitting pieces on the board, there are no additional constraints to play. Thus, our goal was to design a new version of the puzzle, beginning from these observations, and working towards a complete curriculum of puzzles with a rich set of constraints, using EPCG to assist each step of the design process. This paper documents our design process, similar to other efforts that document design (Togelius 2011; Khaled, Lessard, and Barr 2018). This processes shows how EPCG can be used as a co-creative partner, resulting in the product in Figure 1(b).

(a) (b)

Figure 1: (a) Reference puzzle and (b) near-final product

At this end of this work we have created 3D printed puzzles from our design, along with a curriculum of puzzles, which have been popular at lab meetings and other showcases. The broader contributions of this work include:

- A demonstration of the feasibility of using EPCG as a co-creative design partner, resulting in a complete puzzle which has been showcased at various local events.
- A concrete example of how computational systems can be used to support design, including both numerical analysis and runtimes.
- Reflections the experience of using EPCG as a co-creative AI partner. We observe that it is easy to pose questions to an EPCG system which are computationally infeasible. But, these questions are also too broad from a design perspective as well. Much of the design iteration came from honing the questions which are asked both

[1]We found a copy at the large North American warehouse Costco; it is also available from Amazon.

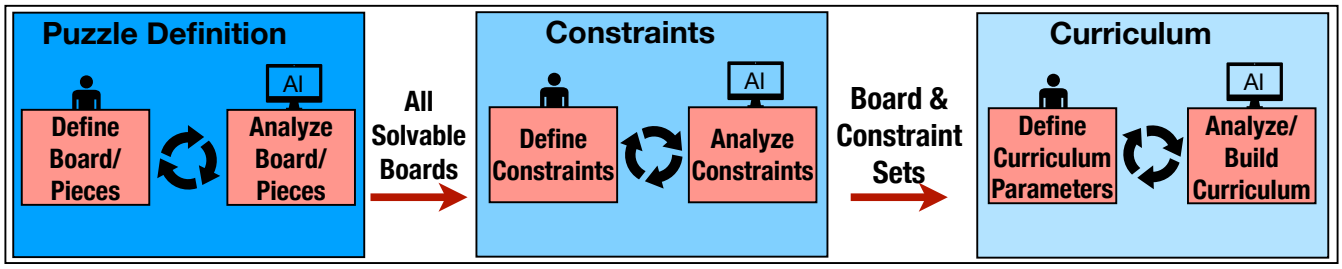


Figure 2: Overall Design Stages

be more precise, which also tends to make them computationally feasible. Thus, the human designers learned about the design space while interacting with EPCG, leading to an even more precise design.

The overall design process had three independent stages, shown in Figure 2. The first stage focused on defining the board and selecting pieces used in the design. The second stage takes the set of solved boards as input, and focused on defining constraints between pieces. The third stage takes all puzzles and constraints and builds a complete curriculum. After describing background work, we cover each of these design stages individually, and then conclude by reflecting on the design process and future possibilities. The final product is described further and available elsewhere (Mahmoud and Sturtevant 2024).

Background and Related Work

In this section we focus primarily on work related to creating complete games and the tools we will use for this purpose. We refer interested readers to an available survey on puzzle design (De Kegel and Haahr 2019).

Complete Game Design

There are many tools that have been built for assisting game design, or designing games from human input. Notable examples of tools for building complete games from scratch include the Ludi system (Browne 2014) and Angelina (Cook, Colton, and Gow 2017a,b), although there have been examples of other generators creating small games from human input (Treanor et al. 2012). Recent work has built on these ideas to build a deliberative game design tool that humans can use for novel game design (Cook 2022).

One notable example of building AI tools to assist in game design is work around the Refraction Game (Smith et al. 2012; Butler et al. 2013), where AI tools provided extensive support in the game design process. This work is closest in intent to our approach here.

Exhaustive Procedural Generation

EPCG (Sturtevant and Ota 2018) is a method of generating content that uses a *generator* to enumerate all possible content, and an *evaluator* to rank the content. Naive applications of EPCG simply enumerate all content, but more sophisticated algorithms can be used to generate and evaluate content more efficiently.

In this work EPCG is used primarily as an oracle to answer design questions, very similarly to a proposal by Paul Tozour in an AI Summit GDC lecture describing a 'Shigi' design tool (Tozour 2013).

For instance, given a set of pieces and a board, we can ask 'how many ways are there to compactly pack pieces on the board'. This query can be transformed into a generator that generates all possible piece combinations, and an evaluator that returns *true* for complete board positions. The answer to the design question is obtained by counting the number of positions that evaluate to *true*.

Due to the scope of this work, we will not describe the mathematics behind the generators used. Instead, we will focus on the queries, their results, and how they impact our design process.

There are other approaches that can be used similarly, including genetic algorithms (Golberg 1989), Answer-Set Programming (ASP) (Brewka, Eiter, and Truszczyński 2011) and Constraint Programming (CP) (Rossi, Van Beek, and Walsh 2006). Each of these have their own strengths and weaknesses; the goal here is to explore the potential of EPCG, not to compare EPCG against alternate tools.

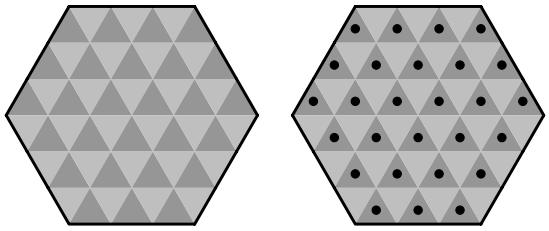
Puzzle Definition

The design space of tangram puzzles is exceedingly broad. We could use EPCG to directly generate all possible tangram puzzles and select the best, but this space is prohibitively large, it is not clear how to measure *best*, and at this stage this would remove almost all human agency from the puzzle design. Breaking down this process into smaller pieces not only increases the co-creative nature of the process, but also introduces design questions that are computationally feasible for EPCG to answer.

Our initial choice was to focus on tangram puzzles. A tangram is a puzzle that is comprised of multiple flat polygon pieces assembled to fill a board without overlap. We additionally chose to constrain our design to something that could be realized both physically and digitally, as this was expected to provide interesting design constraints later in the design process.

A *solved* board is an arrangement of pieces that cover the board without overlap. A puzzle *instance* is the initial state of a board given to a player where one or more pieces may already be given fixed locations on the board.

The first choice is to select the shape of the board and the pieces to be placed. Given that we were seeking to im-



(a) (b)

Figure 3: (a) Hexagon puzzle board. The board is made up of 54 triangles; pieces are placed aligned to triangles. (b) Constraint locations.

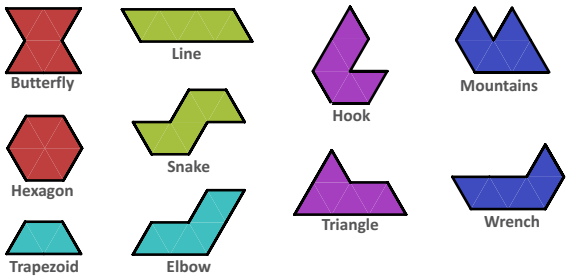


Figure 4: The selected set of pieces and the names assigned to them (excluding the "Trapezoid" piece). Notice how they are all composed of two trapezoids in some configuration.

prove the reference puzzle shown in Figure 1(a), we decided to keep the hexagon shape of the board. Our analysis of the board and the pieces used in the reference puzzle revealed that every piece is either a trapezoid (composed of three underlying triangles), or is the *compound* combination of two trapezoids. The board has 54 underlying triangles on which pieces are placed, shown in Figure 3(a).

Given this, our first design question was: how many unique pieces can be composed from two trapezoids? While we could build an EPCG query to answer this question, this can be enumerated by hand within a few minutes, so it was much faster to answer this question ourselves. All possible pieces are found in Figure 4, along with names that we will use to refer to these pieces. Our reference puzzle has four trapezoid pieces, and does not have the butterfly or snake pieces. Given that there are 54 triangles on the board, and 6 triangles per piece (excluding the trapezoid), it is possible that solved boards with just the nine compound pieces can be found.

So, the next task is to explore what pieces would be interesting to use for the puzzle. To answer this, we want to know the number of puzzle solutions for each possible subset of pieces in Figure 4, where two trapezoids are allowed in any subsets containing the trapezoid. Given there are 54 triangles, there are at most 9 piece types on the board. Thus, computing subsets is equivalent to removing one pieces from the set, and computing the number of solutions with the remaining pieces. This results in the following EPCG query, which, like all remaining queries in the paper, is implemented in C++ code:

Piece	Locations	Piece	Locations
Mountains	192	Wrench	78
Hook	192	Elbow	78
Triangle	156	Line	72
Trapezoid	156	Butterfly	42
Snake	84	Hexagon	19

Table 1: Number of locations for each piece type.

Generator: All possible placements of pieces from Figure 4.
Evaluator: Whether a given placement of pieces is a legal puzzle solution with no overlapping pieces.

Before looking at the results of this query, we consider possible methods of implementating the query. In particular, we can compute the number of locations that each piece can be placed, and then look at placements of all pieces combinatorially. Due to symmetry and the fact that pieces have two different sides, the number of locations for each piece varies from 19 for the hexagon to 192 for the hook and mountains, as shown in Table 1. The number of locations for each piece is acquired by considering not only the location of the piece on the board, but also its orientation and the side it is placed on. Multiplying the combinations together, each subset of pieces has between 10^{17} and 10^{18} total ways it can be placed on the board, which is computationally infeasible. But, almost all of these configurations are not solved boards. So, the computation can be optimized by placing pieces on the board incrementally, detecting illegal combinations as soon as they are encountered, similar to a DPLL search (Davis and Putnam 1960).

The optimized query took 9 minutes and 40 seconds on an Apple M1 MacBook Air 2020 with 8 GB of RAM, and required looking at just 392,326,063 board configurations. Using better rules to detect unsolvable boards earlier in the search, which we discuss later, reduced the running time to just one minute and 18 seconds and 2,595,601 total expansions. This query would not be feasible for humans, but the results are very useful to human designers.

The results of the query are in Table 2. We first learn that although it would be nice to work with all compound pieces, there are only nine unique solutions when trapezoids are not used, which isn't large enough to build an interesting curriculum. If we were to exclude any single piece, eliminating the hexagon would give maximum flexibility later in the design process, as there are 673 unique solutions that don't incorporate the hexagon (with there being one instance of each of the other pieces besides the two trapezoids).

While we had originally intended to select a subset of pieces to build the puzzle curriculum, we decided to keep all pieces in Figure 4, noting that for any puzzle instance a single piece would be forbidden. This brings some level of completeness into the piece selection, given that we have two trapezoids, as well as all pieces that can be built from two trapezoids. The resulting structure is expected to help players understand and solve puzzles more easily, something we will discuss later.

Excluded	Solutions	Excluded	Solutions
Hexagon	673	Hook	179
Butterfly	352	Triangle	176
Wrench	321	Elbow	129
Snake	307	Line	97
Mountains	265	Trapezoid	9

Table 2: Number of solutions found with each possible piece excluded from the subset. There are 2,508 total solutions among all subsets.

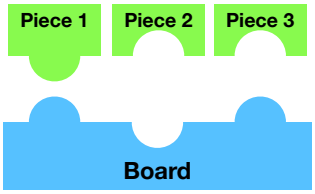


Figure 5: Every triangle on the board or on a piece has either a bump or a hole. Bumps cannot be placed opposite a bump (piece 1), but holes can be aligned with either a hole (piece 2) or a bump (piece 3).

This completes the first design stage shown in Figure 2. The next stage take as input the 2,508 solved boards.

Puzzle Constraints

Given that we have the complete set of solved boards, we can now consider constraints that can be added to make puzzle instances more interesting. In particular, constraints will be defined on properties of the board or pieces that prevent pieces from being placed in given locations or orientations. There are two possible constraints that we consider: constraints between pieces and the board, called *placement constraints*, and constraints between pieces, called both *adjacency constraints* and *color constraints*. Per Figure 2 we execute several iterations of defining and analyzing these constraints before settling on the final set of constraints.

Placement Constraints

At the most abstract level, especially if we had a digital-only puzzle, placement constraints could be designed to prevent any piece from being placed in any subset of the locations in Table 1. Thus, we could, for instance, arbitrarily restrict the triangle piece to 100 of the 156 possible locations. But, there are both too many possible combinations to formulate a feasible EPCG query, and arbitrary constraints are not going to be well-understood by players. So, after considering the most abstract version of the problem, we perform another design iteration to reduce the design choices in a way that will be well-understood by players.

Placement Constraints on the Board One of the primary considerations in designing placement constraints is the practicality of implementing them on a physical puzzle. This can be done, for instance, by altering piece shapes slightly so that they do not fit snugly into the board. After considering different possibilities we chose to incorporate physical bumps and holes into each the implicit triangles which compose both the board and the pieces. This is shown in Figure 5. If both the board and the piece have a bump in the same location, the piece will not be able to be placed there. However, if they both have holes, or one has a hole and one has a bump, then a piece would fit. It is also possible to have holes of different sizes and shapes to further constrain the board, but we chose not to use that in our design. However, to avoid over-constraining the problem, we decided to use the physical nature of the board to give more options. In particular, we can build a board that has no constraints on one side (all holes), but if the board is flipped over, there is a bump pattern on the opposite side that constrains solutions on that side of the board.

Now, the question is, what pattern of bumps should be placed on the pieces and the board? This question can be formulated as an EPCG query for both, but, at least for the board it is not clear that the query is meaningful. We have already decided that one side of the board will have no constraints. On the other side of the board there are $2^{54}-1 = 18$ trillion ways of placing one or more bumps on locations on the board. While it is feasible to enumerate all of these, it is not clear that there is a meaningful difference between many of these combinations. Thus, instead of running this query, we choose a pattern that will constrain the space in an interesting way for players.

We considered two possibilities. First, having bumps on all locations on one side of the board. This would essentially mean that on that side of the board any piece with a bump on it couldn't be placed with the bump facing down, or that it could only be placed on one side. Second, using a fixed pattern of bumps on every other location, as shown in Figure 3(b). With this pattern we can put bumps on all triangles on one side of a piece to prevent the piece from being placed on that side, or put bumps on a subset of triangles to simply limit the number of locations a piece can be placed. Because this option gives more flexibility, but is still easy for a player to understand, we chose this pattern for the second side of the board.

Placement Constraints on Pieces Now we can consider putting constraints on pieces, which will impact their placement on the second side of the board, but not the first. We define the parity of a dot pattern on a piece as odd or even, as seen for the trapezoid in Figure 6(a) and (b).[2] Looking at the problem combinatorially, a piece can have an odd bump pattern, an even bump pattern, no bumps, or all bumps. We exclude all bumps on both sides of the puzzle, as this would prevent the piece from being placed on the second side of the board, so there are $4 \times 4 - 1 = 15$ possibilities for each piece. Allowing each of the two trapezoid to have different constraints gives $15^{11} = 8.6$ trillion possibilities. This results in the following EPCG query:

[2]At the level of description provided in this paper it doesn't particularly matter which pattern is odd or even, just that we distinguish between the two.

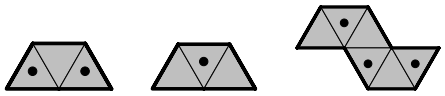


Figure 6: Odd and even bump patterns and the impact of piece symmetry.

Generator: All combinations of dots on all pieces.
Evaluator: How many solved boards are valid with this dot configuration.

From all combinations we select the pattern than maximizes the minimum number of solved boards which are valid with that set of constraints. This is so that we have as many puzzles as possible for building a curriculum.

However, when coding and testing this query, we made an unexpected discovery. The symmetry of the pieces was making many of the design choices being considered in the query irrelevant.

As previously mentioned, certain pieces exhibit forms of symmetry that become apparent when specific transformations are applied to them. The hexagon, butterfly, elbow, and trapezoid pieces appear identical when physically flipped over (and possibly re-oriented), which we call *flip* symmetry. The hexagon, butterfly, snake, and line pieces maintain their appearance when rotated 180 degrees, which we call *rotational* symmetry.

We can leverage these symmetries to shrink the combinatorial space. Consider the flip-symmetric elbow and trapezoid pieces. Suppose that the trapezoid in Figure 6(a) cannot fit onto the board because the bump pattern conflicts with the board, but the opposite side of the piece has opposite bump parity. Then, flipping the piece and rotating it gives the configuration in Figure 6(b), which will not be constrained.

So, pieces with flip symmetry (the trapezoid and the elbow) should not be given different bump parity on different sides of the piece. Thus, there are only three combinations for these pieces (even parity, odd parity, no bumps). Pieces with rotational symmetry (the snake and the line) should not have even or odd parity, because rotating the piece would similarly eliminate any constraints, as shown in Figure 6(c). If we rotate the snake piece 180° it will have the opposite parity. Finally, pieces with both types of symmetry (the hexagon and the butterfly) should not have any bump pattern, because these patterns can never restrict play.

There are 4 pieces with no symmetry which can have any bump pattern, 4 pieces with flip or rotational symmetry which can have three possible bump patterns, and 2 pieces that should have no bumps. Overall this reduces the size of the previous query from 8.6 trillion to 4.1 million possibilities. This not only is a 6-order of magnitude reduction in the query size, it also gives a much deeper understanding of the pieces and their relationships with each other. While we could, at this point, optimize bump patterns for pieces, we now begin to move towards adding adjacency constraints between pieces, which will change the EPCG queries we perform when finalizing constraints.

Adjacency Constraints

We now consider ways of placing constraints between different pieces on the board. Because we are using physical dots and holes to represent the constraints between pieces and the board, it makes sense to use a different method for representing constraints between pieces. In particular, we note that our reference puzzle has pieces with different color, but this is never exploited in the design, so we choose to use color as a way of representing constraints between pieces. In particular, we can either restrict or require that colors are adjacent on the board.

Selecting Colors for Adjacency Constraints To explore these constraints, we began to formulate EPCG queries around color. In particular, we originally considered choosing two or more pieces to have different color than other pieces, formulating queries such as the following:

Generator: All subsets of two pieces.
Evaluator: How many solved boards have adjacent pieces of the same color.

Here we were looking to understand how color constraints would impact the number of puzzles available for constructing a full curriculum. This work began before we completed the previous analysis of bump patterns, but it had not yet given any definitive answers for selecting piece color. But, once we understood the piece symmetries, we realized that we could color pieces according to their symmetries. In this way the colors would implicitly indicate something about the underlying properties of the pieces themselves. We decided to further reinforce this by giving pieces of the same color the same bump pattern. Because there are four pieces with no symmetry, but sets of two pieces with the other types of symmetry, we also decided to split the non-symmetric pieces into groups of two, where the sub-groups would have the same color and bump pattern.

Now, we were able to build a query that would help use choose both the bump patterns and color breakdown between pieces.

Generator: All valid bump patterns given symmetry constraints.
Evaluator: Number of valid puzzles.

For this query there are two groups of asymmetric pieces. The first group of pieces had four possible combinations of bump parity on each side: odd and none, even and none, none and odd, or none and even. The pieces in the second group had a different set of four combinations: odd and even, odd and odd, even and odd, or even and even. There are additionally 6 ways of dividing pieces between the two groups. Thus, the total number of combinations examined by this query is $1 \cdot 1 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 4 \cdot 4 \cdot 4 \cdot 4 \times 6 =$124,416

After executing the EPCG procedure to generate and evaluate all constraint patterns, we arrive at the results shown in Figure 7. The figure illustrates the valid solutions for each possible bump pattern. Over 40% of the patterns have no valid solutions, while 16 patterns have the maximum solution count of 164. Thus, on the second side of the board, only 164 of the 2508 total solutions will be possible. From these, with including color constraints, we have as few as 27

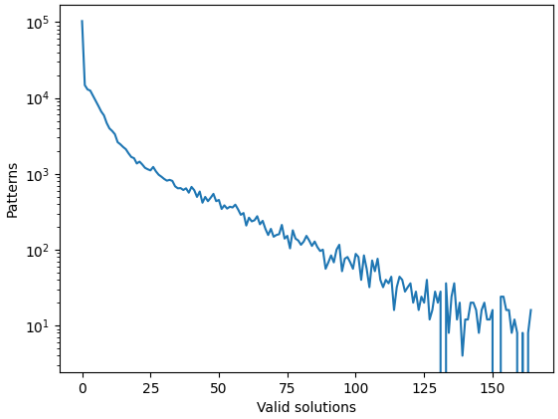


Figure 7: Distribution of patterns in relation to the number of valid solutions they have. The x-axis represents the number of solutions, while the y-axis denotes the number of patterns leading to each respective number of solutions.

puzzles available for a given chapter.

We then move to defining constraints between pieces and their colors. We will run a final EPCG query that considers both bump patterns and color constraints together to finalize the design.

Defining Color Adjacency Constraints Now that we have defined piece colors and bump patterns, and found all boards that are valid for these constraints, we can define constraints between pieces.

Our intent is to build a curriculum, with each chapter using different types of constraints. After considering how pieces of different colors can constrain solutions, we chose to build a total of 12 chapters into the curriculum. The first six consist of puzzles to be solved on the unconstrained side of the board (the side with no placement constraints, i.e., no bumps), while the remaining six are intended for the side of the board with bumps. The six chapters in each half are defined by the type of color constraint applied to them, as outlined in the following list:

1. No color constraints.
2. Pieces of color X must share an edge (note that in this case, we refer to one color, for example, red pieces must share edges with each other. The color is defined differently in each puzzle).
3. Pieces of color X must not share an edge.
4. Pieces of color X must share a corner, but not an edge.
5. Pieces of color X must not share a corner or an edge.
6. Any of the four constraint types above, but between pieces of different colors X and Y.

We can now formulate our final EPCG query for this design stage. This generator once again generates all possible bump patterns and colors, but this time it considers the puzzles that will be passed to the next stage, maximizing the minimum number of puzzles available for use in each chapter.

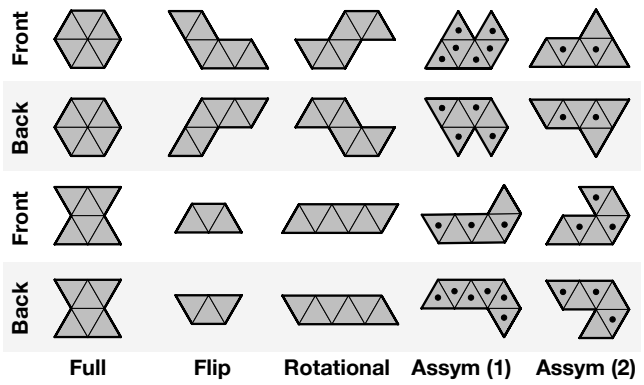


Figure 8: Final piece patterns. Holes are not drawn; bumps are black circles. Pieces are grouped by their symmetry patterns.

Generator: All valid bump patterns given symmetry constraints.
Evaluator: Minimum number of color constraints that can be defined for all valid boards in a chapter.

Running the complete analysis took approximately 50 minutes. Figure 8 shows the piece patterns that were selected for each side of the pieces. The colors in Figure 4 are the colors for each of the pieces.

The final output of the last EPCG run gives, for each chapter that we intend to build in the curriculum, the set of all puzzles that meet the board constraints and all possible color constraints that can be defined between pieces. These are taken as input into the final stage and used to build a curriculum.

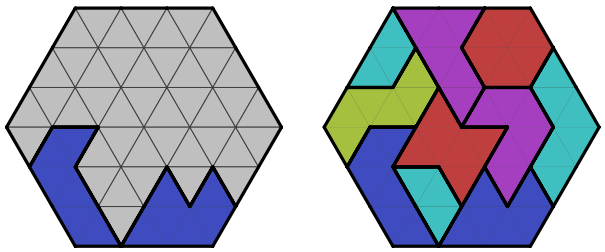
Puzzle Curriculum

To complete the puzzle design, we build a curriculum that presents puzzles which explore all of different constraints. This process has three steps. First, we construct puzzle instances, where a portion of the solved board is presented which guarantees a unique solution. Then, we use recent work on puzzle entropy (Chen, White, and Sturtevant 2023) to sort puzzles by difficulty. From these we select puzzles for the final curriculum.

Building Puzzles Instances

Our first task is to construct puzzle instances from solved boards. An example of a puzzle instance and the corresponding solved board is found in Figure 9.

The input to this step is the set of all solved boards with the corresponding constraints that are met for the board that came from the puzzle constraint analysis. We use an EPCG query to find unique puzzle instances for each solved board in the set. The input to the query is a single solved board as well as the set of all solved boards. The output is a puzzle instance that has a single unique solution.



(a) (b)

Figure 9: (a) A puzzle instance and (b) the unique solved board. (The line piece is disallowed in this puzzle)

Generator: From $k \in 1 \ldots 9$ all puzzle instances that can be constructed from a solved board with exactly k pieces in the instance
Evaluator: Whether the puzzle instance appears elsewhere as a subset of a solved board.

Note that the test of whether a puzzle instance is a subset of another solved board explicitly checks for symmetric solutions, but allows a duplicated subset when the provided constraints prevent all alternate solved boards from being legal solutions. Although there can be many different puzzle instances that can be used for a single solved board, we selected the first such puzzle instance found and then terminating the query, meaning that the query was run semi-exhaustively (Sturtevant and Ota 2018).

The runtime of this query depends on the number of solved boards that must be compared against. When executed for the unconstrained case, it takes approximately 0.8 seconds to find each puzzle instance.

Sorting Puzzles with Entropy

To estimate the difficulty of a puzzle, we use the MUSE algorithm proposed by (Chen, White, and Sturtevant 2023), although the similar TSI algorithm could have been a strong alternative (Shen and Sturtevant 2024), were it available earlier. This algorithm takes a puzzle and a set of inference rules, and returns the puzzle entropy by measuring the number of choices available on the solution path. Thus, to use this approach, we must design inference rules that encode the difficulty of a puzzle by reducing the number of choices available at some states. A full description of these rules with additional details are found in a MSc thesis (Mahmoud 2024); we describe them briefly here.

First, we check to see if there is a piece that can only be placed in one location. If there is, we infer that the piece must be immediately placed in that location. Our second rule is similar; if there is a location on the board that only one piece can cover properly, then we infer that piece must be placed there. Next, we note that pieces are made of either 3 or 6 triangles. If there is a region on the board with size that modulo 3 (or 6 if trapezoids are not available) is not 0, the puzzle will be unsolvable. Finally, even if there is a region size 6, if that region cannot be constructed from two trapezoids, then it cannot lead to a solution.

Table 3 presents statistics for the entropy of the puzzle instances in each curriculum category. Note that we set a hard cap of 3000 puzzles for each category since this number is sufficient when choosing our final set puzzles. This is especially important given that calculating entropy for entire sets can be computationally expensive and time-consuming.

Building a Curriculum

The final curriculum was built by sorting the puzzles in each chapter by their overall entropy and then selecting 12 puzzles evenly distributed from across the ranges of entropy. However, for the first chapter we used a slightly different approach to provide more guidance on the puzzles. In the first chapter, we went back to the original set of 2,508 solved boards and found initial states with more pieces on them. Then, for each of our inference rules, we measured the difference in entropy for the puzzles with no entropy rules, and then with a given inference rules. In the first chapter, we selected puzzles that maximized this difference. The goal here is to find puzzle instances are easy when you know a given rule, but hard otherwise, so they are more likely to teach these rules to players.

In the end this provides a comprehensive curriculum consisting of twelve chapters, each comprising twelve puzzles, yielding a total of 144 puzzles throughout the curriculum. The next step for this process is iterative playtesting, something that was not able to be performed in the scope of this work. However, in the testing we did perform, we found places where further iteration would be possible. For instance, there is a unique inference that can be made to solve the hardest puzzle in the curriculum more easily. The curriculum ordering could likely be refined by finding such rules and incorporating them into the design, something that is ongoing.

Reflections EPCG for Co-Creative Design

Having completed a full puzzle design using EPCG, we make the following observations about the process.

- If an EPCG query is too large to run quickly, the answer may not be interesting - the query is likely too broad or will return too much information to be usable.
- EPCG is very useful for helping making specific design choices (such as bump pattern on pieces), while human designers are good for making broad design decisions, such as the bump patterns on the board and the shape of the board.
- Framing design questions as EPCG queries was useful for making design progress. This process taught us more about the underlying nature of the puzzle, and pointed out where we didn't have a complete grasp of the design choices available. The discipline of attempting to formalize queries led to deliberate, well-informed decisions during design.
- While it did not seem to impact the design of this puzzle, one needs to be careful not to avoid exploring interesting spaces just because they are infeasible for EPCG queries.

Ch	Constraint type(s)	Puzzles	Min	Max	Avg	Med	Std
1	No constraints	2508	0.00	16.61	6.46	6.17	2.43
2	Edges must touch	3000	0.00	17.69	6.75	6.58	2.32
3	Edges must not touch	3000	0.00	16.27	6.84	6.58	2.38
4	Corners must touch	3000	0.00	17.69	6.87	6.58	2.34
5	Corners must not touch	444	1.00	13.85	7.93	7.81	2.16
6	Multi-color constraints	3000	1.00	16.15	7.52	7.39	2.48
7	Bumps	101	3.00	18.25	9.71	9.39	3.33
8	Bumps + Edges must touch	213	3.00	19.20	9.92	9.58	3.14
9	Bumps + Edges must not touch	136	3.00	20.05	10.90	11.28	3.88
10	Bumps + Corners must touch	174	3.00	19.89	9.96	9.58	3.12
11	Bumps + Corners must not touch	27	6.32	18.34	12.04	11.70	2.88
12	Bumps + Multi-color constraints	292	3.00	20.05	11.55	11.57	3.70

Table 3: Entropy statistics for the chapters.

Query	Runtime
Generating all solutions	1min 18sec
Generating placement constraints	35min
Generating adjacency constraints	50min

Table 4: Query runtimes.

After completing this design process and documenting it, we note that an EPCG query can have three results. First, it may produce useful information that finalizes design decisions and moves forward with the design. Second, a query may be computationally infeasible, forcing the design to be refined to make it feasible. Finally, a query can return a result that does not provide useful guidance, requiring further iteration to get to a query that could move the design forward. This last option occurred with our initial forays into defining piece color.

Conclusions and Future Work

This paper has described the process of building a puzzle through iterative EPCG queries interleaved with a human design process. The result of this is a 3D printed puzzle board and curriculum puzzles which have been shared and enjoyed at local events. We are continuing to polish the curriculum and have made the design publicly available (Mahmoud and Sturtevant 2024).

One broad question coming from this work has to do with the fact that we were able to implement our design queries in C++. Not all designers will have this capability. This leaves an open question of how generic tools could be built that will be helpful for novel game designs and less technical designers, something for researchers to consider in the future.

Acknowledgements

This work was supported by the National Science and Engineering Research Council of Canada Discovery Grant Program and the Canada CIFAR AI Chairs Program.

References

Brewka, G.; Eiter, T.; and Truszczyński, M. 2011. Answer set programming at a glance. *Communications of the ACM*, 54(12): 92–103.

Browne, C. 2014. Evolutionary game design: Automated game design comes of age. *ACM SIGEVOlution*, 6(2): 3–16.

Butler, E.; Smith, A. M.; Liu, Y.-E.; and Popovic, Z. 2013. A mixed-initiative tool for designing level progressions in games. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology*, UIST '13, 377–386. New York, NY, USA: Association for Computing Machinery. ISBN 9781450322683.

Chen, E. Y. C.; White, A.; and Sturtevant, N. R. 2023. Entropy as a measure of puzzle difficulty. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 19, 34–42.

Cook, M. 2022. Puck: a slow and personal automated game designer. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 18, 232–239.

Cook, M.; Colton, S.; and Gow, J. 2017a. The ANGELINA Videogame Design System—Part I. *IEEE Transactions on Computational Intelligence and AI in Games*, 9(2): 192–203.

Cook, M.; Colton, S.; and Gow, J. 2017b. The ANGELINA Videogame Design System—Part II. *IEEE Transactions on Computational Intelligence and AI in Games*, 9(3): 254–266.

Davis, M.; and Putnam, H. 1960. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7(3): 201–215.

De Kegel, B.; and Haahr, M. 2019. Procedural puzzle generation: A survey. *IEEE Transactions on Games*, 12(1): 21–40.

Golberg, D. E. 1989. Genetic algorithms in search, optimization, and machine learning. *Addion wesley*, 1989(102): 36.

Guzdial, M.; Liao, N.; and Riedl, M. 2018. Co-Creative Level Design via Machine Learning. arXiv:1809.09420.

Guzdial, M.; and Riedl, M. 2016. Learning to Blend Computer Game Levels. arXiv:1603.02738.

Khaled, R.; Lessard, J.; and Barr, P. 2018. Documenting trajectories in design space: a methodology for applied game design research. In *Proceedings of the 13th International Conference on the Foundations of Digital Games*, 1–10.

Liapis, A.; Yannakakis, G. N.; and Togelius, J. 2013. Sentient sketchbook: computer-assisted game level authoring. In *Foundations of Digital Games*. ACM.

Mahmoud, Y. 2024. *End-to-end Game Design Using EPCG*. Master's thesis, University of Alberta.

Mahmoud, Y.; and Sturtevant, N. R. 2024. Hexagram Tangram Puzzle Artifact. In *Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*.

Rabii, Y.; and Cook, M. 2023. Why Oatmeal is Cheap: Kolmogorov Complexity and Procedural Generation. In *Proceedings of the 18th International Conference on the Foundations of Digital Games*, 1–7.

Rossi, F.; Van Beek, P.; and Walsh, T. 2006. *Handbook of constraint programming*. Elsevier.

Shen, J.; and Sturtevant, N. R. 2024. Generalized Entropy and Solution Information for Measuring Puzzle Difficulty. In *Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*.

Smith, A. M.; Andersen, E.; Mateas, M.; and Popović, Z. 2012. A case study of expressively constrainable level design automation tools for a puzzle game. In *Proceedings of the International Conference on the Foundations of Digital Games*, FDG '12, 156–163. New York, NY, USA: Association for Computing Machinery. ISBN 9781450313339.

Smith, A. M.; and Mateas, M. 2010. Variations Forever: Flexibly generating rulesets from a sculptable design space of mini-games. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*, 273–280.

Smith, G.; Whitehead, J.; and Mateas, M. 2010. Tanagra: An Intelligent Level Design Assistant for 2D Platformers. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 6, 223–224.

Sturtevant, N.; Decroocq, N.; Tripodi, A.; Yang, C.; and Guzdial, M. 2020. A Demonstration of Anhinga: A Mixed-Initiative EPCG Tool for Snakebird. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 16(1): 328–330.

Sturtevant, N.; and Ota, M. 2018. Exhaustive and semi-exhaustive procedural content generation. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 14, 109–115.

Togelius, J. 2011. A procedural critique of deontological reasoning. In *Proceedings of DiGRA 2011 Conference: Think Design Play*.

Tozour, P. 2013. From the Behavior Up: When the AI Is the Design. Presentation at the AI Summit GDC.

Treanor, M.; Blackford, B.; Mateas, M.; and Bogost, I. 2012. Game-o-matic: Generating videogames that represent ideas. In *Proceedings of the The third workshop on Procedural Content Generation in Games*, 1–8.