End-to-end Game Design Using EPCG

by

Yazeed Mahmoud

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

Abstract

Procedural content generation (PCG) algorithms have been utilized for automating the creation of game content such as levels, assets, and narratives. One specific type, Exhaustive PCG (EPCG), systematically generates all possible variations of content before selecting the best, embodying a form of exploratory creativity. While EPCG has been applied to modifying or creating entire levels in games, there has been no research into employing EPCG for generating games end-to-end. This work explores the application of EPCG to game design, specifically focusing on the tangram puzzle genre. The game design process involves collaboration between human designers and computational algorithms. Our aim is to create a puzzle curriculum with multiple chapters, each introducing a new combination of constraints. Through this work, we demonstrate the potential of EPCG in driving end-to-end game design. Additionally, we introduce the concept of differential entropy, which assists in teaching puzzle inference rules, and utilize it to develop a puzzle curriculum that encourages players to understand the puzzle more deeply.

Through understanding of the proper way, our faith, our grace, is increased.

This is for thee.
Mine abundance, my drop of dew.
Quench thy thirst, throughout thy frame.
Blossom and burgeon, time and again.
Grow larger, stronger.
Until the day cometh.
When thou canst share in my dream.

Just as still waters turn foul, stagnation leads to decay. Warriors must remain ever drifting.

Acknowledgements

I would like to express my gratitude, first and foremost, to my family, who have stood by me unwaveringly since the inception of my academic journey. I am profoundly thankful to my late parents, whose influence has played a pivotal role in shaping my identity and character. Additionally, I am immensely grateful to my siblings—Mohammed, Yaseen, Yazan, Razan, Rawan, and Remaz—for their unwavering support and encouragement, which have been integral to my accomplishments.

I extend my heartfelt appreciation to Nathan. His expertise, encouragement, and constructive feedback have been invaluable in shaping the direction and outcomes of this research endeavor. His mentorship has not only enhanced my understanding of the subject matter but has also fostered my growth as a researcher. I am indebted to him for his unwavering dedication and commitment to my academic and professional development. I extend my gratitude to Matthew for sparking my interest in PCG through his exceptional course. Additionally, I am thankful to Rania for being the first professor to recognize my potential and for providing unwavering support throughout my undergraduate studies.

Finally, I would like to thank the friends I have made in Edmonton, who have transformed this city into a home for me. I am particularly grateful to Wisam, Adeel, Paul, Isaac, Diego, Anna, Momen, Kevin, Cathrin, Chen, Fiona, Akalanka, and Eve for their unwavering support and friendship. Additionally, I am thankful to the members of the Moving AI lab for their constructive feedback and encouragement throughout the project. And I am greatful to the Computing Science department who made all of this possible.

This work is a re-implementation of the original exploration performed by

Augustine Blanchonnet, who built a prototype curriculum for the same puzzle using Unity and C#. The codebase was built on the initial software provided by Professor Nathan Sturtevant and written in C++. Nathan also helped with generating the 3D models and printing them as puzzle prototypes. ChatGPT was utilized to assist with grammar and spell-checking.

Contents

Abstract ii

Acknowledgements iv

List of Tables ix

List of Figures x

1 Introduction 1

2 Background 8

2.1 PCG 8

2.2 EPCG 9

2.3 Examples of EPCG 11

2.3.1 Fling! 11

2.3.2 The Witness 12

2.3.3 Anhinga 14

2.4 Puzzle Entropy 15

3 Building the Puzzle 20

3.1 Defining Puzzle Framework 20

3.2 Generating All Solutions 24

4 Constraint Analysis **32**

4.1 Constraint Types 32

4.2 Placement Constraints 33

4.2.1 First Iteration 35

4.2.2 Second Iteration 36

4.2.3 Third Iteration 38

4.2.4 Fourth Iteration 40

4.2.5 Fifth Iteration 43

4.2.6 Final Evaluation 43

4.3 Adjacency Constraints 45

5 Building the Curriculum **55**

5.1 Finding Initial States 55

5.2 Entropy Analysis 59

5.3 Inference Rules 60

5.3.1 Location can only fit a certain piece rule 60

5.3.2 Piece can only go in one place rule 61

5.3.3 Size of empty space rule 61

5.3.4 Piece that fits the space not available rule 62

5.3.5 Pieces are composed of trapezoids rule 64

5.4 Differential Entropy 65

5.5 Design of the Curriculum 67

6 Curriculum Evaluation **71**

6.1 Curriculum Analysis 71

6.2 Playtesting Experiment 76

7 Conclusions and Future Work **80**

Bibliography **84**

Appendix **90**

7.1 Curriculum 90

7.1.1 Chapter 1 91

7.1.2 Chapter 2 94

7.1.3 Chapter 3 97

7.1.4 Chapter 4 100

7.1.5 Chapter 5 103

7.1.6 Chapter 6 106

7.1.7 Chapter 7 109

7.1.8 Chapter 8 112

7.1.9 Chapter 9 115

7.1.10 Chapter 10 118

7.1.11 Chapter 11 121

7.1.12 Chapter 12 124

List of Tables

3.1 Number of locations for each piece type. 27

3.2 Number of solutions found with each forbidden piece type. 31

5.1 Entropy statistics for the chapters. 66

List of Figures

1.1 A tangram puzzle 3
1.2 Puzzle curriculum 4
1.3 Process diagram 4

2.1 A Fling! board 11
2.2 A Witness puzzle panel 13
2.3 A Snakebird level 14
2.4 Anhinga editor 15

3.1 Steps covered in chapter 3 20
3.2 The Ostomachion 21
3.3 Puzzle board 22
3.4 The trapezoid piece 22
3.5 Set of pieces 23
3.6 Scanning the piece board locations 25
3.7 Piece rotation 26
3.8 Rotationally-symmetric pieces 26
3.9 Examples of flipping pieces 26
3.10 Flip-symmetric pieces 27
3.11 Duplicate solution states 29

3.12 Depth-first search tree 30

4.1 Constraint analysis section of the architecture 33
4.2 Two sides of a board 34
4.3 Bumps and holes 37
4.4 Board constraint patterns 39
4.5 Flip-symmetric pruning 41
4.6 Rotation-symmetric pruning 42
4.7 Pattern distribution 44
4.8 Types of adjacency 46
4.9 Selected placement pattern 53
4.10 Selected adjacency pattern 54

5.1 Puzzle's initial state and solution 56
5.2 Location can only fit a pertain piece rule 61
5.3 Piece can only go in one place rule 62
5.4 Size of empty space rule 63
5.5 Piece that fits the space not available rule 64
5.6 Pieces are composed of trapezoids rule 65
5.7 Curriculum building step 67

6.1 Curriculum evaluation step 71
6.2 Instruction pages 73
6.3 First page in introductory chapter 74
6.4 Solving the first puzzle 75
6.5 Final puzzle in the curriculum 78

List of Algorithms

1 EPCG(Generator, Evaluator) 10
2 MUSE(S_{start}, σ) 18
3 MinPathEntropy(s, σ) 18
4 FindAllSolutions() 29
5 PlacementConstraintSpaceSearch(solutions) 35
6 AdjacencyConstraintSpaceSearch(solutions) 50
7 FindInitialState(solution, otherSolutions) 57

Chapter 1

Introduction

The field of procedural content generation (PCG) has emerged as a pivotal area of research and development across various domains, ranging from video game design to creative arts and beyond (Togelius et al., 2011). Procedural content generation refers to the process of algorithmically creating content, such as levels, landscapes, characters, and narratives, through automated means rather than manual authoring (Smith, 2010). This approach holds promise for improving content creation pipelines by enabling the generation of vast, diverse, and in some cases even dynamically adaptive content spaces (Yannakakis & Togelius, 2015).

PCG is notably prevalent in the video game industry, enhancing gameplay experiences and facilitating the creation of expansive game worlds. Numerous games across different genres have leveraged PCG to generate dynamic and diverse content. For instance, *Minecraft* utilizes PCG to procedurally generate its "blocky" terrain (Smith et al., 2019), allowing players to explore and interact with procedurally generated landscapes and structures. *Spelunky* employs PCG to generate its levels, ensuring that each playthrough offers unique challenges and

surprises (Mossmouth, 2008). In *Diablo* and its sequels, PCG is used to generate randomized dungeons, loot, and enemy encounters, contributing to the game's replayability (Blizzard, 1996). Additionally, *Rogue Legacy* features PCG-generated levels and character traits, providing a new experience with each playthrough (Toy et al., 1980). These examples showcase the versatility of PCG in creating dynamic and engaging gameplay experiences across various game genres.

One form of PCG is exhaustive procedural content generation (EPCG), a methodical approach in which all possible variations of content within specified constraints are systematically generated (Sturtevant & Ota, 2018). Unlike stochastic or random generation methods, which rely on probabilities and randomization, EPCG aims to explore every possible combination or permutation of content elements meeting defined criteria. This approach is particularly useful in scenarios where the generated content must strictly adhere to predetermined rules or constraints. EPCG systematically generates all possible variations which ensures thorough exploration of the content space. It offers a comprehensive view of potential content options available. This makes it an example of exploratory creativity (Antonios et al., 2016). However, it is important to note that EPCG can be computationally intensive and may not be feasible for generating large or complex content spaces due to the sheer number of possible variations.

Some examples of EPCG include the construction of new levels for the game *The Witness* (Sturtevant, 2019). Another instance is found in the game *Anhinga* (a clone of the game *Snakebird*), where instead of exhaustively generating levels, which is impractical due to the large number of permutations, researchers chose to exhaustively generate neighbors of a state (a state refers to a configuration of elements or pieces in a level). This approach allowed them to develop a tool that assists designers in building levels incrementally (Sturtevant et al., 2020a). However, no known research has attempted to use EPCG to generate games in

Figure 1.1: A tangram puzzle. Note that in this puzzle, pieces have different colors, although color does not affect the puzzle shown here. This appears to be unused design space, which is why we include color as one of the constraint types we introduce. Image source: https://www.amazon.in/Yigo-Hexagon-Tangram-Handmade-Educational/dp/B07JBVFRFK

an end-to-end manner—that is, starting from a basic idea of a game and arriving at a complete product.

In this work, **our thesis** is that we can employ Exhaustive Procedural Content Generation (EPCG) for end-to-end game design. The puzzle game we are basing our work on is an instance of tangram puzzles, in which the player is given a set of pieces and a board that may contain a few pieces initially. The player's goal is to place all the remaining pieces on the board. An example of a tangram puzzle is shown in Figure 1.1. We attempt to explore an initially shallow puzzle that has no set of levels or constraints and produce a curriculum that would allow players to think about the puzzle computationally and aid them in understanding its underlying mathematics (Aho, 2011; Blow & ten Bosch, 2011).

As shown in Figure 1.2, the curriculum we plan to build consists of a number of chapters (N chapters in the figure). We generate all constraint combinations and assign one to each chapter. Each constraint combination results in a list of

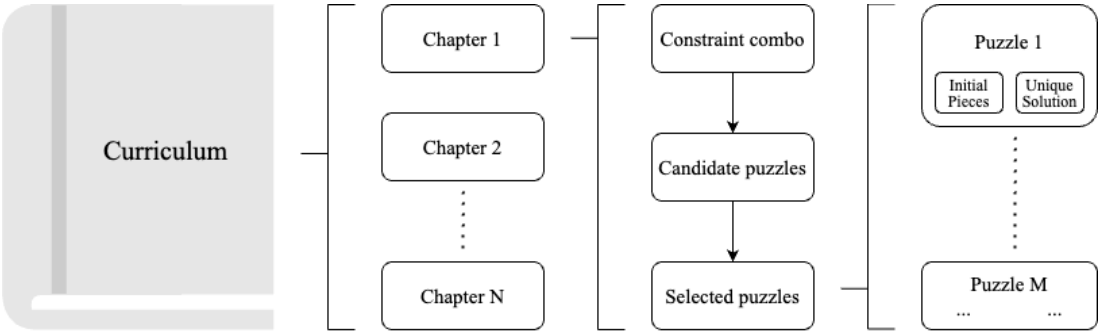


Figure 1.2: Puzzle curriculum.

candidate puzzles that satisfy the constraints. Then from the list of candidate puzzles we choose a list of M selected puzzles per chapter, each puzzle consisting of an initial state and a unique solution.

Figure 1.3 summarizes the major steps we follow in this work. In this process, each step is either taken by the human designer or by the computer, denoted by small icons at the top of each step's block.

The process starts by defining the puzzle, which includes describing the shape of the board and the shapes of the pieces in the set. This step is performed by the designer. The next step involves formulating this puzzle description as a search problem and running search in order to find all possible solutions for the puzzle and compile them into a list. This step is handled by the computer. This list of solutions is then passed to the subsequent step, where various constraints

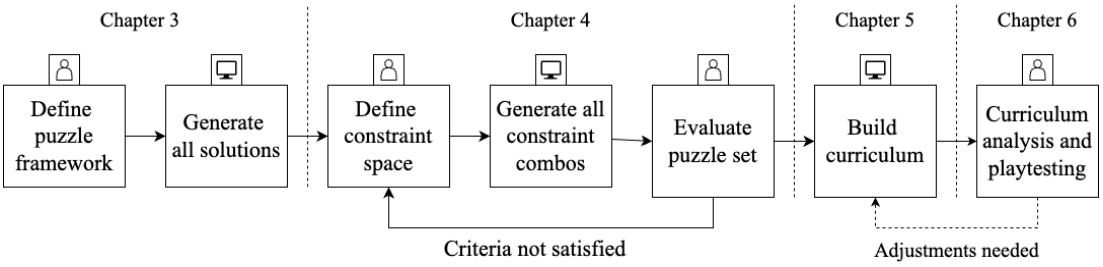


Figure 1.3: Process diagram.

applicable to the puzzle are defined by the designer, as we will discuss momentarily. After defining such constraints, the computer then generates all possible combinations of the constraints and filters the puzzles with solutions that satisfy each constraint.

The constraints we consider fall into two categories: constraints that restrict the allowable locations for a piece on the board, and constraints that specify interactions between two or more pieces. In the latter, we implement these constraints by assigning colors to pieces and then mandating that pieces of certain colors must either touch or remain separate (either by edges or corners). We discuss constraint types further in Chapter 4.

After generating all constraint combinations the next step involves the designer's selection of the best combination of constraints. Our aim is to build a curriculum comprising chapters of puzzles with similar constraints. For each chapter, we want a sizable pool of candidate puzzles to cover a wide range of difficulty levels. Therefore, our criterion for choosing the best combination of constraints specifies that each constraint must have a sufficiently large number of solutions satisfying it. We repeat the process of defining the constraint space and generating constraint combinations until a satisfactory constraint combination is found.

After finalizing a satisfactory constraint combination, the next phase involves constructing a curriculum based on the generated puzzles. This curriculum is structured into chapters, each aimed at introducing either a new constraint or a unique combination of constraints. The first chapter serves as an introduction to the puzzle, comprising a series of easier levels along with ones specifically crafted to teach particular skills to the player. We measure the difficulty of a puzzle using entropy analysis, a method that also allows for the integration of expert

knowledge to refine difficulty estimations (Chen et al., 2023). This approach aids in identifying puzzles that may be perceived as easy by experienced players but pose a challenge for beginners. Such puzzles are employed in the first chapter to help with the learning of essential skills by players. Subsequent chapters of the curriculum feature puzzles arranged in ascending order of difficulty, thereby giving a sense of progression.

The phase that follows building the curriculum is playtesting the puzzle using the curriculum. This entails inviting individuals and guiding them through the curriculum in specific ways to evaluate its effectiveness, collecting player feedback, and attempting to model the players based on their learning. Based on the results of the playtesting phase, further adjustments to the curriculum may be deemed preferable, necessitating perhaps other playtesting and adjustment cycles.

In this work, we evaluate the thesis that it is possible to employ Exhaustive Procedural Content Generation (EPCG) for driving game design from start to finish. This involved creating a complete puzzle game integrated with different constraint types into a final curriculum. Despite the automated nature of the process, developer input was still necessary to ensure the puzzle met specific aesthetic standards and remained intuitive for human players, highlighting the ongoing role of human creativity in game design. Additionally, we introduce the concept of differential entropy and implement an entropy analyzer for the puzzle alongside a set of inference rules. Furthermore, we employ the entropy analyzer we built to help devise a structured curriculum for the game that provides a sense of progression to players. This curriculum, particularly in its introductory chapter, leverages differential entropy to teach players various skills crucial for navigating the puzzle game effectively.

The rest of the thesis is organized as follows. In Chapter 2, we provide the

essential context required to understand EPCG and justify its use. Additionally, we introduce the concept of entropy and its utilization for estimating difficulty and discuss how it helps in building curricula. In Chapter 3, we outline the process of defining the puzzle framework and then generating all the solutions using EPCG. In Chapter 4, we discuss the process of exploring the constraint space for the puzzle, explaining the loop of defining a constraint space, generating all constraint combinations, and then evaluating the resulting puzzle set and looping back to redefine the constraint space if the selected criteria are not satisfied. In Chapter 5, we discuss the process of building the curriculum, starting with the initial state finder function which converts solutions and constraints into puzzles, then presenting the process of entropy analysis and the list of inference rules we developed. After that, we discuss the notion of differential entropy and conclude the chapter with a discussion of the process of using all the previous tools to construct the final curriculum, also discussing the structure of the final curriculum. In Chapter 6, we evaluate the curriculum we constructed, highlighting select puzzles from the introductory chapter. Additionally, we discuss the playtesting experiment conducted with participants from our laboratory. Finally, in Chapter 7, we summarize the contributions of this work and possible future work building on top of it.

Chapter 2

Background

In this chapter, we aim to establish the foundational knowledge necessary to understand and support the application of EPCG. We first introduce the concept of PCG and demonstrate which family of PCG approaches EPCG belongs to, before then delving into a more detailed explanation of EPCG specifically. After that, we introduce the concept of entropy and highlight its role in estimating difficulty, which is the property we use for evaluating the generated solutions.

2.1 PCG

Procedural Content Generation (PCG) is a technique used in computer science and digital media to generate content algorithmically rather than through manual authoring. PCG is applied across domains which include video game development, virtual environments, digital art, and simulation training (Yannakakis & Togelius, 2018). In games, PCG is employed to generate game elements, such as levels, music, sound effects, characters, quests, and so on. One of the primary advantages of PCG lies in its capacity to generate content that is unique and

dynamic, providing players or users with novel experiences with each interaction (Hendrikx et al., 2013).

PCG can be categorized into four paradigms: constructive PCG, constraint-based PCG, search-based PCG, and machine learning PCG (Summerville et al., 2017). Constructive PCG relies on manually crafted rules and functions to construct new content. Constraint-based PCG establishes criteria for defining a "valid" piece of content using constraints and applies these constraints to generate new content. Machine learning PCG employs machine learning techniques to create new content. Finally, search-based PCG defines the content space and utilizes optimization procedures to identify high-quality content within that space. Exhaustive PCG (EPCG) falls under the search-based family of PCG approaches.

2.2 EPCG

Exhaustive Procedural Content Generation (EPCG) describes approaches for generating procedural content where all possible content is methodically generated and evaluated (Sturtevant & Ota, 2018). The content generated by EPCG avoids repetition, which contrasts with randomized algorithms. In randomized algorithms, generating a set of N items randomly may result in duplicates and some items being missed entirely. Exhaustively generating the content saves time and avoids this inefficiency if the goal is to choose the best possible content.

As shown in Algorithm 1, an EPCG procedure takes in two inputs: a generator and an evaluator (*Generator* and *Evaluator* in the example algorithm, respectively). The generator takes in a problem description and produces states according to a predetermined generation algorithm. Meanwhile, the evaluator takes a state and provides a numerical score or "value" for that state, aiding

in the determination of which states deserve attention. The output of an EPCG procedure is the highest-value content. Alternatively, it could be the lowest-value or anything in between, depending on the designer's criteria.

Algorithm 1 EPCG(Generator, Evaluator)

```
1: best ← Generator.Unrank(0)
2: for i = 1 to Generator.MaxRank() − 1 do
3:     state ← Generator.Unrank(i)
4:     if Evaluator(state) > Evaluator(best) then
5:         best ← state
6: return best
```

For the generator in EPCG to produce all potential content, we must provide it with the size of the complete state space. This often entails a full combinatorial analysis that determines all the ways in which content can be arranged. The use of ranking and unranking functions is typically necessary in this case. A ranking function (Myrvold & Ruskey, 2001) takes in a state and produces an integer value (a hash), an unranking function on the other hand converts this integer value back to a state. The rank can be seen as an index accessing an implicit database of content produced by a given generator. This implicit content database is an array of all possible content states that the generator iterates through using the rank.

In EPCG, a content generator (*Generator* in the example algorithm) is a set of functions that includes two functions that we call $MaxRank()$ and $Unrank(rank)$. The $MaxRank()$ function returns the total number of states that can be generated. The $Unrank(rank)$ function takes in a rank (in other words, an index) and converts it into a state. This rank ranges between *zero* and $MaxRank() - 1$. An additional optional function is the $Rank(state)$ function which converts a state back into a rank.

2.3 Examples of EPCG

2.3.1 Fling!

In the game *Fling!*, released in 2011 by Bevin Software OU, players aim to clear all pieces from a 7x8 game board by flinging them into each other until none remain. In the original game, a valid board possesses a single sequence of moves (excluding the last move) leading to a solution. An example of a Fling! board is shown in Figure 2.1. In previous work, EPCG was applied to create a system for interactively designing and analyzing puzzles for this game (Sturtevant, 2021).

Assuming that a board for this game has k pieces and n positions to place those pieces (10 and 56 in the example given in the figure, respectively), the total number of ways in which the pieces can be placed on the board is $\binom{n}{k}$ (which also is the max rank). Subsequently, the board ranks that the generator iterates

Figure 2.1: A Fling! board. Image source: https://www.pocketgamer.com/fling/fling-review/

through are from zero to $\binom{n}{k} - 1$, each representing a different configuration of the k pieces on the board.

The evaluator takes in a board and compares the size of its brute-force search tree with the size of its constrained tree that an expert would explore, selecting the puzzles with the highest ratio. The brute-force search tree is the full search tree when attempting to solve the puzzles with no expert knowledge, while the constrained tree is one in which some parts are pruned since an expert can see that they do not lead to the solution. This aids in selecting content that optimizes an expert player's ability to apply their specialized knowledge when solving puzzles.

2.3.2 The Witness

In the 2016 game *The Witness* by Jonathan Blow and Thekla, Inc., players navigate a vast world filled with puzzle panels that they must solve to progress. An example of a puzzle panel from this game is shown in Figure 2.2. To solve a puzzle, the player starts from the lower-left corner of the panel and aims to reach the goal at the upper right corner within a grid without crossing a vertex more than once, and while adhering to specific puzzle constraints. For this game, EPCG was used to exhaustively generate all content for a particular panel size and constraint set (Sturtevant & Ota, 2018).

In *The Witness* puzzle, the arrangement of pieces follows a similar approach to the game *Fling!*, but with the distinction that the order of piece placement matters. For a *Witness* panel with k pieces and n positions to place those pieces, the total number of ways the pieces can be arranged on the panel is k^n. The total number of locations is then the product of the number of locations for the pieces and the arrangement of the piece types, and the total number of configurations is then obtained by multiplying this number by the ways in which constraints

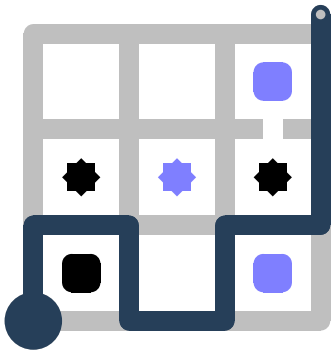


Figure 2.2: A Witness puzzle panel. The dark grey line shows the solution

can be defined. The generator iterates through ranks from zero to $maxRank$, each corresponding to a different panel, with a specific selection of piece types, locations for those pieces, and set of constraints.

The evaluator in this case determines the number of solutions for each puzzle, selecting puzzles with the fewest solutions, and among these, prioritizing the ones with the longest solution lengths. It was found that running this evaluation is time consuming since the search trees for finding solutions can be exceedingly large, for which reason the panels are broken down into smaller sub panels and each solved individually but jointly (i.e. the final solution when all the sub-panels are connected must form a valid connected line). One benefit realized with this approach is that if the evaluation of part of the panel yields a lower value than that of a previously evaluated complete panel, then the new panel being generated can be disregarded. This principle is implemented using the branch and bound search algorithm (Land & Doig, 1960).

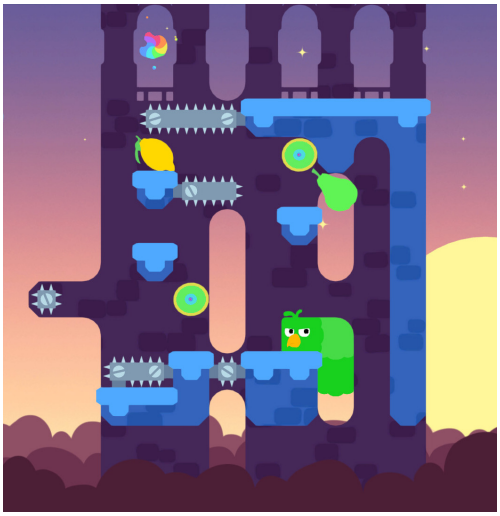


Figure 2.3: A Snakebird level. The player must navigate the green snakebird to eat both the pear and the lemon while avoiding the spikes, before exiting through the rainbow disc in the top left corner. Also, note the two portals shown as green circles, which can be used to teleport the snakebird. Image source: https://store.steampowered.com/app/357300/Snakebird/

2.3.3 Anhinga

The game *Snakebird*, released in 2015 by Noumenon Games, tasks players with guiding one or more snakebirds to consume all the fruit in a level before exiting, while avoiding obstacles in the process. An example of a level is shown in Figure 2.3. EPCG was utilized to generate and evaluate all levels for this game. It was also used for developing a level editor for a clone of the game titled *Anhinga* (Sturtevant et al., 2020b).

The generator in the level editor's EPCG procedure produces all possible single-tile changes in a level, with each tile being either sky, ground, or spikes. To calculate how many such changes there are, assuming we have k piece types that we can place into n locations in the level, the total number of possible changes is k multiplied by n. This means that the generator iterates through ranks from *zero* to $k * n - 1$.

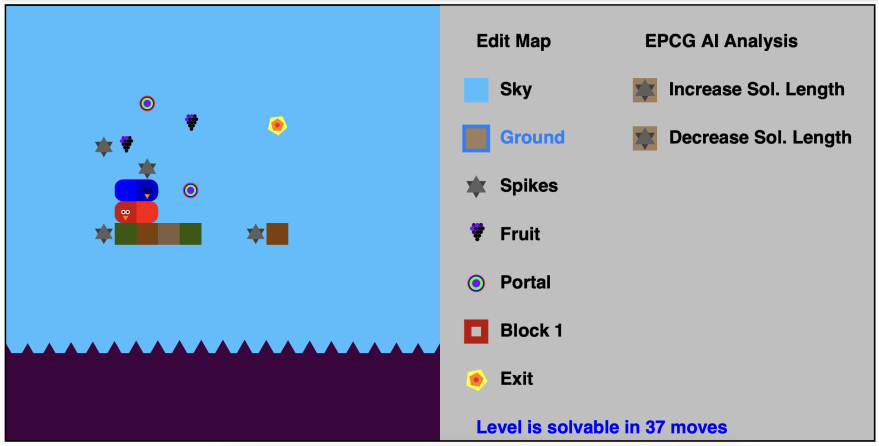


Figure 2.4: Anhinga editor. The panel adjacent to the level view enables the manipulation of gameplay elements or the application of EPCG analysis. Image source: Sturtevant et al. (2020b)

The evaluator employs a breadth-first search to determine the shortest solution to each level considering all potential modifications suggested by the generator. Subsequently, the option is given to select the change that either maximizes or minimizes the optimal solution length. Opting to maximize the solution length typically results in a more challenging level, while minimizing the solution length tends to make levels easier. Figure 2.4 shows the level editor that was built for the game.

2.4 Puzzle Entropy

As described in Section 2.2, an EPCG procedure requires defining both a generator and an evaluator. As seen from the examples provided in Section 2.3, the evaluator has generally been game-specific, implying that the same evaluator cannot be used across multiple games without altering its implementation to suit the rules of each game. This raises the question: Is there a way to define a more general form of an evaluator that can be applied across different games?

A standardized measure of difficulty not only assists in evaluating levels but also can be used to predict whether puzzles will be interesting to players of different skill levels, identify specific skills influencing puzzle difficulty, and aid in organizing and generating puzzles within game frameworks (Nielsen & Smith, 2018). Furthermore, a difficulty metric can facilitate the construction of a puzzle curriculum that requires players to acquire knowledge to advance (Lee et al., 2019). Such a universal measure of difficulty can extend to educational games or broader learning contexts, where the central objective revolves around enhancing student knowledge (Valls et al., 2017). By adjusting the difficulty of problems, one can affect levels of enjoyment and confidence, ultimately leading to heightened engagement and facilitating enhanced learning outcomes (Zhang et al., 2019).

The evaluator in an EPCG procedure takes in a state and returns a value for that state. In the examples above, the property used for evaluating levels tended to correlate, in some way or another, with the difficulty of completing those levels. For the first two examples, *Fling!* and *The Witness*, levels that have the highest perceived difficulty difference between a beginner and an expert were considered the best. In *Anhinga*, the two options for changes given to the user both relate to solution length, one maximizing it and the other minimizing it, and solution length typically reflects the difficulty of a level.

While basic puzzle metrics like solution quantity or solution length provide insights into puzzle difficulty, they are insufficient representations of puzzle complexity. Various methods, such as search and strategic depth scores or constraint satisfaction solvers, have been proposed to measure puzzle difficulty effectively (Shaker et al., 2016).

Information entropy, defined as the measure of uncertainty in random variable outcomes, has been studied in relation to game complexity and player learning.

In previous work, researchers have explored the correlations between entropy and game complexity, as well as its role in facilitating continuous, long-term learning among players (Holmgård et al., 2017).

Entropy quantifies the uncertainty of a random variable Z, which can take k possible outcomes $\{z_1, ..., z_k\}$ (Shannon, 1948). When the probability of an outcome z is represented by a probability function $P(z)$, the formula for information entropy is as follows:

$$H(Z) \doteq \sum_{n=1}^{k} P(z_n) log_2 \frac{1}{P(z_n)} \tag{2.1}$$

For instance, consider a biased double-sided coin that always lands on heads. In this case, there's no uncertainty about the outcome. If we define the random variable $Z_{unfair} = z_{heads}, z_{tails}$, then the probabilities are $P(z_{heads}) = 1$ and $P(z_{tails}) = 0$. Consequently, its entropy is $H(Z_{unfair}) = (1 \log_2 \frac{1}{1}) + (0 \log_2 \frac{1}{0}) = 0$ bits. A fair coin, where heads or tails are equally likely outcomes has probabilities $P(z_{heads}) = 0.5$ and $P(z_{tails}) = 0.5$, resulting in an entropy of $H(Z_{fair}) = (0.5 \log_2 \frac{1}{0.5}) + (0.5 \log_2 \frac{1}{0.5}) = 1$ bit. In simple terms, it would take 1 bit of information to describe the outcomes of a fair coin flip.

In prior work, general techniques were introduced for measuring the uncertainty involved in puzzle-solving (Chen et al., 2023). This uncertainty measure can be regarded as the amount of information required by an oracle, who knows the puzzle's solution, to guide a player fully through each step of puzzle-solving. Information entropy has been employed to represent the uncertainty present at each puzzle state, much like how uncertainty is represented in coin or dice outcomes.

The encountered entropy at a state s can be determined by considering the

number of legal actions available in the puzzle at that state, denoted as $|A(s)|$. A special case is one in which only one legal action is available, which means that there is no uncertainty about the action to take, thus the entropy is 0. Another special case is when there are no possible actions, or $|A(s)| = 0$, it is impossible to encode the solution path from state s in any number of bits, as there are no future outcomes or successor states. In such cases, ∞ is the entropy value. Based on this, a method for evaluating single-player, turn-based puzzles was introduced, which is called Minimum Uniform Solution Entropy (MUSE). MUSE quantifies uncertainty in puzzle-solving by computing the entropy of the solution with the least uncertainty. This method represents the puzzle's uncertainty experienced by a simulated player consistently at each state. Algorithm 2 shows simple pseudo code for this algorithm.

Algorithm 2 MUSE(S_{start}, σ)

```
entropies ← [ ]
for each s_start ∈ S_start do
    entropies.Append(MinPathEntropy(s_start, σ))
return min(entropies)
```

Algorithm 3 MinPathEntropy(s, σ)

```
if σ(s).count == 1 then
    return 0
if σ(s).count == 0 then
    return ∞
childEntropies ← [ ]
for each succ ∈ σ(s) do
    successorEntropies.Append(MinPathEntropy(s, σ))
localEntropy ← H(Z_|σ(s)|)
return min(successorEntropies) + localEntropy
```

The process begins by calculating the entropy for all possible start states to determine the minimum among them. The entropy measured is the minimum

path entropy, representing the entropy value of the solution path for a state with the least entropy. To compute this value, a recursive function is utilized, as depicted in Algorithm 3. This function takes a state and a successor function (denoted as σ) as input and yields the minimum path entropy. Initially, the function evaluates the number of successors for the input state. If there is only one successor, the entropy for that state is zero; if the state has no successors, its entropy is infinity, as discussed earlier.

An important point to note here is that the list of successors need not be exhaustive; that is, it does not need to include all possible successors to a state. This is because if any form of player modeling is involved, we have to assume that players may not examine all successors. Based on their skill level, they may consider some states unnecessary to explore because they would not lead to the solution. This concept is implemented through inference rules, which aim to reduce the uncertainty encountered in logic puzzles. Inference rules are utilized to model the uncertainty experienced by skilled players while solving puzzles. In the provided pseudo code, inference rules are implemented as part of the successor function and are used to remove some of the successors. Inference rules will be discussed in more detail in Chapter 5.

The resulting entropy of a state s comprises the sum of the local entropy, determined by the number of actions $|A(s)|$, and the entropy associated with the successor state having the lowest entropy.

Chapter 3

Building the Puzzle

In this chapter, we introduce the problem domain and define the puzzle game framework that we will be working with for the remainder of the study. Afterwards, we discuss the process of generating all solutions to the puzzle game. Figure 3.1 highlights the steps of the process covered in this chapter.

3.1 Defining Puzzle Framework

The first step in the process involves defining the puzzle framework. In this study, we focus on a specific type of puzzle, which belongs to the category known as tangram puzzles. The tangram is a puzzle that comprises multiple flat polygon

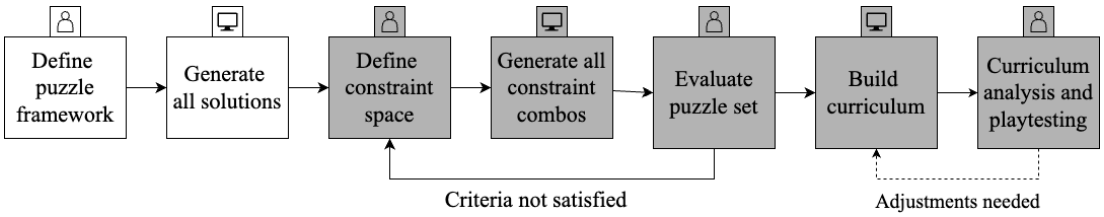


Figure 3.1: Steps covered in chapter 3.

Figure 3.2: The Ostomachion, an old instance of tangram puzzles (source: https://en.wikipedia.org/wiki/Ostomachion). Notice the variety of shapes and sizes among the pieces.

pieces assembled to create various shapes. Players aim to replicate a given pattern, often provided as an outline with some or no initial pieces placed, using all pieces without overlap. The example shown in Figure 3.2 is an instance of this puzzle called The Ostomachion, which was developed by the Greek mathematician Archimedes more than two thousand years ago and found in the Archimedes Palimpsest (Morelle, 2007). Popularized in China in the early 1800s, tangram puzzles gained popularity in America and Europe through trading ships shortly thereafter (Slocum, 2003). Renowned worldwide, the tangram serves multiple purposes, including entertainment, artistic expression, and as an educational tool (Forbush, 1914; Slocum, 2001; Campillo-Robles et al., 2022).

The first step in designing the instance of the tangram puzzle we will be working with is selecting the shape of the board (the outline) and the pieces. For this, we build on previous work that explored a tangram puzzle with a hexagon-shaped board (Blanchonnet, 2021). The building blocks of the board are triangles, as can be seen inside the board shown in Figure 3.3. The board contains 54 triangles in total.

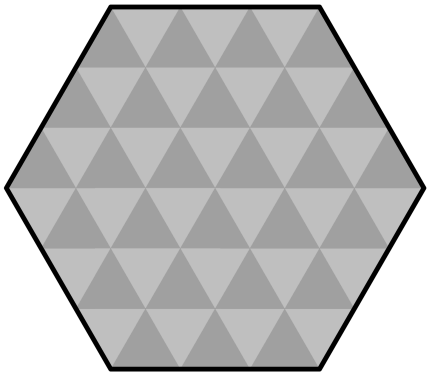


Figure 3.3: Puzzle board. Notice the small triangles composing it. Some triangles are shaded to improve visibility.

Given that the board units are triangles, it follows that the units comprising the pieces must also be triangles. At this point, we face a decision: either conduct the analysis for all potential piece shapes and combinations, an endeavor deemed prohibitively costly, or opt for a subset of shapes sharing a particular attribute. Following the previous work that investigated this puzzle, we choose the latter option. The attribute chosen to be enforced was that all pieces must consist of "Trapezoids." A trapezoid comprises three adjacent triangles, as illustrated in Figure 3.4.

Using the trapezoid as the foundation for the pieces ensures the validity of all potential piece shapes. Since a trapezoid is a valid shape that can fit onto the board, any pieces constructed from trapezoids must also be valid, unless they

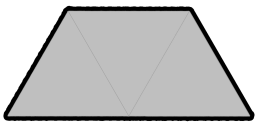


Figure 3.4: The trapezoid piece.

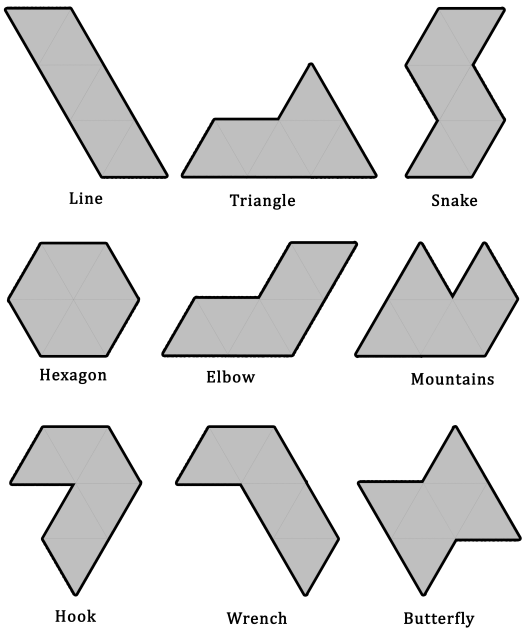


Figure 3.5: The selected set of pieces and the names assigned to them (excluding the "Trapezoid" piece). Notice how they are all composed of two trapezoids in some configuration.

exceed the board's capacity. This yields a total of 10 different possible piece shapes, inclusive of the trapezoid itself, each comprising either one trapezoid or two adjacent ones. Figure 3.5 shows the nine pieces composed of two trapezoids each. We opted against any pieces larger than that, as they would occupy excessive space on the board, resulting in fewer potential board configurations, as we will explore later in Section 3.2.

At this point, we have a board of size 54 and pieces consist of trapezoid pieces with a size of 3 triangles and two-trapezoid pieces with a size of 6 triangles. With this setup, we infer that the board can accommodate a maximum of 54 / 3 = 18 pieces (if all are trapezoids) or a minimum of 54 / 6 = 9 pieces (if all are larger pieces).

We now need to determine the types of pieces and how many instances of each

to provide to the player. Following the previous work, we opted for a piece set comprising all nine larger pieces and two of the small trapezoid pieces, resulting in a total of 11 pieces. However, this results in having 9 * 6 + 2 * 3 = 60 triangles, which exceeds our board's size by six triangles. This means that in each solution, either one of the nine larger pieces or both trapezoids have to be marked as "Forbidden", meaning they must not be included in the solution.

3.2 Generating All Solutions

Now that both the board and piece descriptions are complete, we move to the next step in the process. This step involves generating all solutions to the puzzle, which entails generating all possible configurations of pieces that form a complete board without any excess (i.e., pieces extending beyond the board's boundaries, which would not occur if the board is filled).

At this stage, we can conduct an EPCG analysis to generate all the solutions to the puzzle. To do this, we must define both the generator and the evaluator. As discussed in Section 2.2, in EPCG, for the generator we define both the maximum rank and the unranking function. In this puzzle, the maximum rank represents the total number of configurations for all the pieces in the set on the board.

As previously mentioned, using all 11 pieces in a solution would exceed the board's size of 54. Instead, we must always exclude one piece from the solution (or both trapezoid pieces). Following that, there are 10 distinct sets of solutions, each set excluding one of the piece types.

In order to determine the maximum rank, we first need to obtain all possible locations for each piece. To achieve this, we execute a search that systematically scans the board using the piece, from left to right and top to bottom, incrementing

either the horizontal or vertical position by one at each step. Figure 3.6 illustrates piece movements.

It is important to note that each piece can also rotate, as illustrated in Figure 3.7. This rotation property expands the potential locations for pieces further, as for each piece we must now consider all six possible angles of rotation: 0, 60, 120, 180, 240, and 300 degrees.

It is worth noting that certain pieces appear identical when rotated by 0 or 180 degrees. We refer to these pieces as "rotationally-symmetric" pieces, which are depicted in Figure 3.8.

We also observe another interesting property of the pieces that was mentioned in the previous study: their ability to be flipped (Blanchonnet, 2021). This aligns with our intention to create a physical version of the puzzle, where the pieces can be physically flipped and positioned on the board with their reverse side. The illustration in Figure 3.9 demonstrates how pieces appear when flipped.

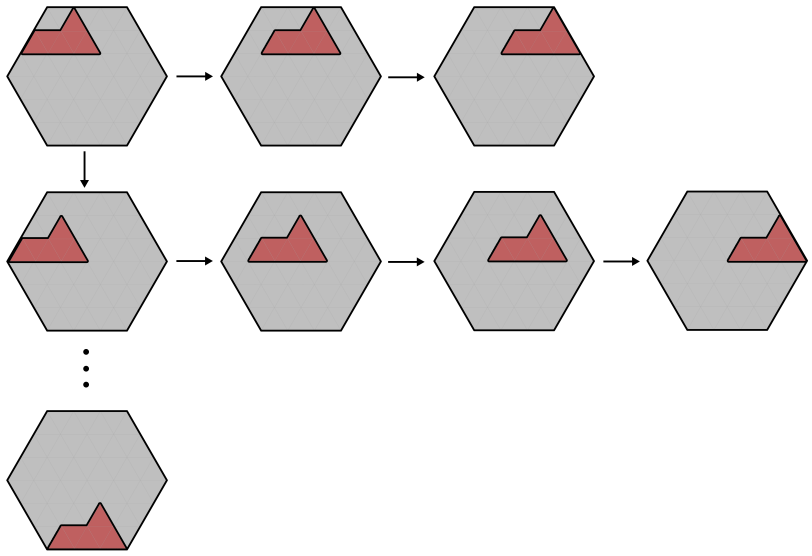


Figure 3.6: Scanning the piece board locations.

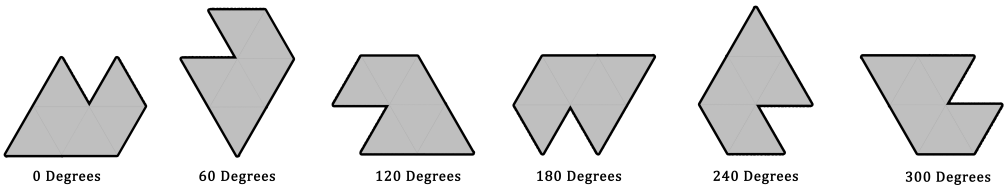


Figure 3.7: Different angles of rotation for a piece.

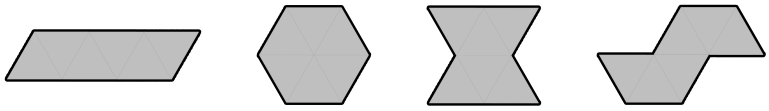


Figure 3.8: Rotationally-symmetric pieces.

Similar to the case with rotation, we observe that certain pieces appear identical when flipped. These pieces are termed "flip-symmetric" pieces, depicted in Figure 3.10.

After outlining all the possible movements for a piece, we proceed to list all the potential locations for each piece. Using this information, we obtain the maximum rank for each set of pieces by simply multiplying the number of locations of each piece in the set. Table 3.1 shows the number of locations for

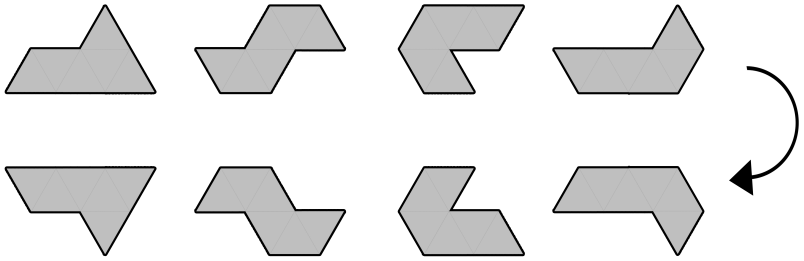


Figure 3.9: Examples of flipping pieces.

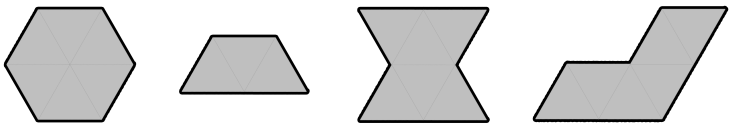


Figure 3.10: Flip-symmetric pieces.

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

Table 3.1: Number of locations for each piece type.

each of the ten piece types. With this data, the maximum rank for the puzzle ranges between 1.372×10^{17} and 1.386×10^{18}, depending on which pieces are in the set. The numbers within this range are prohibitively large, making exhaustively generating all states in this way infeasible. That is why we chose to formulate the generator as a depth-first search algorithm. This also means it is unnecessary to explicitly write an unranking function since we will not be iterating through the ranks in the usual manner.

The next step involves defining the evaluator that we will be using. In this case, the evaluator will check whether each state constitutes a valid solution. If so, it is added to our list of generated solutions. A valid solution state is one in which all pieces in the set are placed on the board without overlap, filling up the

entire board.

With both the generator and the evaluator in hand, we can begin the EPCG process to systematically generate all the solutions. However, it is important to note that each solution will omit one of the piece types. Therefore, we execute 10 different searches, each with a piece set that disregards a different piece type. As mentioned earlier, we perform a depth-first search to iterate through all content in a more efficient manner.

At the start of each depth-first search, we select one of the pieces (excluding the forbidden one) and identify all possible locations where the piece can be positioned. Subsequently, for each location of the first piece, we find all valid locations for the second piece (without displacing the first). This process is repeated for all pieces until either a solution is reached (all pieces are successfully placed) or a state is encountered where none of the remaining pieces can be placed without making the state invalid (due to overlapping or out-of-bounds pieces).

Formulating this as a depth-first search offers the advantage that, at least for this specific puzzle, we can evaluate partial states to determine whether they would lead to a valid solution. In other words, a fully exhaustive generator would generate all states regardless of their validity, but when using a depth-first search, invalid states are pruned immediately. This is because during the search, when placing pieces, we make sure not to place them in locations where they would overlap with other pieces. The pseudo code in Algorithm 16 shows a simplified version of the algorithm.

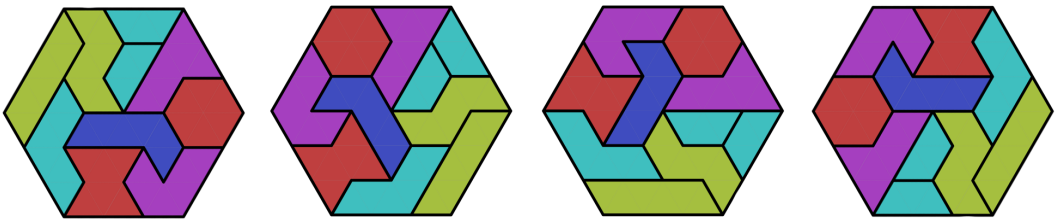


Figure 3.11: Duplicate solution states. Notice how from each of the shown solutions one can get to any of the others by rotating and/flipping the solution one or more times.

Algorithm 4 FindAllSolutions()

```
1:  solutions ← [ ]
2:  depth ← 0
3:  board ← CreateEmptyBoard()
4:  actions[0] ← board.GetActions()
5:  while depth > 0 or actions[depth].count > 0 do
6:      if actions[depth].count > 0 then
7:          board.apply(actions[depth].last)
8:          depth ← depth + 1
9:          actions[depth].Add(board.GetActions())
10:     else
11:         depth ← depth − 1
12:         board.UndoAction(actions[depth].last)
13:         actions[depth].Pop()
14:     if SolutionValid(board) and ¬solutions.Contains(board) then
15:         solutions.Add(board)
16: return solutions
```

In the search algorithm for finding the solutions, as seen in Figure 3.12, we begin by creating an empty board and then finding all the possible actions for that board. The list of actions contains all the possible single-piece configurations of the board for all piece types in the piece set. After that, we take an action from the list, apply that action, and then generate the list of all possible actions succeeding it and put them at a higher depth index in the list. Note that the

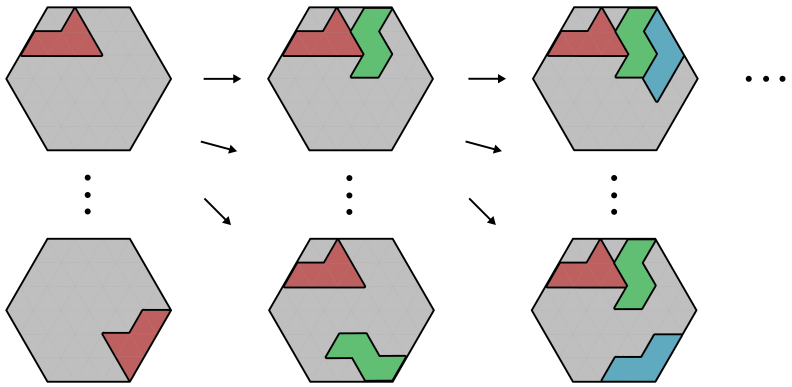


Figure 3.12: Depth-first search tree.

succeeding actions contain the piece from the first action. We repeat the process and propagate deeper into the tree until we reach a point where no other pieces can be placed. At this point, we check whether the board constitutes a valid solution, and if so, whether we already have that solution in our solutions list. The inclusion test also considers all the possible rotations of the solution and attempts to flip it and rotate it as well to ensure that we do not have duplicate solutions in our list (Figure 3.11 illustrates duplicate solutions). In either case, whether the board constitutes a valid solution or not, if no more pieces can be placed, we undo the last action performed and iterate through the rest of the actions at the previous depth index. Once we reach the root, which is the empty board, with no more actions left to apply, the search ends, and we return the list of solutions.

After running this for all the possible piece sets, we end up with the number of solutions shown in Table 3.2.

The search procedure was implemented in C++ and compiled using Xcode. Executing the complete search takes approximately 9 minutes and 40 seconds on an Apple M1 MacBook Air 2020 with 8 GB of RAM. The total number of

Forbidden piece	Solutions
Hexagon	673
Butterfly	352
Wrench	321
Snake	307
Mountains	265
Hook	179
Triangle	176
Elbow	129
Line	97
Trapezoid	9
Total	**2508**

Table 3.2: Number of solutions found with each forbidden piece type.

expansions is $392,326,063$, which constitutes $0.283 \times 10^{-8}\%$ of the number of expansions that would have been necessary in the naive exhaustive case.

We note that we ran an experiment here after the study was complete, in which we utilized the inference rules that will be discussed in Chapter 5 to reduce the number of successors from each state. This resulted in an 86.52% speedup, meaning that the complete search now takes only approximately one minute and 18 seconds when run on the same setup as earlier. With the total number of expansions being $2,595,601$ in this case.

Chapter 4

Constraint Analysis

In this chapter, we begin by introducing the two types of constraints chosen for the puzzle. Then, for each of the two constraint types, we discuss the process of defining the constraint type, generating all relevant constraint combinations, and evaluating the set of puzzles resulting from those combinations. The constraints generated in this stage are used later to construct the chapters in the curriculum.

4.1 Constraint Types

Following the generation of all solutions, we now have 2508 solutions in total. The next step involves defining constraints that could diversify the chapters within our curriculum, as highlighted in Figure 4.1.

Regarding the constraints, we define two general types. Firstly, there are constraints that pertain to the relationship between the piece and the board. These constraints restrict the number of locations on the board where a piece can be placed. For instance, specifying that a particular piece cannot be flipped, or that it can only be placed on the lower half of the board and not the upper.

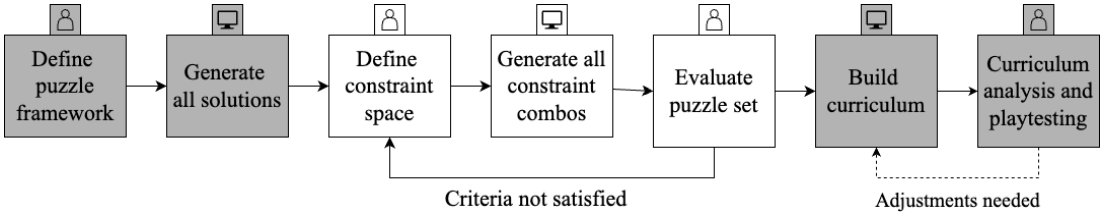


Figure 4.1: Constraint analysis section of the architecture.

We call this type of constraints “Placement constraints”.

The second type of constraint involves the interplay between the pieces themselves. These constraints dictate the locations a piece can occupy based on the placement of another piece on the board. For instance, one constraint might require that a piece be adjacent to another piece, while another might specify that they cannot be adjacent for the solution to be valid. We call this type of constraints “Adjacency constraints”.

Given that these two constraint types are independent of each other, we execute the loop depicted in Figure 4.1, which involves defining the constraint space, generating all constraint combinations, evaluating the resulting puzzles, and making adjustments. We repeat this process twice, once for each of the two constraint types. However, it’s worth noting that this is an iterative process. Adjustments to the constraint definitions may be necessary. This is to address aesthetic concerns and/or to further limit the size of the constraint space.

4.2 Placement Constraints

As previously mentioned, the board we intend to use is physical, which enables us to establish distinct functions for each side, as shown in Figure 4.2. Consequently, in this study, we apply placement constraints to one side of the board and not the other.

Figure 4.2: The two sides of a physical board.

We start by defining a constraint pattern, which is a configuration of individual piece constraints. A piece constraint restricts which locations on the board a piece can occupy. This can be achieved by altering the design of pieces and certain locations on the board to limit the possible placements for a specific piece, as will be discussed shortly. An example of a constraint pattern is one that allows the "Hexagon" piece to be placed anywhere while restricting all other pieces to one or two specific locations. Bearing this in mind, we proceed to conduct a comprehensive analysis of all such patterns and determine the number of valid solutions for each.

To execute EPCG and exhaustively generate and evaluate all the possible constraint patterns, we must define both the generator and the evaluator. For the generator, we need to devise an unranking function and establish a $maxRank$ to apply with that function. In this case, $maxRank$ corresponds to the number of constraint patterns, which we will discuss momentarily. As for the unranking function, it is used to extract the constraint configuration for each piece from the integer index, for which we use modulo operations. As for the evaluator, it checks how many of our total of 2508 solutions satisfy each constraint pattern.

Algorithm 5 PlacementConstraintSpaceSearch(solutions)

```
1: mostPatternSolutions ← 0
2: for pattern = 0 to numPatterns − 1 do
3:     patternSolutions ← 0
4:     for i = 0 to solutions.size() − 1 do
5:         if SolutionValid(solutions[i], pattern) then
6:             patternSolutions + +
7:     if patternSolutions > mostPatternSolutions then
8:         mostPatternSolutions = patternSolutions
9:         bestPattern = pattern
10: return bestPattern
```

As illustrated in Algorithm 5, the complete loop operates as follows: for each constraint pattern, and then for each of our 2508 solutions, we check whether that solution is valid given the placement-constraint pattern. During the execution of the procedure and within the top-level loop (which iterates over the patterns), we keep track of the number of valid solutions for each specific pattern. Our criteria involve selecting the pattern that yields the greatest number of solutions. We then use the evaluator to satisfy this criterion.

4.2.1 First Iteration

To determine the number of ways to define the placement constraint patterns for all pieces, we first need to understand how many constraints can be applied to each individual piece. For a piece P with N_P distinct locations on the board (accounting for all rotations and flipping the piece), there is $\binom{N_P}{0} = 1$ way to constrain the piece to none of its possible locations. Similarly, for constraining the piece to *one* location, there are $\binom{N_P}{1} = N_P$ ways. Then, for two locations, there are $N_P \times (N_P - 1)$ ways, or $\binom{N_P}{2}$. For three locations, it is $\binom{N_P}{3}$, continuing until we reach $\binom{N_P}{N_P}$ for all N_P locations, which evaluates to *one*. This yields the

total number of possible placement constraints C_P for piece P as follows:

$$C_P = \binom{N_P}{0} + \binom{N_P}{1} + \binom{N_P}{2} + ... + \binom{N_P}{N_P} = 2^{N_P} \tag{4.1}$$

Combining this with all the other $M - 1$ pieces in the set we end up with:

$$C = C_{P1} \times C_{P2} \times ... \times C_{PM} = 2^{N_{P1}} \times 2^{N_{P2}} \times ... \times 2^{N_{PM}} \tag{4.2}$$

This expression evaluates to $10^{4.13 \times 10^{16}}$ when evaluated for the set of pieces that does not contain the "Hexagon" piece, a number far greater than the number of atoms in the observable universe (estimated to be 10^{82} based on recent estimates (Planck Collaboration et al., 2016)). Evaluating each pattern individually to see how many solutions satisfy it is infeasible given this magnitude. However, we observed that such an approach might not be necessary, as many of these patterns appear to be somewhat arbitrary.

4.2.2 Second Iteration

One of our primary considerations in designing placement constraints is the practicality of implementing them on a physical puzzle. One straightforward approach involves incorporating physical bumps onto the board's triangles, allowing only pieces with corresponding holes of the same shape to fit. An example of this is shown in Figure 4.3. This method can accommodate various bump shapes and sizes, and can even permit bumps to accept multiple hole shapes, and vice versa.

To simplify the process, we opted for a single size and shape for both the bumps and holes. We chose a hemisphere as the shape because we believed it would facilitate the fitting of bumps into holes in the physical puzzle. With this

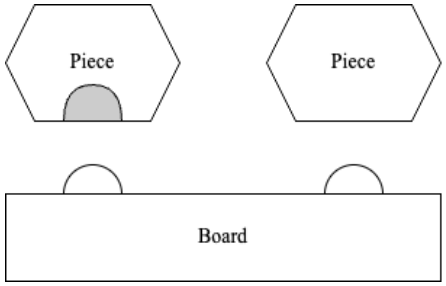


Figure 4.3: Bumps and holes. Notice how the piece on the left would fit onto the board because the bump can fit inside the hole, while the flat piece on the right would not. Note, however, that in the physical puzzle, no triangles are actually flat; flat piece triangles have bumps on them, and flat board triangles have holes on them. This design helps make the pieces more stable on the board when placed and does not affect the analysis.

decision, we can now calculate the number of ways to place bumps on the board using the following equation:

$$Patterns_{bumps} = 2^{boardTriangles} = 2^{54} \tag{4.3}$$

Where $boardTriangles$ corresponds to the number of triangles on the board. This equation is derived from the fact that for each triangle, there is a binary decision: whether to place a bump or not. When considering all triangles simultaneously, the result is two to the power of the number of triangles.

Likewise, considering the pieces, each piece either has 12 (for the nine larger pieces) or 6 triangles (for the two trapezoid pieces) when both its sides are taken into account. Consequently, we arrive at the following number of ways to assign holes to a piece's triangles for a piece P1:

$$holes_{P1} = 2^{T1} \tag{4.4}$$

Where $T1$ is the number of triangles in piece P1. Which yields the following

as the total number of patterns for the pieces:

$$Patterns_{holes} = holes_{P1} \times holes_{P2} \times ... \times holes_{PM} = 2^{12\times9} \times 2^{6\times2} = 2^{120} \quad (4.5)$$

Combining the two leads to the following as the total number of patterns:

$$Patterns_{total} = Patterns_{bumps} \times Patterns_{holes} = 2^{54} \times 2^{120} = 2^{174} \simeq 2.39 \times 10^{52} \quad (4.6)$$

4.2.3 Third Iteration

While this figure is significantly smaller than what we had before, it remains too large to explore exhaustively, necessitating further restrictions on the patterns. To address this, we opted to implement a checker board pattern for bumps on the board. Figure 4.4 illustrates the two possible patterns we derive for the board, which are equivalent when one of them is rotated by 60 degrees - resulting in $Patterns_{bumps}$ being reduced to just *one*.

Regarding the pieces, in addition to alternating hole configurations, and to broaden the range of patterns to explore and select from, we allow each of the two sides of a piece to have one of the following configurations: “full” (holes on all triangles), “even” (alternating holes, starting from a hole in the bottom left triangle), “odd” (alternating holes, starting from a flat triangle in the bottom left), or “none” (no holes in any of the triangles). As each side can have a configuration independent of the other side, we end up with 16 possible overall configurations for each piece. This yields the following number of patterns for

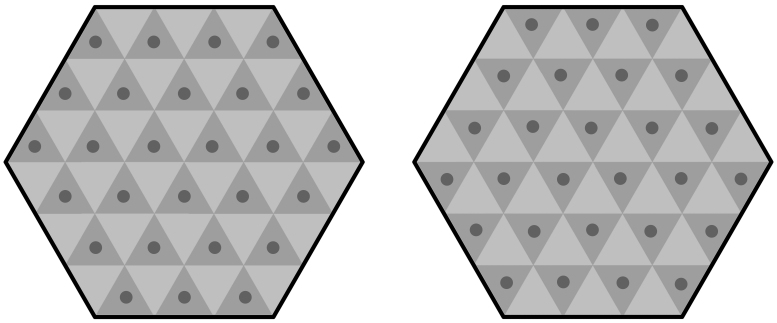


Figure 4.4: Board constraint patterns. The bumps are shown as circles and the triangles containing them are shaded for visibility.

holes:

$$Patterns_{holes} = Configs^{M} = 16^{11} = 2^{44} \tag{4.7}$$

Prior to computing the total, it is essential to acknowledge that one of the sixteen possible configurations for each piece involves having no holes on either side. Applying such a configuration renders the piece unable to fit anywhere on the constrained side of the board. Consequently, any pattern containing this configuration will not yield valid solutions, considering the distribution of bumps on the board as illustrated earlier. This is why we opted to exclude this configuration. Therefore, $Patterns_{holes}$ is effectively reduced from 16^{11} to 15^{11}.

We now calculate the total number of patterns as follows:

$$Patterns_{total} = Patterns_{bumps} \times Patterns_{holes} = 1 \times 15^{11} = 15^{11} \simeq 8.511 \times 10^{12} \tag{4.8}$$

This number is more manageable than what we had earlier, so we attempted to generate and evaluate all the constraint combinations of this type. The estimated execution time for this procedure is around 20 hours in our implementation. However, we first tested it on a smaller portion of the state space for debugging purposes. During this test, we noticed that there were still some symmetries that, if detected and discarded, would make the state space much smaller. Therefore, we looped back to the step of defining the constraint space.

4.2.4 Fourth Iteration

As previously mentioned, certain pieces exhibit forms of symmetry that become apparent when specific transformations are applied to them. For instance, the "Hexagon", "Butterfly", "Elbow", and "Trapezoid" pieces appear identical when flipped (rotated 180 degrees around the x-axis), while the "Hexagon", "Butterfly", "Snake", and "Parallelogram" pieces maintain their appearance when rotated 180 degrees.

Now, we can leverage these identified symmetries to further shrink our pattern space. Consider the "Elbow" and "Trapezoid" pieces from the "flip-symmetric" category for now (the "Hexagon" and the "Butterfly" present special cases, which will be addressed later). For these two pieces, we observe that both sides are equivalent; the player can simply flip the piece and use the less restrictive side or the one that facilitates placement on the board in that particular location. This concept is clarified further with a graph, as illustrated in Figure 4.5.

This leads to the conclusion that it would suffice to require those pieces to have the same hole configuration on both sides, effectively limiting their number of configurations to 3. Note that it is not 4 because having no holes on either side is not a considered configuration because it disallows the piece from being

placed anywhere on the constrained side of the board.

Regarding rotation-symmetric pieces, we can observe that rotating a piece by 180 degrees results in the piece retaining the same shape. However, this rotation also causes its hole configuration to switch between odd and even if the original configuration was either odd or even. Figure 4.6 illustrates this.

With this consideration, pieces exhibiting such symmetry do not gain any distinction from having odd or even hole configurations on either of their sides. In other words, assigning either side an odd or even configuration would effectively result in a "full" configuration. This holds true due to the specific arrangement of bumps on the board; if the board had a different bump pattern, the resulting configuration might not equate to a "full" configuration. However, in our case, we can exploit this to eliminate both even and odd configurations from both sides of the corresponding pieces. This reduces the total configurations for those pieces from 15 to 3.

Now focusing on the two pieces possessing both types of symmetry, namely the "Hexagon" and the "Butterfly", we first consider flip symmetry. It follows that these pieces must have the same hole configuration on both sides: odd, even, or full. Additionally, considering their rotational symmetry, we can discard both

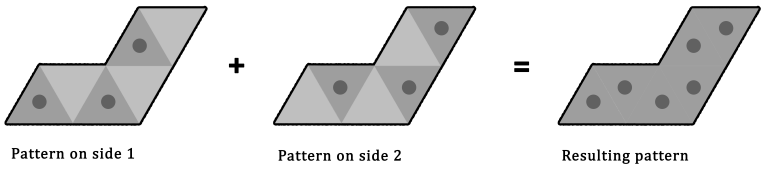


Figure 4.5: Flip-symmetric pruning. The piece has an "odd" hole pattern on one side and an "even" one on the other side. The resulting pattern effectively is shown to the right.

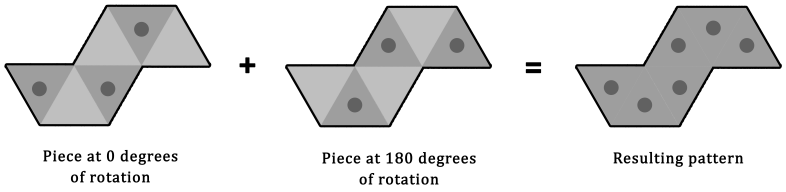


Figure 4.6: Rotation-symmetric pruning. The piece has an "odd" hole pattern on one of its two sides. The first graph shows the pattern on the piece at zero degrees of rotation, the second one shows the pattern when we rotate the piece by 180 degrees. The graph on the right shows the resulting pattern, effectively.

odd and even configurations as they effectively result in a "full" configuration. Therefore, for these two pieces specifically, the number of valid configurations is reduced to only *one*.

After taking this analysis into consideration, we end up with the following final estimate for the true number of patterns:

$$Patterns_{total} = 1 \cdot 1 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 15 \cdot 15 \cdot 15 \cdot 15 = 4,100,625 \tag{4.9}$$

We again attempted to test the generation on a subset of the state space and evaluated a few constraint patterns. When evaluating the those patterns, we observed that it was possible to limit those patterns to ones that were more intuitive and would better connect with the symmetry types we discussed earlier. Our plan was to break down the piece types into five groups based on the types of symmetries they exhibit. We have two pieces that are symmetric in both ways, two that are solely rotationally symmetric, two that are solely flip-symmetric, and four that lack any type of symmetry.

4.2.5 Fifth Iteration

For the four non-symmetric pieces, we opted to divide them into two groups, each containing two pieces. Following this division, we decided to constrain the hole configurations for the first group to one of four possibilities: odd and none, even and none, none and odd, or none and even (for their respective sides). This means that one side will have either an odd or an even hole configuration, while the other side will have no holes at all. This restriction reduces the number of possible configurations for a piece in this group to just *four*. As for the other group, we restrict them to have one of the following configurations: odd and even, odd and odd, even and odd, or even and even (for their respective sides). Here, we limit each side to have either an odd or an even hole configuration, resulting in each piece in this group to also have only *four* configurations.

With the newly added stipulations to the pieces with no symmetry, the total number of patterns reduces to the following:

$$Patterns_{total} = 1 \cdot 1 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 4 \cdot 4 \cdot 4 \cdot 4 = 20,736 \tag{4.10}$$

It is crucial to note that there are now 12 ways to select which of the four pieces are assigned to each of the two-piece groups. This factor must be considered during the exhaustive analysis. Consequently, the number of patterns effectively increases to $248,832$.

4.2.6 Final Evaluation

After executing the EPCG procedure to generate and evaluate all constraint patterns, we arrive at the results shown in Figure 4.7. The figure illustrates

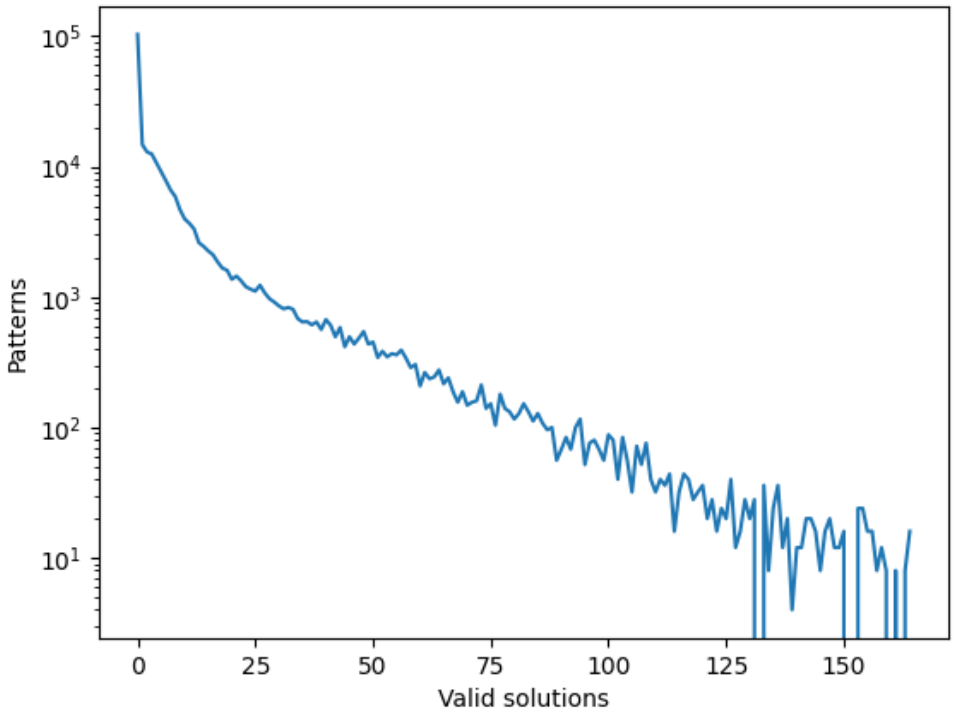


Figure 4.7: Distribution of patterns in relation to the number of valid solutions they have. The x-axis represents the number of solutions, while the y-axis denotes the number of patterns leading to each respective number of solutions.

the distribution of patterns based on the number of valid solutions each pattern yields. We observe that over 40% of the patterns have no valid solutions. Subsequently, the frequency of patterns decreases exponentially for each increment in the number of solutions until only 16 patterns are observed, with the maximum solution count of 164.

Running the complete analysis takes about 35 minutes in XCode on an 8-threaded Apple M1 MacBook Air 2020 with 8 GB of RAM.

Upon conducting this analysis, we found that the pattern that has the highest number of solutions has 164 distinct solutions in total. In essence, when applied

physically to the pieces, this pattern allows reaching exactly 164 out of the total 2508 solutions.

4.3 Adjacency Constraints

We now shift our focus to constraints that govern the interaction between the pieces themselves. Bear in mind that this is another iteration of the three steps shown in Figure 4.1 pertaining to constraint analysis. Here, we consider the different ways in which one piece may influence the viability of placing another piece in various locations on the board. Exhaustively analyzing this proves to be tedious, as it could entail situations where a piece's impact on others varies based on the placement of specific pieces on the board. We observe that such complexity would result in an arbitrary array of constraint definitions that would be difficult to communicate to the player.

To tackle this challenge, we revert to one of our core criteria: crafting a puzzle that introduces its constraints in a user-friendly manner, particularly for beginners. To achieve this, our definitions for these constraints must be clear and straightforward. Hence, we opt to implement piece-to-piece constraints based on adjacency because we deem that to be simple enough for users to understand.

Now, to define adjacency, we must delineate what constitutes two pieces being adjacent. To address this, we define two types of adjacency. Firstly, edge adjacency denotes two pieces sharing at least one edge, or more precisely, sharing a minimum of two vertices. The other type of adjacency we introduce is where two pieces are connected at their corners, or more precisely, they share exactly one vertex. Figure 4.8 aids in visualizing the two adjacency types.

With these two types of adjacency defined, we now require a method to convey

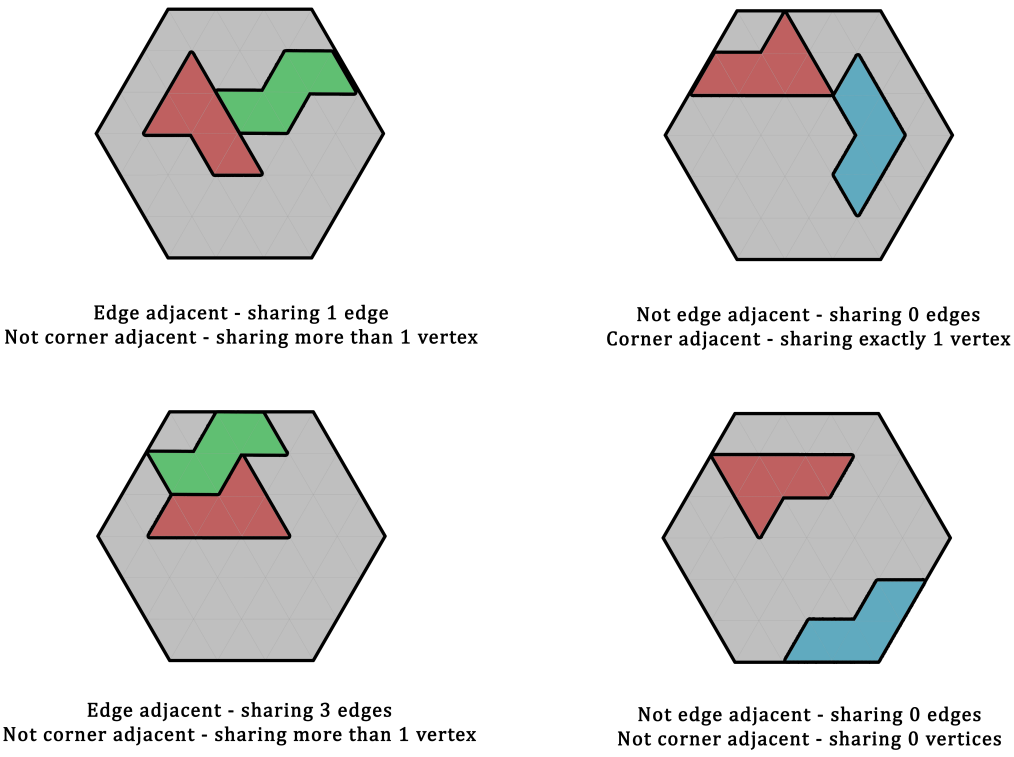


Figure 4.8: Types of adjacency.

them to the player effectively. One approach is to assign a unique color to each piece and then specify, for instance, that a red piece must not share an edge with any green piece. Furthermore, we can allow multiple pieces to share the same color. With this setup, we can start exploring this new constraint pattern space.

Now, concerning the constraints themselves, we have up to four constraint types resulting from the two types of adjacency: pieces of colors A and B must share an edge, pieces of colors A and B must not share an edge (but can share a corner), pieces of colors A and B must share a corner (but cannot share an edge), and pieces of colors A and B must not share a corner (nor an edge, for that matter). It is worth noting that the second constraint type effectively combines the third and fourth. However, we opt to retain it. This decision is to have more chapters in the curriculum, providing greater breadth. We discuss this further in Chapter 5. Therefore, we have five distinct constraint types between colors, either with no constraints between them or with one of the aforementioned four.

Considering that we have five distinct constraint types that can be applied between any of these colors (even within the same color group, as we will discuss shortly), we arrive at the total number of patterns, which also represents the number of possible constraint rules for puzzles in a chapter, as follows:

$$Patterns_{adjacency} = Perms_{colors} \times (Perms_{colors} - 1) \times 5 \tag{4.11}$$

The term $Perms_{colors}$ denotes the number of ways we can assign colors to pieces, as we will see shortly.

Now, we proceed to calculate the total number of patterns in this color adjacency constraint space. We begin by determining the number of ways we can assign colors to pieces, as given by the following equation.

$$Perms_{colors} = Perms_{1color} + Perms_{2colors} + ... + Perms_{11colors} \tag{4.12}$$

For one color, this evaluates to 1 permutation. For two colors, the number of permutations is given by:

$$Perms_{2colors} = \binom{N}{1} + \binom{N}{2} + ... + \binom{N}{N-1} = 2^N - 2 \tag{4.13}$$

Where N is the number of pieces (eleven in our case). For three colors:

$$Perms_{3colors} = (N-1)\times(2^{N-1}-2)+(N-2)\times(2^{N-2}-2)+...+(N-(N-1))\times(2^{N-(N-1)}-2) \tag{4.14}$$

With just three colors, we can observe exponential growth in the number of possible permutations. Therefore, we decided not to explore this exhaustively but instead to assign colors to pieces based on the symmetry groups discussed earlier. Each group of pieces is assigned the same color. We arrived at this decision with the expectation that it would be more intuitive to assign pieces from the same symmetry group to the same color.

This means that we now have a total of five colors. When calculating the number of different constraints we can apply between pieces of the same color in a puzzle, it evaluates to 4 (constraint types, excluding the "no constraint" type) multiplied by 5 (number of colors), which is 20. As for constraints between pieces of different colors, we have 5 (for the choice of the first color) multiplied by 4 (for the choice of the second color) multiplied by 4 (constraint types), resulting in 80 combinations.

Our goal now is to combine this with the previous analysis of placement constraint patterns. We aim to find a placement constraint pattern (bumps and holes) that would not only result in having the largest number of valid solutions but also have enough solutions that are valid given all the newly defined color constraint types, resulting in a large enough pool of candidate puzzles for each chapter in the curriculum. We modify the evaluator to achieve this.

The pseudocode shown in Algorithm 6 shows how we analyze all the adjacency constraint patterns given the list of solutions that are reachable by the selected placement constraint pattern.

Algorithm 6 AdjacencyConstraintSpaceSearch(solutions)

```
1: maxMinBin ← 0
2: bestPattern ← null
3: for placementPattern = 0 to numPlacementPatterns do
4:     solutions ← FilterByPattern(allSolutions, placementPattern)
5:     puzzles ← [[], [], [], []]
6:     for t = 1 to 4 do
7:         for i = 0 to solutions.size() − 1 do
8:             s ← solutions[i]
9:             valid ← true
10:            for clr1 = 1 to 5 do
11:                for clr2 = 1 to 5 do
12:                    if clr1 == clr2 and s.numPiecesOfColor[clr1] == 1
   then
13:                        continue
14:                    for p1 in s.piecesOfColor[clr1] do
15:                        for p2 in ¬s.piecesOfColor[clr2] do
16:                            if t == 1 and g.edge(p1, p2) then
17:                                valid ← false
18:                            if t == 2 and s.edge(p1, p2) then
19:                                valid ← false
20:                            if t == 3 and (s.edge(p1, p2) or ¬s.vertex(p1, p2)
   then
21:                                valid ← false
22:                            if t == 4 and (s.edge(p1, p2) or s.vertex(p1, p2)
   then
23:                                valid ← false
24:                    if valid then
25:                        puzzles[t].add((s, clr1, clr2))
26:    minBin ← ∞
27:    for t = 1 to 4 do
28:        if puzzles[t].size() < minBin then
29:            minBin = puzzles[t].size()
30: if minBin > maxMinBin then
31:    maxMinBin = minBin
32:    bestPattern = placementPattern
33: return bestPattern
```

In this procedure, we iterate through all placement constraint patterns, and for each of them, we go through all the solutions that are valid given that pattern and see how many different color constraint instances it can support. That is, we iterate through the four different color constraint types, and for each of them, we iterate through all the possible same-color combinations and all two-color ones. Then, for each of those combinations, we evaluate whether introducing a constraint defined using them would still keep the solution valid; that is, checking if the solution already satisfies that constraint. For example, a constraint could be that blue and green pieces must share an edge, so we verify in the solution whether this condition is met for all blue and green pieces.

Following the validation of a solution according to a constraint rule, we append a new tuple to the list of solutions that comply with that specific constraint type. This tuple comprises the solution itself and the two colors involved. Subsequently, upon examining all possible adjacency-constraint configurations for a given placement constraint pattern, we proceed to identify the constraint type that results in the fewest number of valid solutions. Upon completing the loop for all placement constraint patterns, we then determine the pattern that yields the highest minimum count and designate it as the "best pattern" for further processing.

The goal of this search is to find the pattern that would lead to having the "richest" curriculum, meaning the curriculum with the largest number of puzzles in each of its different chapters. At this point, we have to define what types of chapters we want our final curriculum to have.

As mentioned earlier, we aim to incorporate several distinct chapters into our puzzle, each introducing a new type of constraint. Thus far, we have established one type of placement constraint, involving the side of the board with bumps,

and the holes on piece triangles adhering to specific configurations outlined by the selected constraint pattern. Additionally, we have four types of color constraints, which can be imposed between pieces of the same color or between those of two different colors. With this array of constraints, we can divide the puzzles into six categories. The first category has no constraints and the subsequent ones each feature one of the aforementioned constraint types. Furthermore, we create a category that encompass combinations of multiple constraint types.

Following this decision, we opted to incorporate a total of 12 chapters into the curriculum. The first six consist of puzzles to be solved on the "clear" 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 not 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.
5. Pieces of color X must not share a corner.
6. Any of the four constraint types above, but between pieces of different colors X and Y.

At this point, we specify that our chosen placement constraint pattern should not only maximize the overall number of solutions but also yield the highest

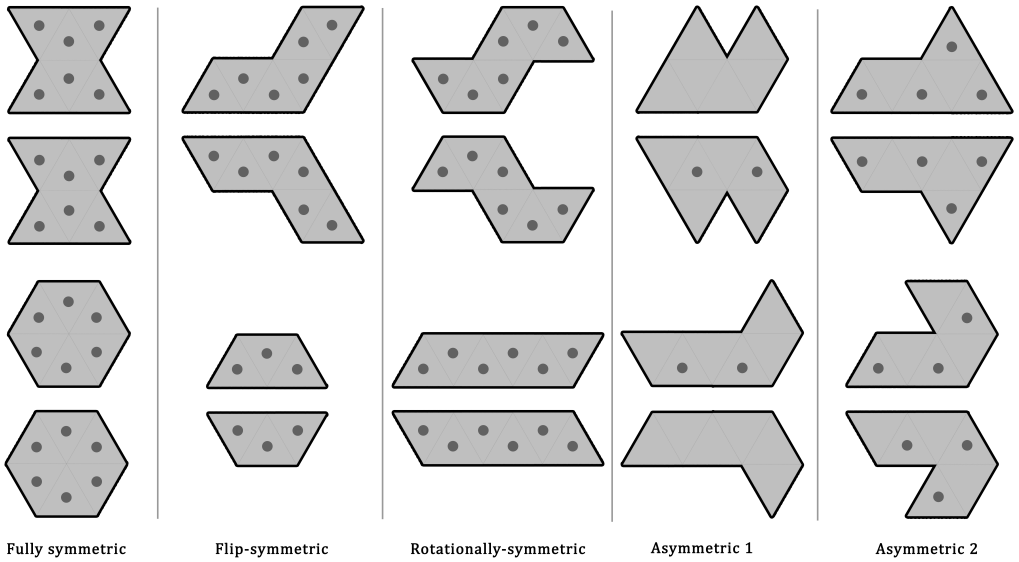


Figure 4.9: Selected placement pattern. Holes are denoted as gray circles. Pieces are grouped together based on what type of symmetry they hold. The last two groups contain pieces that are not symmetrical in any way.

minimum number of puzzles across the twelve chapters we have outlined. This criterion is crucial as it ensures a broader pool of puzzles within each category, thereby enhancing the curriculum's design, as we will discuss in Chapter 5. The final pattern for color assignments is depicted in Figure 4.10, and the associated best placement pattern is shown in Figure 4.9. This pattern has 101 solutions, which is lower than what we arrived at in the previous section; however, it is the best when it comes to the maximum minimum number of puzzles per category, with that being 27 puzzles compared to 21 from the pattern selected originally in the previous section.

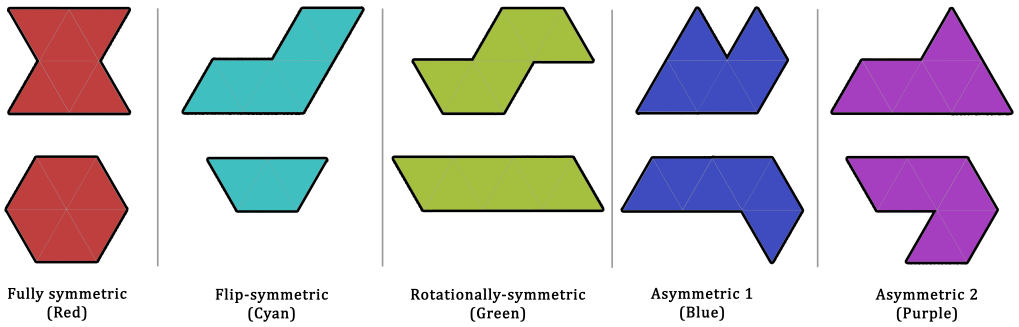


Figure 4.10: Selected adjacency pattern. We assigned colors to pieces according to which symmetry group they belong to.

Chapter 5

Building the Curriculum

In this chapter, we discuss the process of building the curriculum. We begin by explaining how we derive initial states for puzzles from their solutions. Next, we introduce entropy analysis and discuss the concept of inference rules, presenting a selection relevant to our puzzle. We demonstrate how these rules reflect player skills and puzzle comprehension. Then, we introduce the concept of differential entropy, outlining its potential contribution to the development of more effective curricula. Finally, we show how we utilize entropy analysis and differential entropy to structure the finalized curriculum from the set of puzzles derived by the initial state finder from the solutions generated in previous steps of the process.

5.1 Finding Initial States

Before beginning to build the curriculum, a few key components need to be prepared. The first of these is the initial state finder. After completing the constraint analysis portion of the process, we now have a list of solutions for each constraint category. To convert these solutions into puzzles, we need to determine

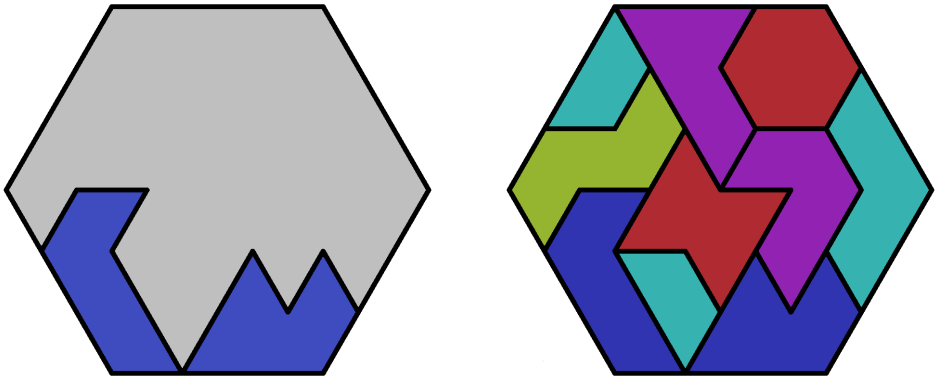


Figure 5.1: A puzzle's initial state and solution.

suitable initial states for players to start from and reach these solutions. An example of an initial state and the corresponding unique solution is shown in Figure 5.1.

We define two stipulations for identifying these initial states. Firstly, each initial state must exclusively lead to a single solution. This is to ensure the inclusion of as many distinct solutions as possible in the curriculum. We note that adherence to this requirement depends on the set of constraints within the puzzle. For instance, if an initial state yields two solutions but one of them fails to comply with the puzzle's constraints, that solution can be disregarded, rendering the initial state valid. Secondly, each initial state must have the minimum number of pieces necessary to uniquely lead to the solution.

Algorithm 7 FindInitialState(solution, otherSolutions)

```
1: for pattern = 0 to numPatterns − 1 do                                  ▷ generator
2:     unique ← true
3:     clone ← solution.clone()
4:     for piece = 0 to numPieces − 1 do
5:         if pattern.bits[piece] == 0 then
6:             clone.remove(p)
7:     for other in otherSolutions do                                     ▷ evaluator
8:         if solution.forbiddenPiece == other.forbiddenPiece then
9:             continue      ▷ only compare to solutions with the same starting pieces
10:        if ¬ValidateConstraints(other, solution.constraints) then
11:            continue      ▷ only compare to solutions that satisfy the defined constraints
12:        otherClone ← other.clone()
13:        for piece = 0 to numPieces do
14:            if pattern.bits[piece] == 0 then
15:                otherClone.remove(p)
16:        if equivalenceTest(clone, otherClone) then
17:            unique ← false
18:            break
19:    if unique then
20:        return pattern
21: return ϕ
```

To construct the initial state finder function, we formalize it as an EPCG procedure, as shown in Algorithm 7. In this scenario, the generator produces all possible sets of initial pieces for the puzzle. Each set can contain any number of pieces, ranging from none to all solution pieces. The idea is that, for each set of initial pieces, we remove all the pieces from the solution that are not included in that initial set and consider that as an initial state for the solution.

To determine the *maxRank* for the generator, we calculate the total number of possible initial piece sets. If we consider the case with the trapezoid pieces, there are 10 total pieces in those sets, including both trapezoids, while there are

9 pieces for the piece set that does not include trapezoids. For each piece in the solution, it can either be in the initial set or not, resulting in two possibilities for each piece. This leads to a total number of configurations equal to 2^{10} or 2^9, depending on whether trapezoids are part of the solution or not.

With the *maxRank* calculated, we proceed to define the *unRank* function. A rank in this case is a binary number, with each bit corresponding to a specific piece, representing whether that piece is in the initial piece set or not. To convert this rank back to a state, we iterate through its bits and remove the pieces with zero values from the solution state. This results in a state where only the pieces in the initial set are present, and they are all in the same locations as they were in the solution.

As for the evaluator, in this case, we use it to determine whether an initial piece set leads to a unique solution or not. To assess this, we iterate through all the other solutions that have the same set of solution pieces (i.e., the same "forbidden" piece) and also adhere to the constraints applied to the solution. For each of these solutions, we compare the locations of the pieces in the initial piece set with those in the solution we are assessing. If the locations match, it indicates that the initial piece set can lead to multiple solutions, rendering it invalid based on our criteria. Otherwise, we label it as the initial state for the solution and proceed to the next one.

One crucial aspect to consider when comparing the initial locations of pieces is that states may be rotationally or flip equivalent to each other. This implies that we must rotate the initial state several times to ensure that none of its rotated and/or flipped versions have their pieces in the same locations as those of the solution we are comparing it to.

To ensure that we encounter the first valid initial piece set with the minimum

number of pieces, we need to iterate through the ranks in the generator in a specific order. This can be achieved by initially iterating through the case with no pieces in the initial set, then progressing to the cases with only one piece in the initial set, followed by those with two pieces, and so on.

The runtime of the initial state finder function depends on the number of solutions that adhere to the same constraints defined for the puzzle under assessment. When executed for the unconstrained case, it takes approximately 0.8 seconds per solution. This performance was achieved by running the function in XCode on an Apple M1 MacBook Air 2020 with 8 GB of RAM.

5.2 Entropy Analysis

For the puzzle game we are building, entropy analysis serves as an appropriate method for evaluating difficulty. This involves measuring information entropy, which quantifies uncertainty in random variable outcomes. As discussed previously in Chapter 2, the MUSE entropy analysis algorithm, which we use in this work, takes as input a state, a successor function, and inference rules, producing an entropy score as output. We utilize the initial states derived using the method discussed in the previous section as input for the entropy analyzer. The successor function for this puzzle generates all possible actions that one can take from the current state, constituting a list of all potential placements for each new piece, excluding the "forbidden" piece, on the board. Inference rules represent expert knowledge, aiding in reducing entropy scores, as discussed in the following section.

5.3 Inference Rules

Players of logic puzzles typically establish a set of guidelines to assist them with puzzle-solving. For instance, in traditional rectangular jigsaw puzzles, players often can tell that two straight edges are corner pieces, and these corners must be neighboring other pieces that posses straight edges. These deductions can be formalized into inference rules.

Inference rules reduce uncertainty in logic puzzles by modeling the uncertainty experienced by skilled players during puzzle-solving. For example, in jigsaw puzzles, experienced players understand that connecting pieces with straight edges to those without them is unnecessary, significantly reducing the number of connections to explore. By decreasing the number of actions that experienced players need to consider, inference rules diminish the uncertainty encountered at each step of puzzle-solving.

In the context of the puzzle we are constructing, we have devised a set of inference rules that we anticipate players will learn as they progress through the game, thereby facilitating puzzle-solving. These rules were primarily formulated based on observations made during puzzle design; however, this list is not exhaustive, and there are other inference rules yet to be discovered and implemented.

5.3.1 Location can only fit a certain piece rule

When solving a puzzle, there are instances where a state features vacant spaces that perfectly match one of the remaining pieces. A vacant or empty space is a group of empty triangles connected by edges. Since we have only one piece of each shape (excluding the trapezoids, which require consideration for each one separately), it becomes evident that the piece corresponding to the empty space

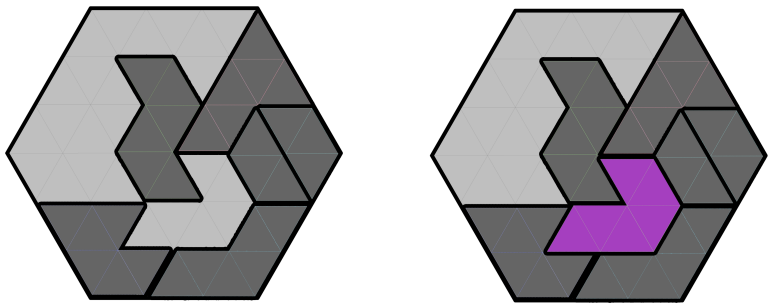


Figure 5.2: An example where a location can only one certain piece. In this case the purple "Hook" piece.

is the correct choice. Figure 5.2 illustrates an example of this scenario.

5.3.2 Piece can only go in one place rule

This scenario bears some resemblance to the previous one, but it occurs when a piece has only one possible position on the board. A distinguishing factor is that in this case, the designated location may not exclusively accommodate that particular piece; it could potentially fit other pieces as well. Figure 5.3 provides a clearer illustration of this scenario.

5.3.3 Size of empty space rule

Here, we observe that each piece is comprised of either three or six triangles, both of which are divisible by three. Consequently, if an empty area within the board contains a number of triangles that is not divisible by three, the current state is invalid, which means it is a state that we know we cannot reach the solution from. This is because the remainder after division by three will consist of triangles that

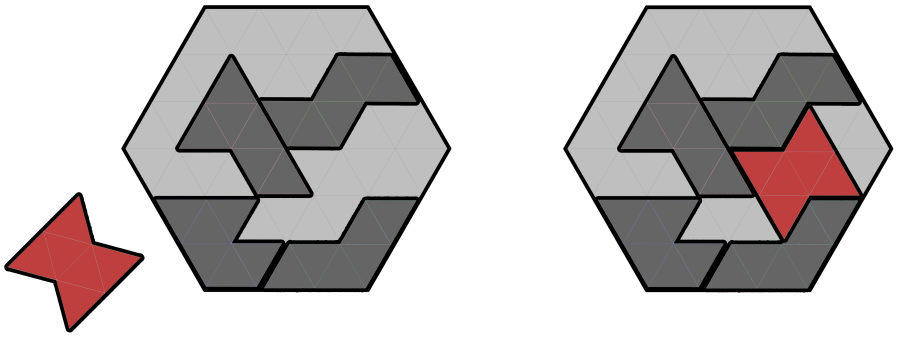


Figure 5.3: An example where a piece can only go into one location on the board. In this case the red "Butterfly" piece.

cannot be filled, given that no piece can accommodate them, as all pieces have a number of triangles divisible by three. Figure 5.4 illustrates a few examples of such instances.

To verify if a state has this property, we first identify the sizes of connected regions consisting of empty triangles. We then compile these triangles into a list to avoid redundant exploration. With the list of empty regions in hand, we determine the size of each and ensure that it is either divisible by 6 (if trapezoids are not available, either because they are forbidden or have both been placed), or that it is divisible by 3 (if at least one trapezoid remains). Finally, we return a boolean value indicating whether the state includes at least one empty area with an invalid size.

5.3.4 Piece that fits the space not available rule

If a space can only accommodate a specific piece, meaning that the space matches the shape and size of the piece, but the piece is either forbidden or already in use,

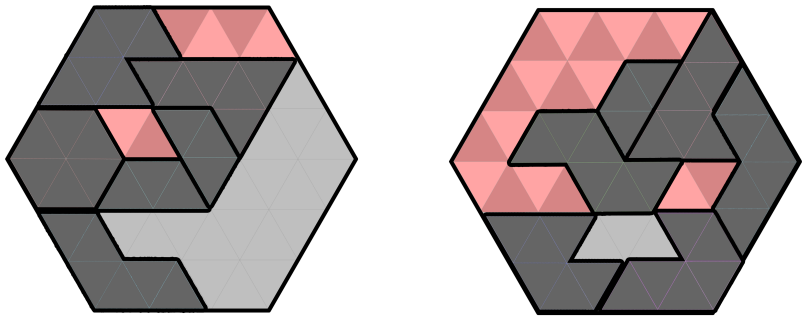


Figure 5.4: Examples where the first inference rule applies. In both examples we have empty regions of sizes non-divisible by three shaded red. In the first board we have two empty spaces of sizes 4 and 2, and in the second one we have one large empty space of size 13 and a smaller one of size 2 that are invalid.

it is reasonable to conclude that the solution cannot be derived from that state. However, this conclusion does not hold if the piece that fits the space is not a trapezoid, but both trapezoids are available. In this scenario, the two trapezoids can be arranged to resemble the piece precisely, as we previously discussed in Chapter 4.1 (all the larger pieces are merely variations of the two trapezoids sharing an edge or more). Figure 5.5 illustrates examples of states where this rule would apply.

To conduct this check, we first compile a list of empty spaces. If any of them is of size 3 and no trapezoids are available, we return a value indicating that the state is invalid. Otherwise, we examine each space of size 6, ensuring that there is a remaining piece with a suitable location to fit into it. If no such piece is found, we return a value indicating that the state is invalid. If all the size 6 spaces pass the check, we return a value indicating that the state is valid.

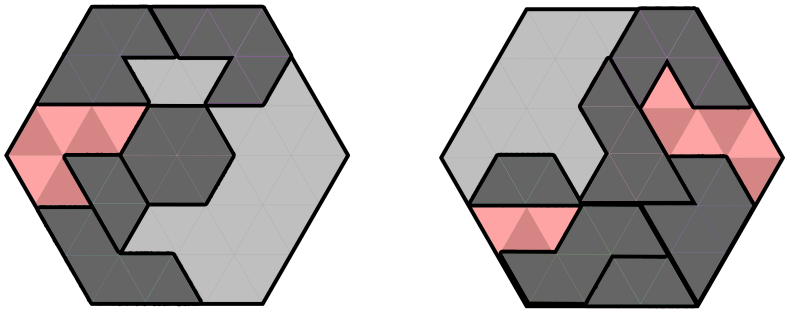


Figure 5.5: Examples where the fourth inference rule applies. In both examples, the piece that corresponds to the shaded empty space is unavailable. In the first instance, the space resembles the "Hook" piece, which has already been used in the solution. Additionally, a trapezoid has been positioned, eliminating the option of using the two trapezoids to form the shape of a "Hook" piece. The second example operates under similar principles, but with the "Snake" and trapezoid pieces. In both cases, we note that these pieces have already been placed.

5.3.5 Pieces are composed of trapezoids rule

As previously mentioned in Chapter 4.1, in this particular puzzle, all larger pieces consisting of 6 triangles are constructed from two trapezoids sharing one or more edges. This characteristic proves useful when encountering a state with an empty space that cannot accommodate any of our pieces. While this rule somewhat overlaps with the previous one, it is generally more efficient to directly dismiss a state with an "incoherent" empty space shape, rather than comparing its shape to all available pieces before discarding it. Figure 5.6 illustrates some examples of this scenario.

To execute this verification, we start by clearing the list of spaces, focusing solely on those of size 6. This rule specifically applies to spaces of this size; it is not relevant for larger or smaller spaces. For each of these size 6 spaces, we

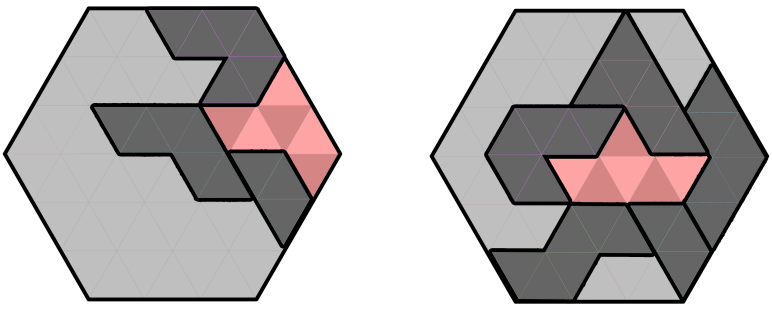


Figure 5.6: Examples where the fifth inference rule applies. In both examples, the size of both the shaded empty areas is 6, which qualifies them under the first inference rule. However, it is evident that the configurations described cannot be constructed using any combination of two trapezoids, indicating that such pieces do not exist in our set to begin with.

examine whether there is a configuration involving two trapezoids that would fit within the space. We conduct this check by attempting to position two trapezoids within the space without overlap, ensuring they completely occupy the area. If we cannot find such a configuration, it indicates that the state is invalid.

Table 5.1 presents statistics for the entropies of puzzles in each constraint 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 entropies for entire sets would be computationally expensive and time-consuming.

5.4 Differential Entropy

Incorporating inference rules aids in refining the entropy score of a puzzle. This is because they eliminate certain actions that are deemed unnecessary due to

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 5.1: Entropy statistics for the chapters.

their potential to lead to invalid future states. A notion arises here: measuring the entropy of a puzzle with and without a specific inference rule. This approach allows us to quantify the reduction in entropy resulting from the addition of the inference rule.

$$DE(s, r) = E(s, []) - E(s, [r]) \tag{5.1}$$

As discussed in Section 5.2, the entropy analyzer takes as input a state, a successor function, and a set of inference rules. As can be seen in Equation 5.1, to calculate differential entropy, we first run the analyzer on the initial state of a puzzle with that inference rule in the input set of rules, record the resulting score ($E(s, []$) in the equation, where s is the initial state and the square empty brackets represent having an empty set of inference rules), and then run the analyzer again without that rule in the set, recording the result ($E(s, [r]$) in the equation, where r inside the square brackets represents the inference rule). Finally, we subtract the entropy score with the inference rule from that without it; this difference is

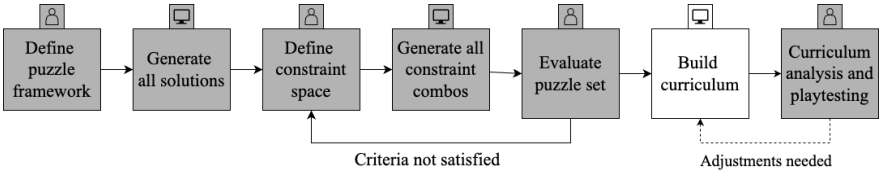


Figure 5.7: Curriculum building step.

the differential entropy ($DE(s, r)$ in the equation).

For each puzzle in our final set, and for each inference rule individually, we conduct this analysis. Initially, we measure the entropy for all these puzzles without any inference rules. Subsequently, we repeat the process with each inference rule enabled individually. The resulting data is organized into a table, with puzzles sorted from highest to lowest difference in entropy between having no inference rules and having a specific rule enabled. A higher difference indicates a more significant reduction in difficulty resulting from learning the corresponding inference rule. We leverage this information when constructing our final curriculum, as it provides insight into how players can utilize specific inference rules when solving puzzles.

5.5 Design of the Curriculum

Now that all the analysis tools have been implemented, we can begin the process of building the curriculum. This involves obtaining all the puzzles, which serve as the initial states for the solutions across all constraint categories. After obtaining these puzzles, along with their associated entropy scores with all inference rules enabled, and after calculating the differential entropy for all puzzles that do not have constraints, our goal is to construct a curriculum using all these pieces. This step is the one before last in the process, as shown in Figure 5.7.

The curriculum contains twelve chapters in total, each introducing a new constraint or combination of constraints. With the first chapter being dedicated to introducing players to the puzzle.

The introductory chapter of the curriculum comprises two main parts. In the first part, we present a series of easy puzzles, all without any constraints. To select these puzzles, we prioritize those with the lowest entropies among all the puzzles that do not have constraints. However, during this process, we observed that even the easiest puzzles in the set may not be easy enough for beginners as suggested by their entropy estimates. This is because, in generating the initial states for these puzzles, as indicated in Section 5.1, we chose initial states that have the minimum number of pieces required to uniquely lead to the solution.

To address this, we devise a function to estimate the entropy difference resulting from adding each remaining piece to the initial state. Consequently, for each puzzle, we have the option to either choose pieces that minimally decrease entropy or those that decrease it maximally. Which is similar to the previous work on Snakebird as discussed in Chapter 2. We opted for the latter and obtained three easy puzzles that still provide some level of challenge, encouraging players to engage in reasoning while solving them.

The next segment of the introductory chapter aims to teach players using the various inference rules we identified. To accomplish this, we regenerate all initial states for the puzzles that do not have constraints defined for them to include more pieces at the start (as overly difficult puzzles might defeat the purpose of teaching beginners). Then, we calculate the entropies for all these puzzles with all inference rules disabled. After that, we repeat the process with each rule enabled individually. This allows us to ascertain a differential entropy score for each puzzle for each inference rule. Next, for each inference rule, we arrange the

puzzles in descending order of the differential entropy for that rule. After that, we select the top three puzzles for each rule and append them to the list of puzzles in the introductory chapter. The assumption here is that by presenting players with puzzles exhibiting high differential entropies, we implicitly encourage them to learn the associated inference rules to navigate these puzzles more adeptly. This assumption necessitates evaluation through experimentation, as explored in Chapter 6.

The remaining chapters of the curriculum each introduce a new constraint or combination thereof. The first six chapters (including the introductory chapter) feature puzzles to be solved on the unconstrained side of the board (the side without bumps). The distinction lies in the type of color constraint imposed in each chapter, progressively going from: no constraint to touching edges, non-touching edges, touching corners, non-touching corners, and finally a chapter encompassing all these constraint types, however defined between different colors. The second half of the curriculum mirrors this sequence, with the stipulation that puzzles from the second half must be solved on the constrained side of the board (the side with bumps) in those chapters.

For each chapter in the curriculum following the introductory one, we first order all puzzles based on entropy. And then, we choose puzzles closest to the following entropy value:

$$entropy_i = entropy_{min} + (entropy_{max} - entropy_{min}) * i/11 \tag{5.2}$$

Where i is an integer that goes from *zero* to 11, denoting the index of the puzzle within the chapter we are building (assuming zero indexing). $entropy_{min}$ gives the entropy of the lowest-entropy puzzle in the set of candidate puzzles for the chapter, and $entropy_{max}$ gives that of the highest-entropy puzzle. Through

using this formula, we attempt to find puzzles that sample the entropy spectrum for the set uniformly.

Now, we have a comprehensive curriculum consisting of twelve chapters, each comprising twelve puzzles, yielding a total of 144 puzzles throughout the curriculum. In the next chapter, we delve into the curriculum, offering examples of puzzles from each chapter and providing statistics about them. Additionally, we share the findings of a playtesting experiment conducted to assess the effectiveness of the curriculum. Note that more can be done here; this process allows us to build a curriculum, but further iteration and refinement are possible and may be necessary.

Chapter 6

Curriculum Evaluation

In this chapter, we present the curriculum we built and discuss its essential parts. We analyze a few puzzles from the first chapter of the curriculum. Then, we explain the process of the playtesting experiment that we conducted and note observations that were made. This constitutes the final part of the process, as shown in Figure 6.1.

6.1 Curriculum Analysis

The curriculum we developed comprises a total of 12 chapters, each containing 12 puzzles. With the exception of the first chapter, each subsequent chapter was crafted to introduce either a new constraint or combination of constraints.

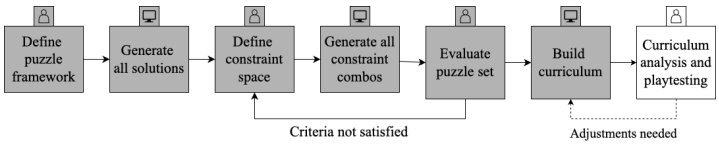


Figure 6.1: Curriculum evaluation step.

Moreover, the puzzles within each chapter are arranged in ascending order of difficulty (based on entropy measures), spanning from the lowest to the highest among all candidate puzzles for that particular chapter.

The first two pages in the curriculum include some basic instructions that introduce the player to the concept of the game. Figure 6.2 depicts those pages. The first one gives a general idea of how the curriculum should be navigated, while the second page introduces the different symbols used in the puzzle to denote the various constraints.

Subsequently, the chapters are presented one by one, starting with the introductory chapter. The puzzles in the first chapter are broken into two main types: easy puzzles that introduce the basic concept of the game, and slightly harder ones that introduce the different inference rules.

Figure 6.3 depicts the first page of the introductory chapter, which includes the first three puzzles (labeled 1 to 3). These puzzles were designed to familiarize the player with the different piece shapes. All of these puzzles have an entropy estimate of *zero*, indicating that the "shortest" path to the solution involves no choice at all. In other words, each piece can only fit into one location, which is its correct position in the solution. We define the shortest solution to a puzzle as the solution in which the player consistently attempts to place the most constrained piece next.

Figure 6.4 illustrates the process of solving the first puzzle in the curriculum. We begin from the initial state shown at the top left, which also indicates that the forbidden piece in this puzzle is the "Mountains" piece. We start by placing the most constrained piece in the set, which in this case is the "Butterfly" piece because it can only occupy one location on the board. Next, we observe that the "Hexagon" piece also has only one available position, so we place it accordingly.

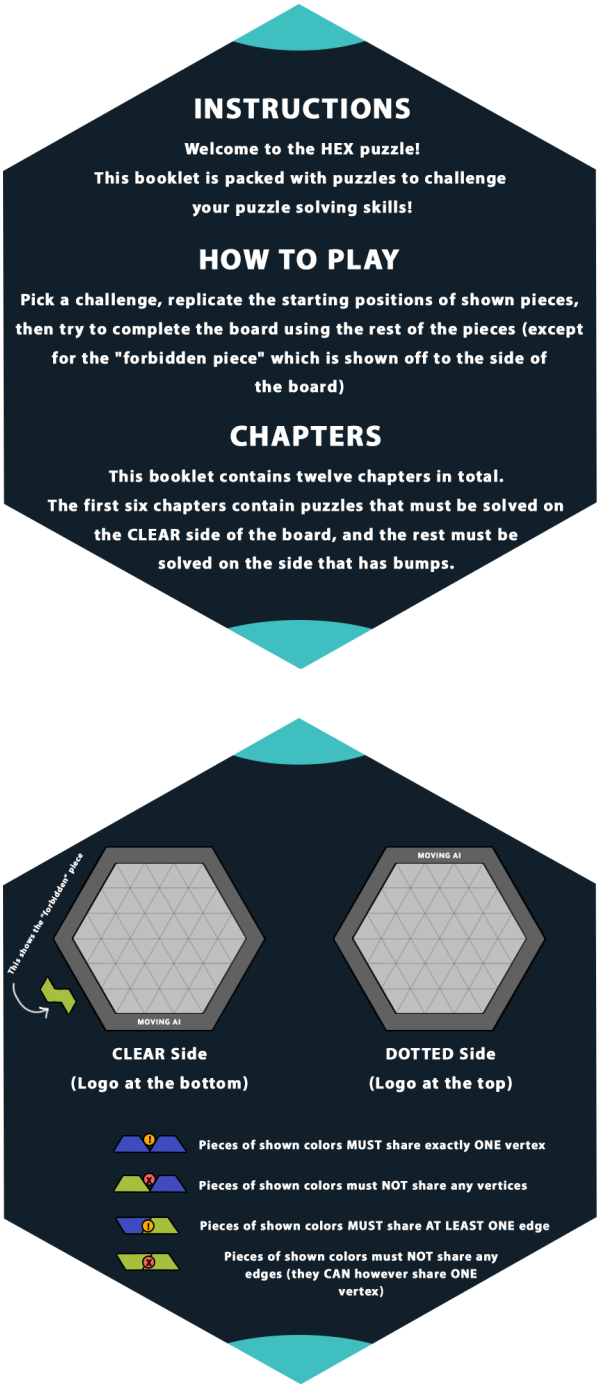


Figure 6.2: Instruction pages at the start of the curriculum.

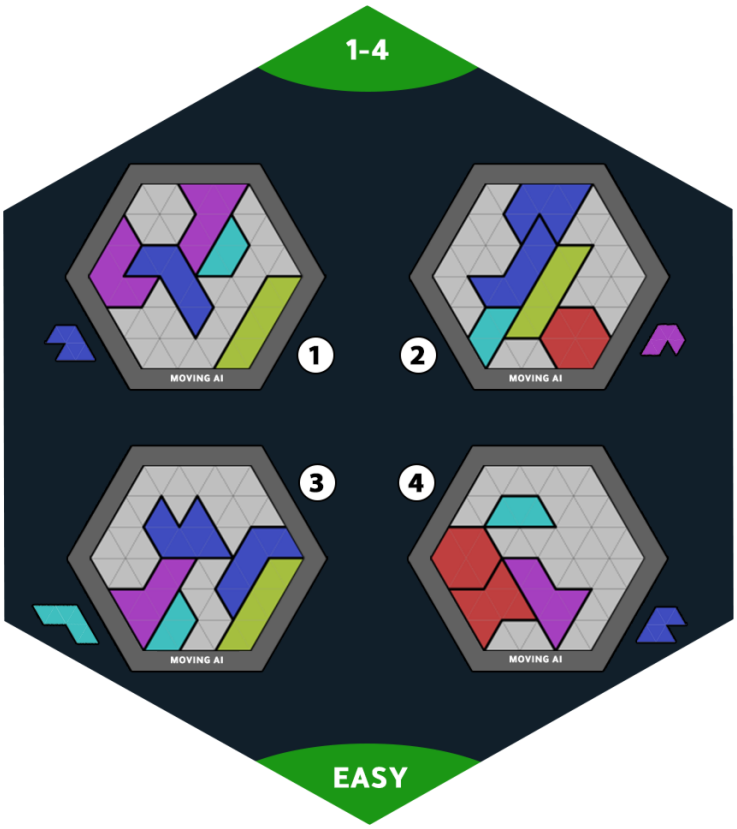


Figure 6.3: First page in the introductory chapter of the curriculum.

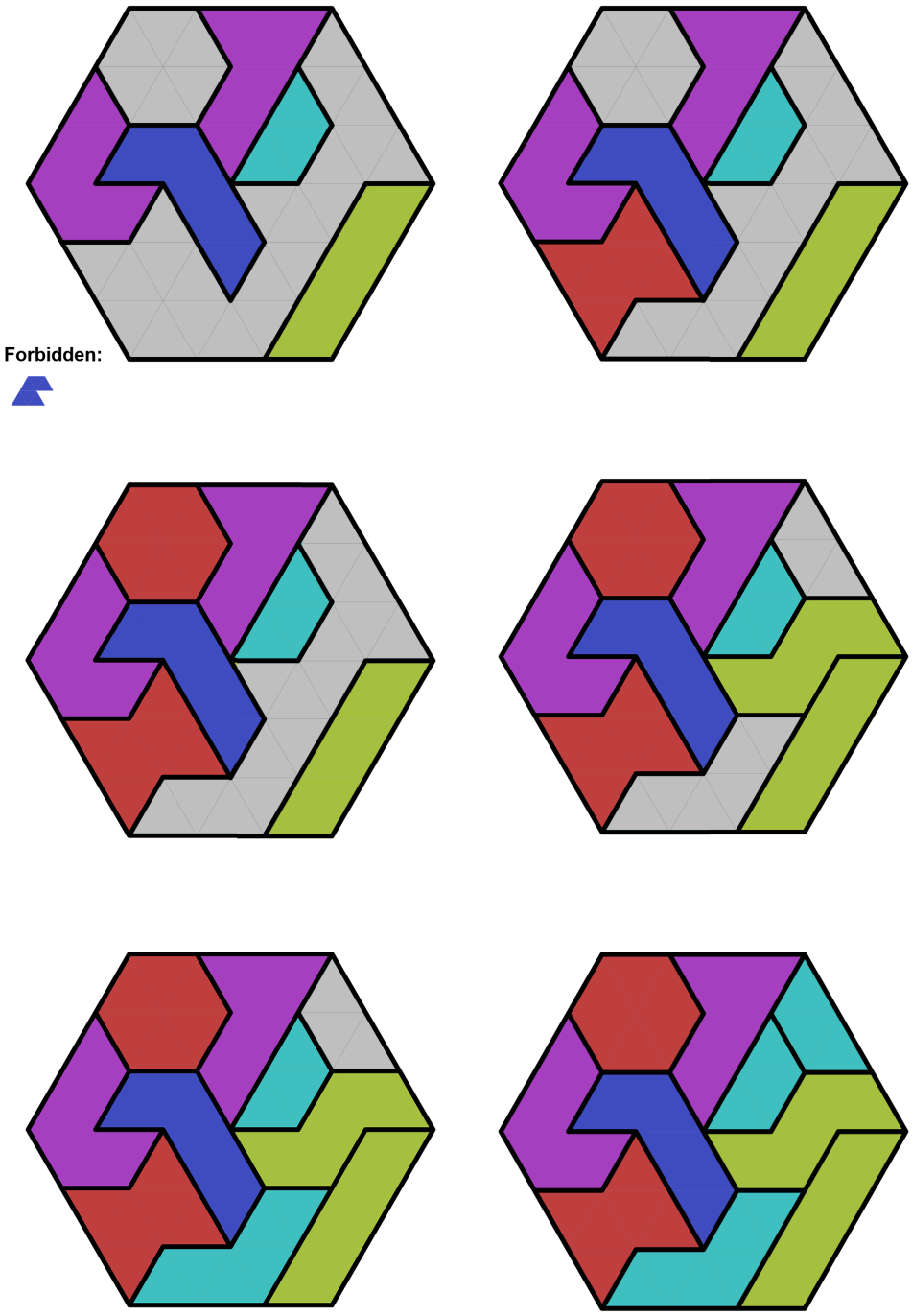


Figure 6.4: Solving the first puzzle.

Afterward, we are left with three pieces: "Snake", "Trapezoid", and "Elbow". Among these, the "Snake" piece is the most constrained as it can only occupy the middle-right location on the board. At this point, we have the option to place either the "Trapezoid" or the Elbow piece. We opt to place the latter because the "Trapezoid" can fit into three possible locations at this state. Finally, we place the "Trapezoid" piece and achieve the goal state.

The next three puzzles attempt to teach the player the "Size of empty space" inference rule. This rule stipulates that each empty space on the board must be of a size divisible by 6 if no trapezoids remain, or by 3 otherwise. Puzzles aiming to teach this rule are those with the highest differential entropy, meaning the entropy difference between having that rule enabled in the entropy estimator and having it disabled.

The remaining six puzzles in the first chapter attempt to teach two more inference rules. The first three are for the "Piece that fits the space not available" rule, and the other three are for the "Pieces are composed of trapezoids" rule. For each, we again use the concept of differential entropy to determine which puzzles the player will benefit from the most by knowing those rules. The two figures below show the selected puzzles for this section. The remaining chapters also include twelve puzzles each, with puzzles ordered incrementally in terms of entropy. The complete curriculum can be found in the appendices section at the end of this document.

6.2 Playtesting Experiment

Upon completing the curriculum, we conducted an informal playtesting experiment involving members of our Moving AI lab, comprising six participants in

total.

Each participant received a 3D-printed physical board featuring both the clear side and the side with bumps. Additionally, they were provided with a complete set of colored pieces and copies of the curriculum.

Initially, we asked participants to attempt solving the last puzzle in the introduction set without first completing the preceding eleven puzzles. After approximately 15 minutes, none of the participants were able to solve the puzzle. Subsequently, we instructed them to work through the first few puzzles in the chapter before revisiting the challenging puzzle. Following this guidance, two participants successfully solved the previously challenging puzzle. For the remaining participants, we advised them to complete all the puzzles leading up to the twelfth one before attempting it again. At this stage, four out of the five participants were able to solve the previously difficult puzzle.

Following these initial tasks, participants were given the freedom to navigate through the curriculum as they desired. Some chose to tackle the more challenging puzzles, while others preferred to progress through the curriculum incrementally. We observed that when a participant encountered difficulty with a puzzle, they tended to either backtrack to the beginning of the chapter containing that puzzle and work their way back to it, or persistently attempted to solve it until success was achieved.

An interesting observation made by one of the participants (who is also the supervisor of this thesis study) during the challenge to complete the final puzzle in the curriculum (puzzle number 144) was that the puzzle was actually easier than its entropy measure initially suggested. The puzzle is depicted in Figure 6.5. What the participant noticed was that three of the pieces were highly constrained because the color constraint required those pieces to be touching. Specifically, the

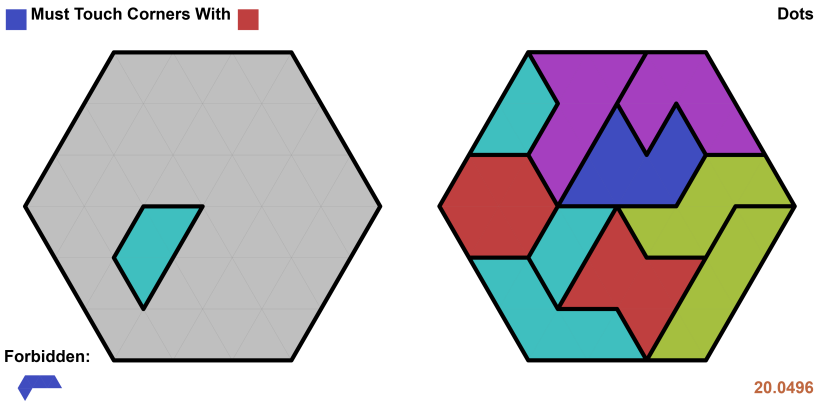


Figure 6.5: The final puzzle in the curriculum, with the initial state on the left and the solution on the right. In the top-left corner, a constraint is defined for this puzzle, indicating that blue and red pieces must touch corners. In the bottom-left corner, we see that the forbidden piece is the "Wrench". The top-right corner indicates that this puzzle is to be solved on the side with dots (later renamed to bumps). Finally, in the bottom-right corner, we can see the entropy estimate for the puzzle.

"Hexagon" and the "Butterfly" pieces were both required by the constraint rule to be touching corners with the "Mountains" piece. The participant explained that once this constraint was satisfied, the rest of the puzzle became straightforward to solve. Interestingly, this highlighted a limitation in our entropy analysis algorithm; specifically, it revealed that we overlooked an inference rule which would have resulted in a much lower entropy estimate for that puzzle. This could be resolved with further iteration on the curriculum, as indicated at the end of the previous chapter.

The unimplemented rule appears to pertain to color constraints involving the placement of three different pieces that must be connected. In our implementation, we only considered two pieces at a time when evaluating constraint-related inference rules. The second part of the figure above illustrates an example of this, which occurred in the final puzzle.

Additionally, one participant suggested that including a written description of the inference rule on the instructions page would aid players in learning these rules, rather than having them attempt to learn them blindly. While we agree that this approach facilitates rule learning, we also acknowledge that it may be somewhat limited in the general case, as some games have inference rules that are not as straightforward to describe and may be challenging to communicate directly through instructions without causing confusion among players.

Overall, the general consensus among participants was that having a curriculum indeed helped them navigate the puzzle more effectively and understand it more deeply. However, we recognize that drawing firm conclusions on curriculum evaluation without conducting a complete and formal user study is challenging.

Chapter 7

Conclusions and Future Work

In this study, we built a complete puzzle end-to-end using Exhaustive Procedural Content Generation (EPCG). We began by determining the puzzle type we intended to work with and then proceeded to develop a corresponding definition. Subsequently, we generated all solutions for the puzzle. Following this, we devised definitions for constraints and generated all the constraint combinations, repeating this process until we reached a set of constraints that are both intuitive aesthetically and applicable to a sufficiently large number of puzzles. After this, we built an entropy analyzer that provides an estimate of the difficulty for each puzzle and leveraged it to devise the final curriculum for the puzzle.

The main contributions of our work can be summarized as the following:

1. We demonstrated the feasibility of utilizing EPCG for driving end-to-end game design. We crafted a complete puzzle accompanied by a range of diverse constraint types. Nonetheless, this process still required input from the developers to ensure adherence to specific aesthetic requirements, making the puzzle intuitive enough for human players. We note that while it

may be possible to execute this process in its entirety without any human input, it is likely that it would be challenging to derive compelling games solely through automated means.

2. We built an entropy analyzer for the puzzle together with a set of inference rules.

3. We introduced the concept of differential entropy, which posits that puzzles exhibiting the greatest difference in entropy when a specific inference rule is enabled versus when it is disabled are most beneficial for learning that rule. However, this premise remains speculative until validated through formal experimental evaluation.

4. We designed a curriculum for the game aimed at providing a sense of progression, utilizing entropy analysis and differential entropy. Specifically, we applied the concept of differential entropy in selecting puzzles for the introductory chapter of the curriculum. The objective was to leverage these puzzles to teach players the various inference rules, each corresponding to distinct skills that aid in navigating the puzzle more effectively.

In terms of future work, there a few directions to consider: improving the entropy analyzer, running more concrete evaluations, generating other types of constraints, or generating initial states in different ways.

Improving the entropy analyzer. This can be achieved by introducing additional inference rules, or through improving the existing ones. An example of a potential improvement to an inference rule is to extend the color constraint inference rule to handle multiple touching pieces as noted in the playtesting experiment discussed at the end of Chapter 6.

Running more evaluations As noted in Chapter 6, the playtest experiment we conducted was informal and may not be the best method for evaluating the approaches employed. There are several ways to conduct a more formal evaluation. One approach is to evaluate the entropy analyzer by performing a depth-first iterative deepening search on the puzzles to attempt to solve them, comparing the number of expansions to the puzzle's entropy measure. Another method involves creating a policy learning agent and assessing the order in which it learns the puzzles to determine if it aligns with the entropy measures' recommendations. A third approach is to conduct a formal user study, which could involve distributing puzzles to random participants or specifically to puzzle design experts and collecting their evaluations on the constrats' interestingness or the puzzles' levels of difficulty to assess how well the entropy measure aligns with their perceptions.

Generating other types of constraints This could be done either through introducing new constraint types beyond placement and adjacency constraints or by exploring different ways to define the established constraints. As illustrated in Chapter 4.1, our criteria for selecting the best constraint pattern for each constraint type were partially subjective, as they depended on some aesthetic choices that we made. This suggests that alternative approaches might lead to different sets of constraints for this puzzle. An example of a new constraint could be one that prevents two pieces of the same color from being placed in the same row or column, though this would require defining what constitutes a row or column in this puzzle. Additionally, one could explore new ways to combine existing constraints, such as puzzles featuring two adjacency constraint rules instead of one.

Generating initial states in different ways The initial state function we developed aims to find, for each puzzle, the initial state that uniquely leads to the desired solution with the minimum number of initial pieces. However, this is not strictly necessary, and initial states can be tailored to other objectives. For instance, one could attempt to find puzzles that begin with no pieces, given a set of constraints. Alternatively, one could introduce a chapter containing puzzles that all start with the same piece but in different locations and with different sets of constraints.

Bibliography

Alfred V. Aho. Computation and computational thinking. *Ubiquity*, January 2011.

Liapis Antonios, N Yannakakis Georgios, Alexopoulos Constantine, and Lopes Phil. Can computers foster human users' creativity? theory and praxis of mixed-initiative co-creativity. *Digital Culture & Education*, 2016.

Second-Lieutenant Augustine Blanchonnet. Development of a curriculum for a tangram type puzzle. 2021.

Blizzard. Diablo. In *Proceedings of the 1996 Annual Conference*, 1996.

Jonathan Blow and Marc ten Bosch. Designing to reveal the nature of the universe. Talk presented at Indiecade, October 2011.

Jose M Campillo-Robles, Ibon Alonso, Ane Gondra, and Nerea Gondra. Calculation and measurement of center of mass: An all-in-one activity using tangram puzzles. 2022.

Eugene You Chen Chen, Adam White, and Nathan R Sturtevant. Entropy as a measure of puzzle difficulty. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2023.

William Byron Forbush. *Manual of play*, 1914.

Mark Hendrikx, Sebastiaan Meijer, Joeri Van Der Velden, and Alexandru Iosup. Procedural content generation for games: A survey. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, 2013.

Christoffer Holmgård, Antonios Liapis, and Georgios N Yannakakis. Measuring game complexity using information theory. *IEEE Transactions on Games*, 2017.

Arthur H Land and Alvin G Doig. An automatic method of solving discrete programming problems. *Econometrica: Journal of the Econometric Society*, 1960.

Jongpil Lee, Haoran Huang, Xiangyu Wang, and Marcelo Milrad. Developing a serious game for introductory programming: A case study. In *Proceedings of the 2019 3rd International Conference on Educational Innovation and Philosophical Inquiries (ICEIPI 2019)*, 2019.

Rebecca Morelle. Text reveals more ancient secrets. *BBC News*, April 2007. URL https://www.bbc.com/news/science-environment-44223824. Archived from the original on 19 February 2009. Retrieved 2009-03-31.

Mossmouth. Spelunky. In *Proceedings of the 2008 Annual Conference*, 2008.

Wendy Myrvold and Frank Ruskey. Ranking and unranking permutations in linear time. *Information Processing Letters*, 2001.

Thorbjørn Nielsen and Rachel Smith. Difficulty, flow, and enjoyment: An empirical study of computer games. *Journal of Entertainment Computing*, 2018.

Planck Collaboration, P. A. R. Ade, N. Aghanim, M. Arnaud, M. Ashdown, J. Aumont, C. Baccigalupi, A. J. Banday, R. B. Barreiro, J. G. Bartlett, N. Bartolo, E. Battaner, R. Battye, K. Benabed, A. Benoît, A. Benoit-Lévy, J. P.

Bernard, M. Bersanelli, P. Bielewicz, J. J. Bock, A. Bonaldi, L. Bonavera, J. R. Bond, J. Borrill, F. R. Bouchet, F. Boulanger, M. Bucher, C. Burigana, R. C. Butler, E. Calabrese, J. F. Cardoso, A. Catalano, A. Challinor, A. Chamballu, R. R. Chary, H. C. Chiang, J. Chluba, P. R. Christensen, S. Church, D. L. Clements, S. Colombi, L. P. L. Colombo, C. Combet, A. Coulais, B. P. Crill, A. Curto, F. Cuttaia, L. Danese, R. D. Davies, R. J. Davis, P. de Bernardis, A. de Rosa, G. de Zotti, J. Delabrouille, F. X. Désert, E. Di Valentino, C. Dickinson, J. M. Diego, K. Dolag, H. Dole, S. Donzelli, O. Doré, M. Douspis, A. Ducout, J. Dunkley, X. Dupac, G. Efstathiou, F. Elsner, T. A. Enßlin, H. K. Eriksen, M. Farhang, J. Fergusson, F. Finelli, O. Forni, M. Frailis, A. A. Fraisse, E. Franceschi, A. Frejsel, S. Galeotta, S. Galli, K. Ganga, C. Gauthier, M. Gerbino, T. Ghosh, M. Giard, Y. Giraud-Héraud, E. Giusarma, E. Gjerløw, J. González-Nuevo, K. M. Górski, S. Gratton, A. Gregorio, A. Gruppuso, J. E. Gudmundsson, J. Hamann, F. K. Hansen, D. Hanson, D. L. Harrison, G. Helou, S. Henrot-Versillé, C. Hernández-Monteagudo, D. Herranz, S. R. Hildebrandt, E. Hivon, M. Hobson, W. A. Holmes, A. Hornstrup, W. Hovest, Z. Huang, K. M. Huffenberger, G. Hurier, A. H. Jaffe, T. R. Jaffe, W. C. Jones, M. Juvela, E. Keihänen, R. Keskitalo, T. S. Kisner, R. Kneissl, J. Knoche, L. Knox, M. Kunz, H. Kurki-Suonio, G. Lagache, A. Lähteenmäki, J. M. Lamarre, A. Lasenby, M. Lattanzi, C. R. Lawrence, J. P. Leahy, R. Leonardi, J. Lesgourgues, F. Levrier, A. Lewis, M. Liguori, P. B. Lilje, M. Linden-Vørnle, M. López-Caniego, P. M. Lubin, J. F. Macías-Pérez, G. Maggio, D. Maino, N. Mandolesi, A. Mangilli, A. Marchini, M. Maris, P. G. Martin, M. Martinelli, E. Martínez-González, S. Masi, S. Matarrese, P. McGehee, P. R. Meinhold, A. Melchiorri, J. B. Melin, L. Mendes, A. Mennella, M. Migliaccio, M. Millea, S. Mitra, M. A. Miville-Deschênes, A. Moneti, L. Montier, G. Morgante, D. Mortlock, A. Moss, D. Munshi, J. A. Murphy, P. Naselsky, F. Nati, P. Natoli, C. B. Netterfield,

H. U. Nørgaard-Nielsen, F. Noviello, D. Novikov, I. Novikov, C. A. Oxborrow, F. Paci, L. Pagano, F. Pajot, R. Paladini, D. Paoletti, B. Partridge, F. Pasian, G. Patanchon, T. J. Pearson, O. Perdereau, L. Perotto, F. Perrotta, V. Pettorino, F. Piacentini, M. Piat, E. Pierpaoli, D. Pietrobon, S. Plaszczynski, E. Pointecouteau, G. Polenta, L. Popa, G. W. Pratt, G. Prézeau, S. Prunet, J. L. Puget, J. P. Rachen, W. T. Reach, R. Rebolo, M. Reinecke, M. Remazeilles, C. Renault, A. Renzi, I. Ristorcelli, G. Rocha, C. Rosset, M. Rossetti, G. Roudier, B. Rouillé d'Orfeuil, M. Rowan-Robinson, J. A. Rubiño-Martín, B. Rusholme, N. Said, V. Salvatelli, L. Salvati, M. Sandri, D. Santos, M. Savelainen, G. Savini, D. Scott, M. D. Seiffert, P. Serra, E. P. S. Shellard, L. D. Spencer, M. Spinelli, V. Stolyarov, R. Stompor, R. Sudiwala, R. Sunyaev, D. Sutton, A. S. Suur-Uski, J. F. Sygnet, J. A. Tauber, L. Terenzi, L. Toffolatti, M. Tomasi, M. Tristram, T. Trombetti, M. Tucci, J. Tuovinen, M. Türler, G. Umana, L. Valenziano, J. Valiviita, F. Van Tent, P. Vielva, F. Villa, L. A. Wade, B. D. Wandelt, I. K. Wehus, M. White, S. D. M. White, A. Wilkinson, D. Yvon, A. Zacchei, and A. Zonca. Planck 2015 results. XIII. Cosmological parameters. , September 2016.

Noor Shaker, Julian Togelius, Mark J Nelson, et al. Search-based procedural content generation: A taxonomy and survey. *Transactions on Computational Intelligence and AI in Games*, 2016.

CE Shannon. A mathematical theory of communication. the bell systems technical journal, 27: July 379–423, 1948.

Jerry Slocum. The tao of tangram, 2001.

Jerry Slocum. The tangram book: The story of the chinese puzzle, 2003.

Adam M. Smith, Ahmed Khalifa, Ahmed E. Metwally, Julian R. Clarke,

Michael Cerny Green, and Julian Togelius. Procedural generation of dungeons for minecraft. In *Proceedings of the 14th International Conference on the Foundations of Digital Games (FDG)*, 2019.

Gillian Smith. Challenges in procedural content generation. *ACM Computers in Entertainment (CIE)*, 2010.

Nathan Sturtevant. An argument for large-scale breadth-first search for game design and content generation via a case study of fling! *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, Jun. 2021. URL https://ojs.aaai.org/index.php/AIIDE/article/view/12594.

Nathan Sturtevant and Matheus Ota. Exhaustive and semi-exhaustive procedural content generation. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2018.

Nathan Sturtevant, Nicolas Decroocq, Aaron Tripodi, Carolyn Yang, and Matthew Guzdial. A demonstration of anhinga: A mixed-initiative epcg tool for snakebird. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, Oct. 2020a. URL https://ojs.aaai.org/index.php/AIIDE/article/view/7451.

Nathan Sturtevant, Nicolas Decroocq, Aaron Tripodi, Carolyn Yang, and Matthew Guzdial. A demonstration of anhinga: A mixed-initiative epcg tool for snakebird. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2020b.

Nathan R. Sturtevant. Exploring epcg in the witness. In *Knowledge Extraction from Games (AAAI workshop)*, 2019. URL http://www.cs.ualberta.ca/~nathanst/papers/puzzles.pdf.

Adam Summerville, Sam Snodgrass, Matthew Guzdial, Christoffer Holmgård, Amy K. Hoover, Aaron Isaksen, Andy Nealen, and Julian Togelius. Procedural content generation via machine learning (PCGML). *CoRR*, 2017. URL http://arxiv.org/abs/1702.00539.

Julian Togelius, Renzo De Nardi, Simon M Lucas, Mike Preuss, Nicola Beume, and Sebastian Risi. Search-based procedural content generation. In *Proceedings of the 2011 IEEE Conference on Computational Intelligence and Games*, 2011.

Michael Toy, Glenn Wichman, Ken Wichman, and Jon Wichman. Rogue. In *Proceedings of the 1980 Annual Conference*, 1980.

Fernando Valls, Mireia Usart, Jordi Mateu, and Josep M Duart. Evaluation of a multi-criteria assessment tool for educational games in elementary schools. In *2017 9th International Conference on Virtual Worlds and Games for Serious Applications (VS-Games)*, 2017.

Georgios N Yannakakis and Julian Togelius. Experience-driven procedural content generation. *IEEE Transactions on Affective Computing*, 2015.

Georgios N Yannakakis and Julian Togelius. *Artificial intelligence and games*, 2018.

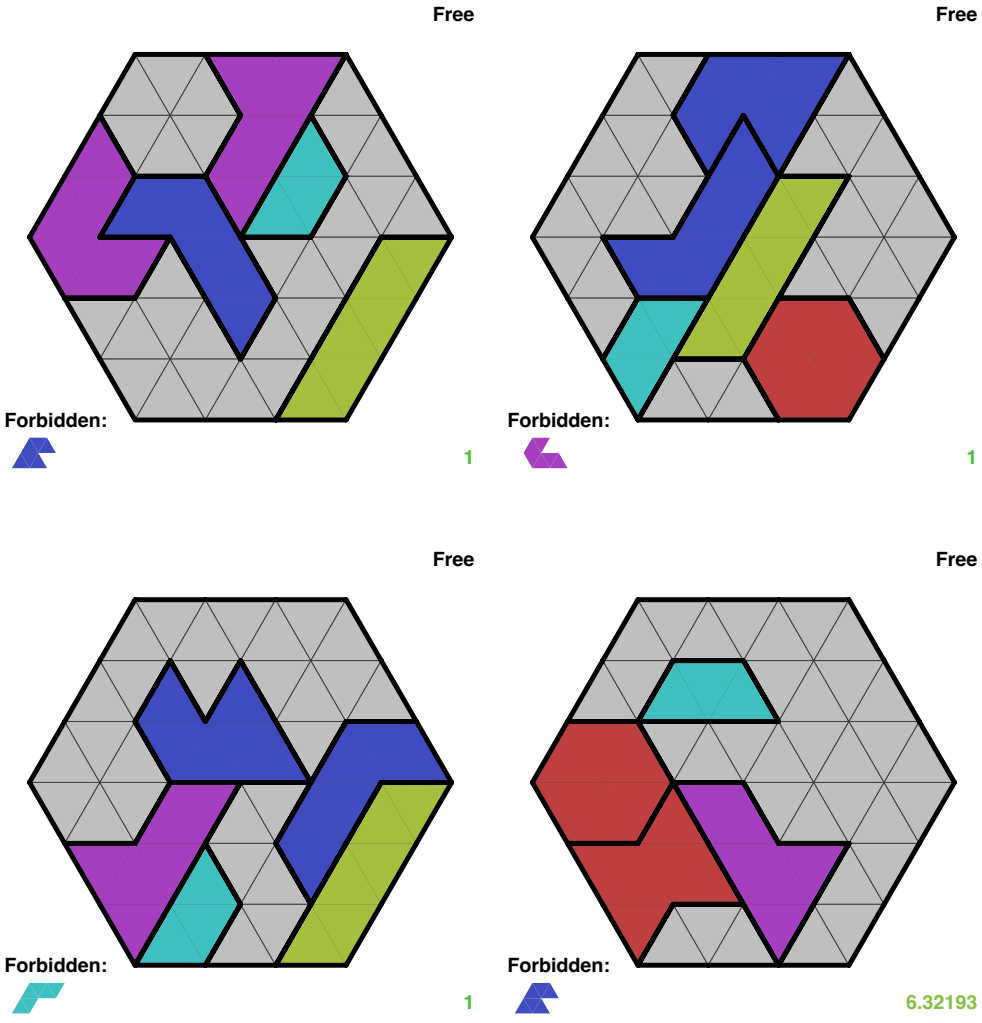
Zhongyu Zhang, Bei Lian, Chongguang Chen, Kang Li, Yanzhang Li, and Xiaolong Liu. Enhancing game-based learning outcomes using an adaptive difficulty adjustment mechanism. *Educational Technology Research and Development*, 2019.

Appendix

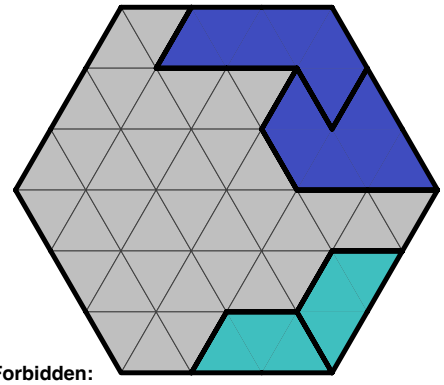
7.1 Curriculum

This is the complete curriculum built for this puzzle, containing 12 chapters in total. The top left corner shows the adjacency constraint. The text on the top right indicates which side of the board the puzzle must be solved on; "Free" denotes the side without bumps, while "Dots" refers to the side with bumps. The bottom left corner indicates which piece must not be used in the solution. Finally, in the bottom right corner, the entropy estimate for the puzzle can be found (with all inference rules enabled).

7.1.1 Chapter 1



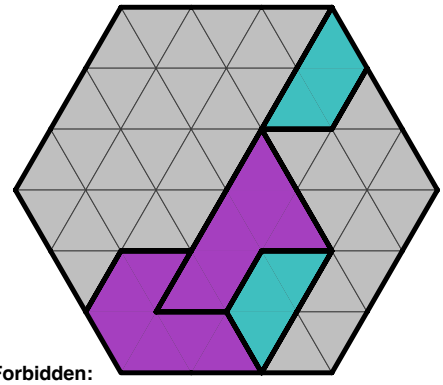
Free



Forbidden:

10.1098

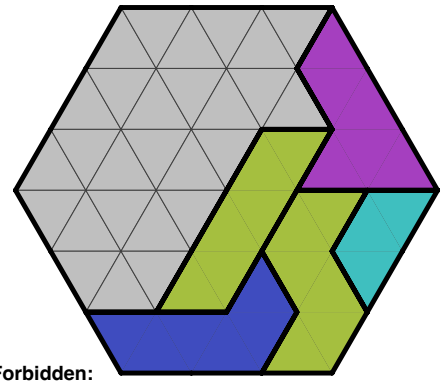
Free



Forbidden:

9.2854

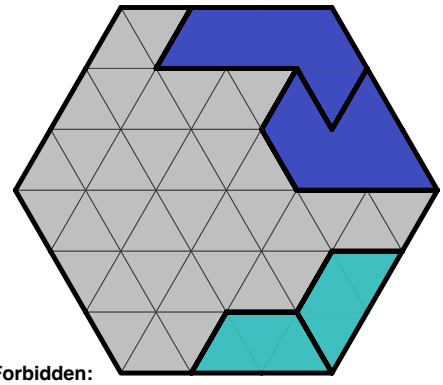
Free



Forbidden:

7.80735

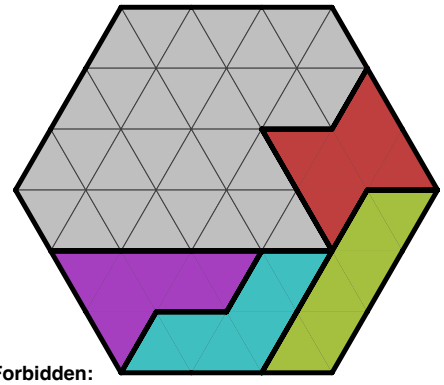
Free



Forbidden:

14.4919

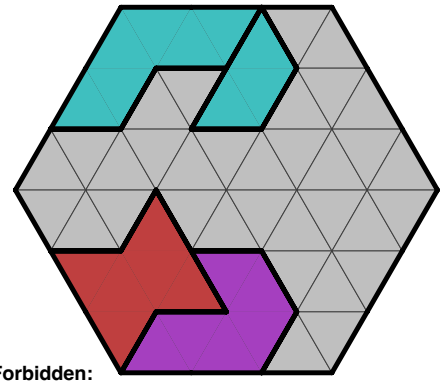
Free



Forbidden:

9.83289

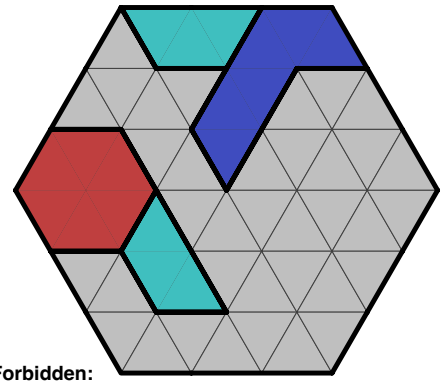
Free



Forbidden:

4.58496

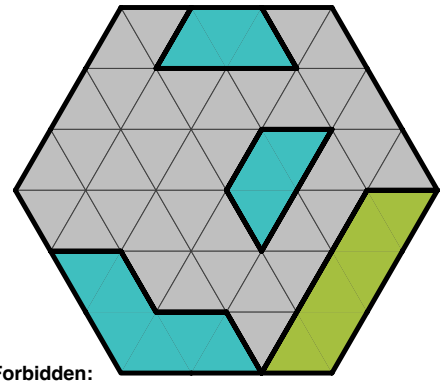
Free



Forbidden:

3

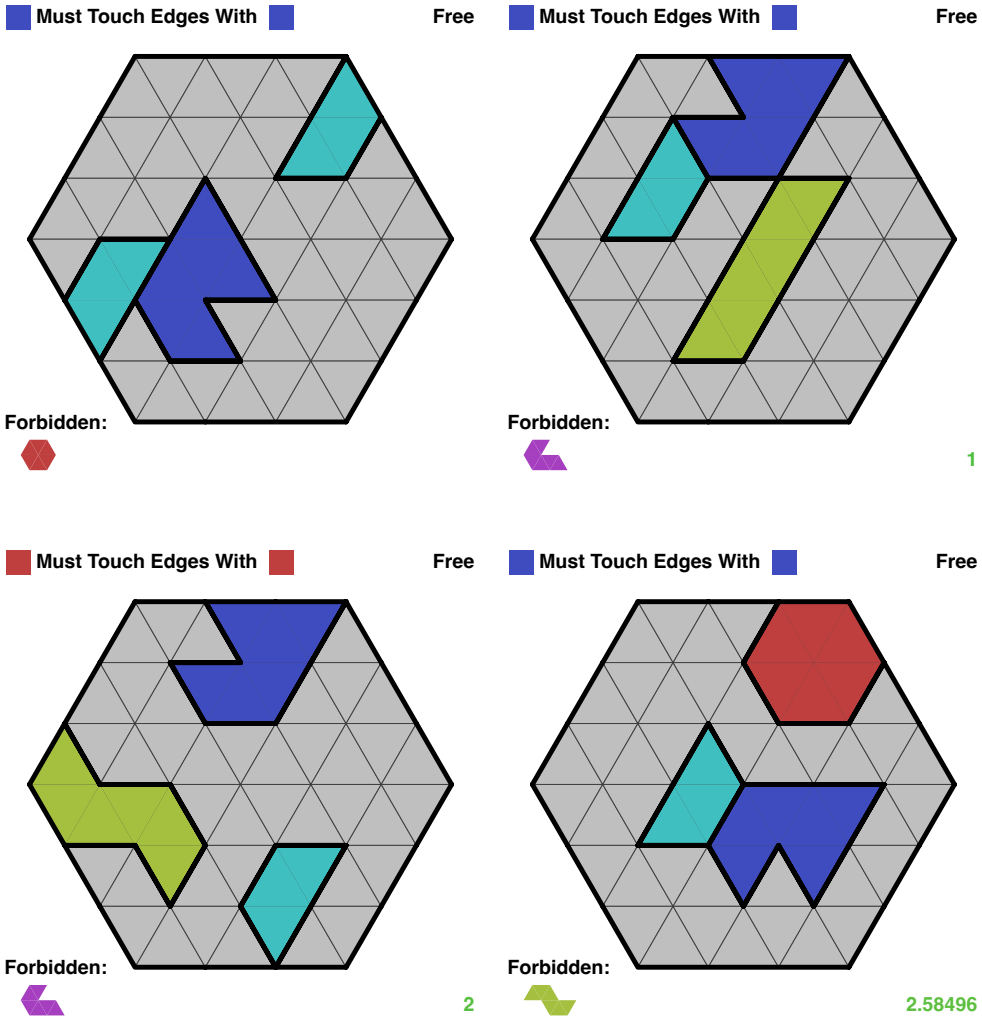
Free



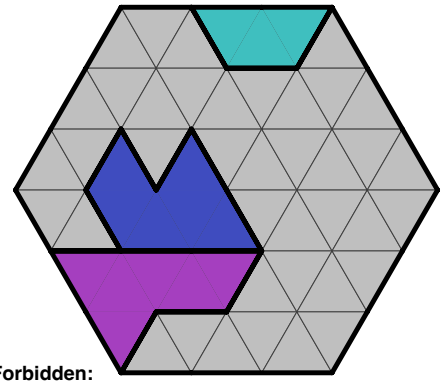
Forbidden:

3.32193

7.1.2 Chapter 2



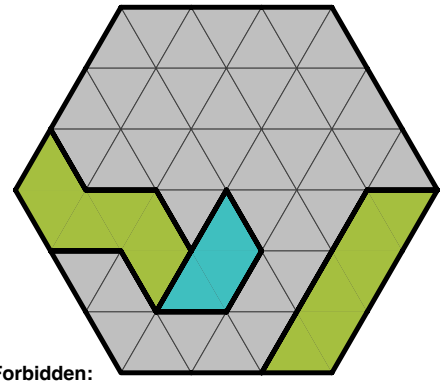
Must Touch Edges With Free



Forbidden:

3.58496

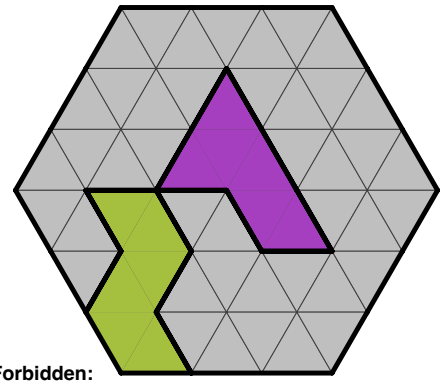
Must Touch Edges With Free



Forbidden:

5.16993

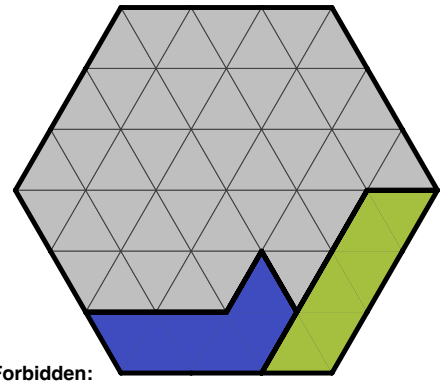
Must Touch Edges With Free



Forbidden:

7

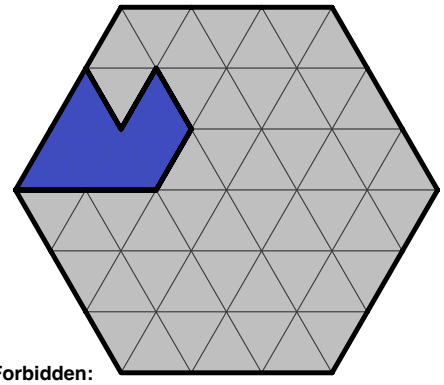
Must Touch Edges With Free



Forbidden:

8.78136

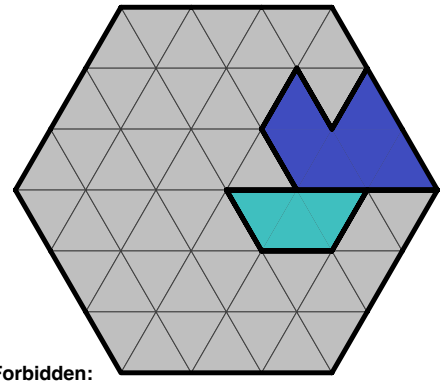
Must Touch Edges With Free



Forbidden:

10.5078

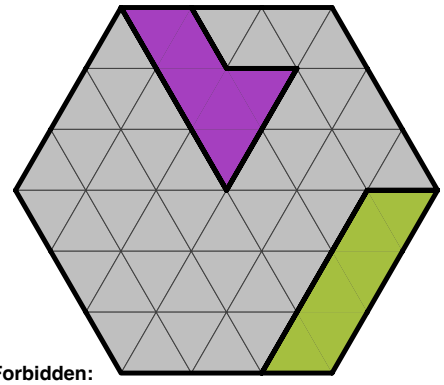
Must Touch Edges With Free



Forbidden:

12.2668

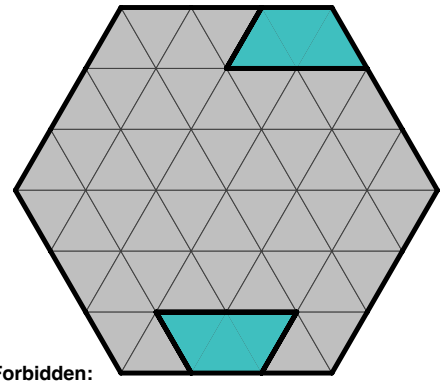
Must Touch Edges With Free



Forbidden:

14.1472

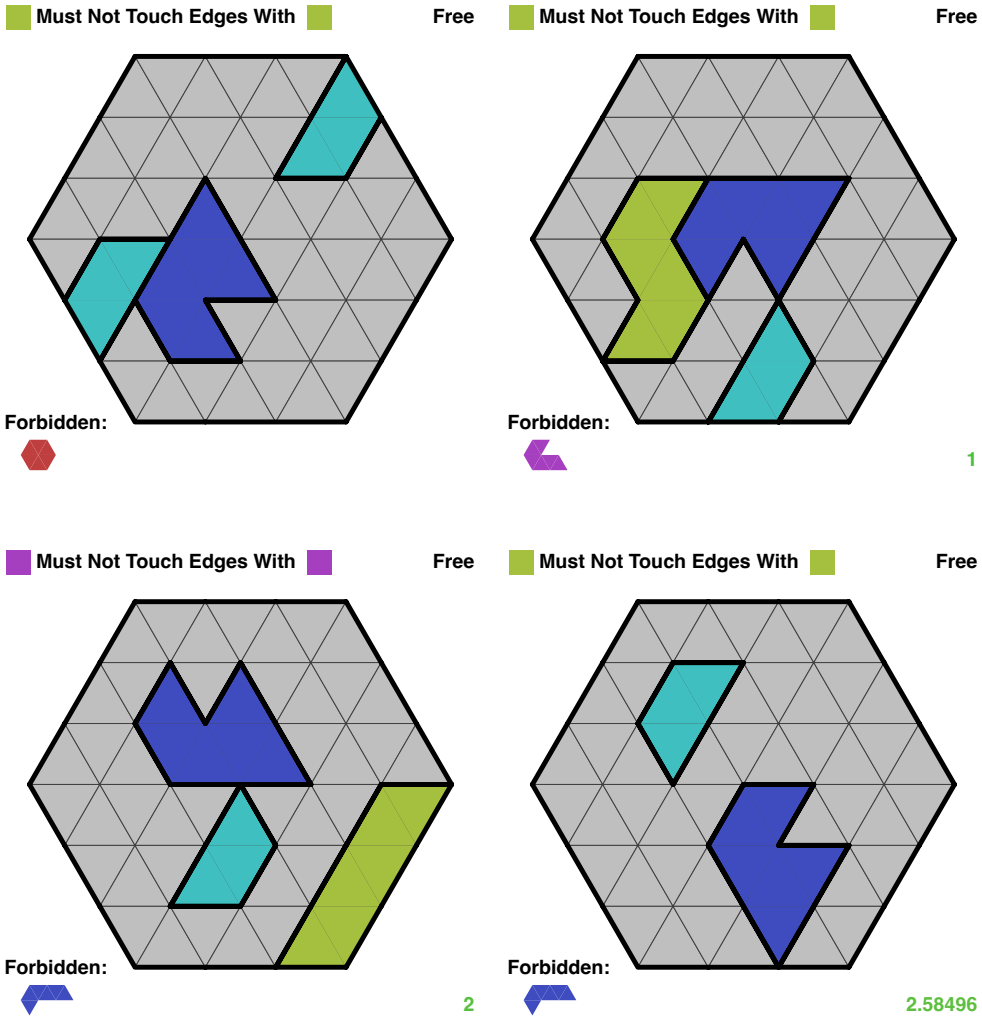
Must Touch Edges With Free



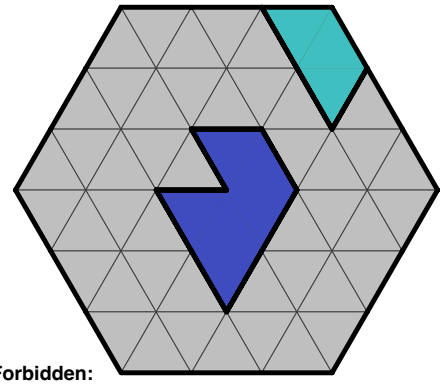
Forbidden:

16.2703

7.1.3 Chapter 3



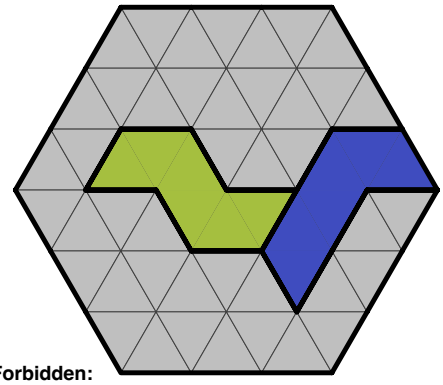
Must Not Touch Edges With Free



Forbidden:

3.80735

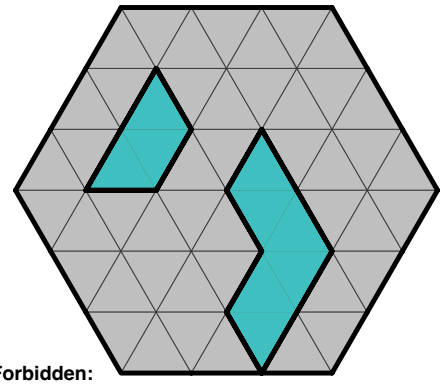
Must Not Touch Edges With Free



Forbidden:

5.58496

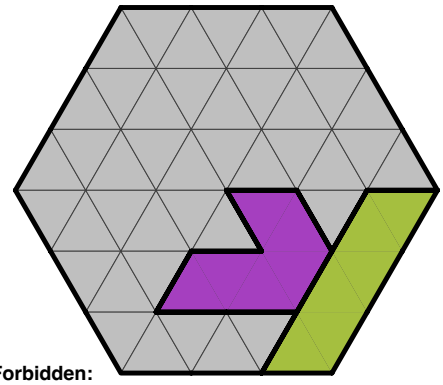
Must Not Touch Edges With Free



Forbidden:

7.22882

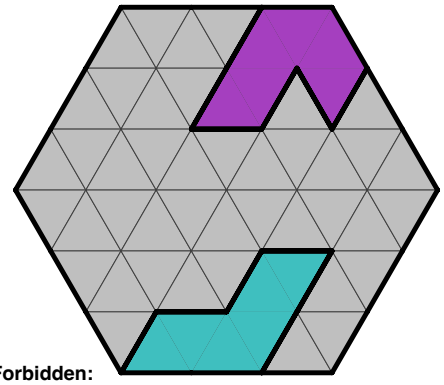
Must Not Touch Edges With Free



Forbidden:

9

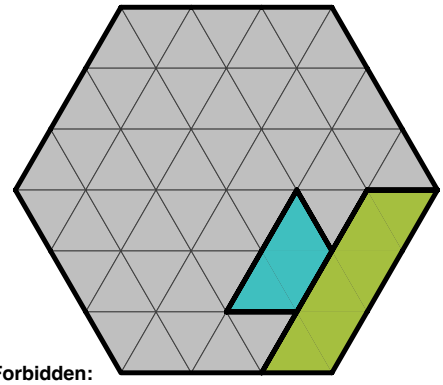
Must Not Touch Edges With Free



Forbidden:

10.7549

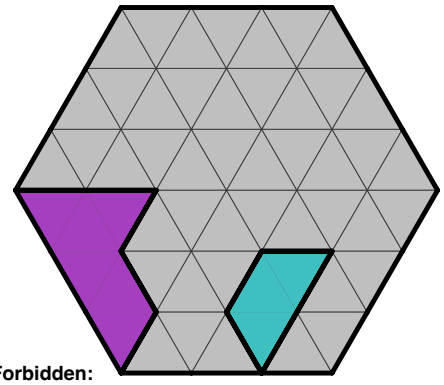
Must Not Touch Edges With Free



Forbidden:

12.5622

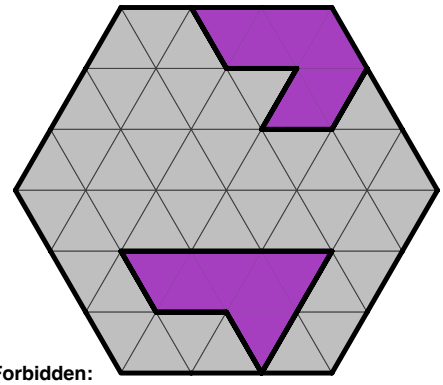
Must Not Touch Edges With Free



Forbidden:

14.5179

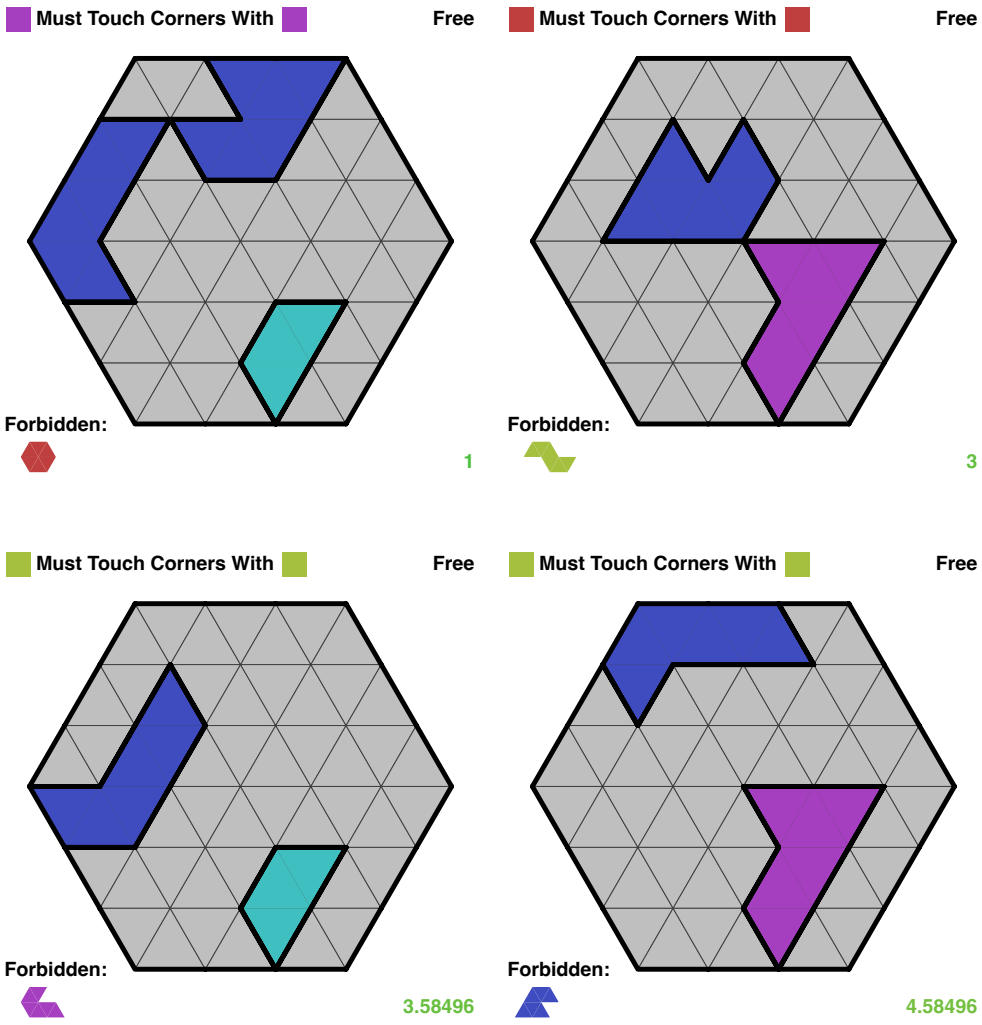
Must Not Touch Edges With Free



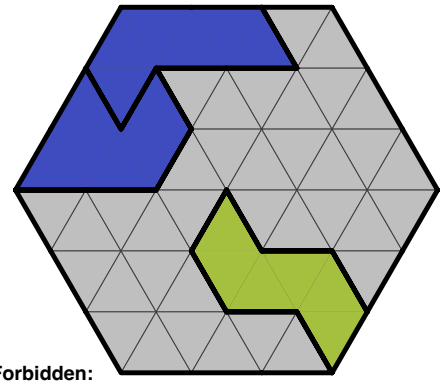
Forbidden:

17.6915

7.1.4 Chapter 4



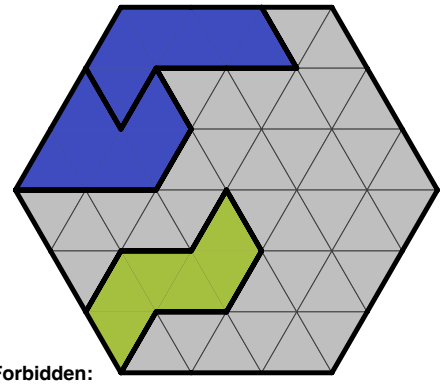
Must Touch Corners With Free



Forbidden:

5

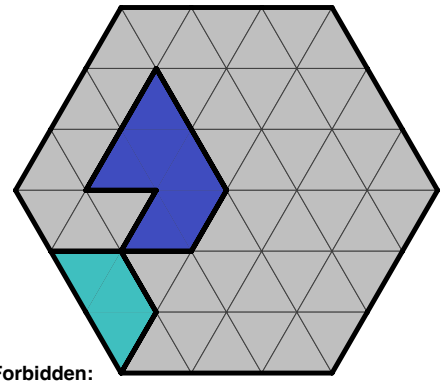
Must Touch Corners With Free



Forbidden:

5.58496

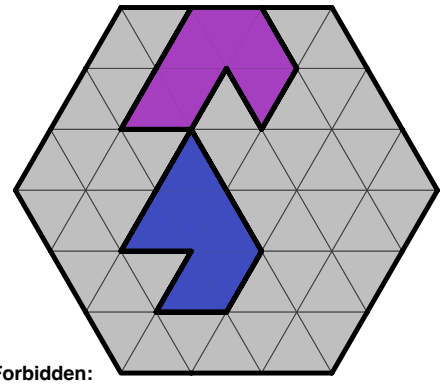
Must Touch Corners With Free



Forbidden:

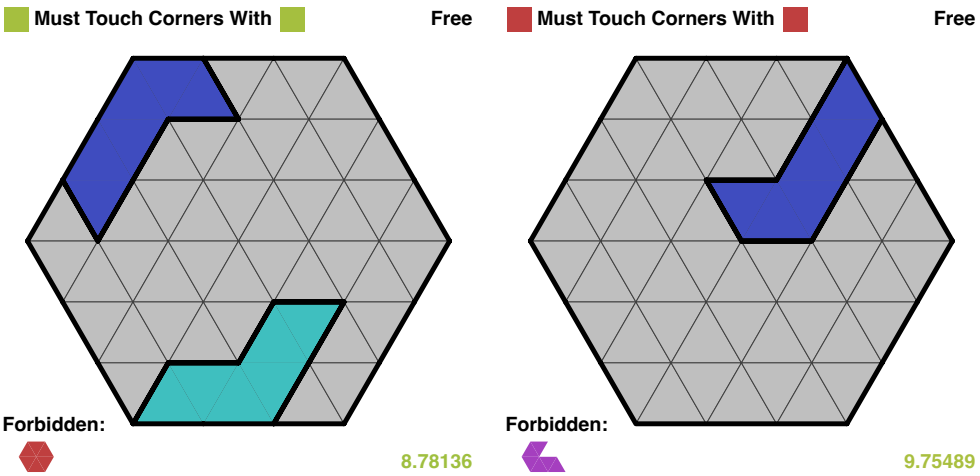
6

Must Touch Corners With Free



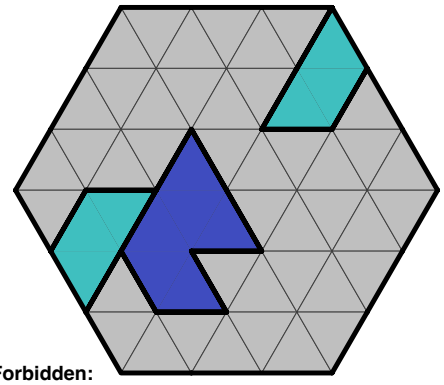
Forbidden:

7.32193


Must Touch Corners With
Free
Forbidden:
8.78136
Must Touch Corners With
Free
Forbidden:
9.75489
Must Touch Corners With
Free
Forbidden:
11.4919
Must Touch Corners With
Free
Forbidden:
13.8517

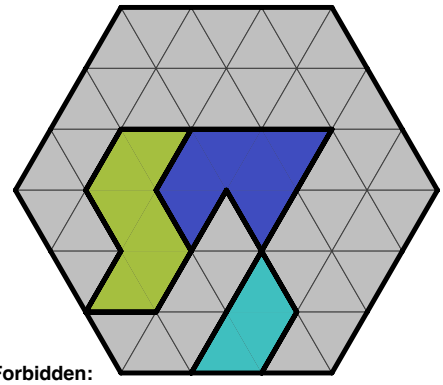
7.1.5 Chapter 5

Must Not Touch Corners With Free



Forbidden:

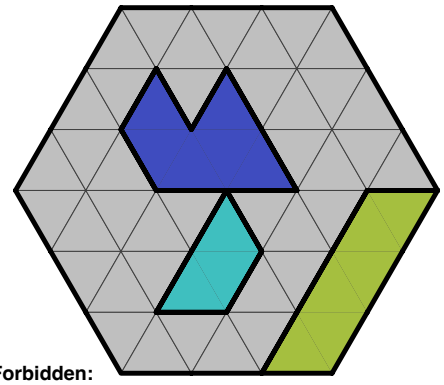
Must Not Touch Corners With Free



Forbidden:

1

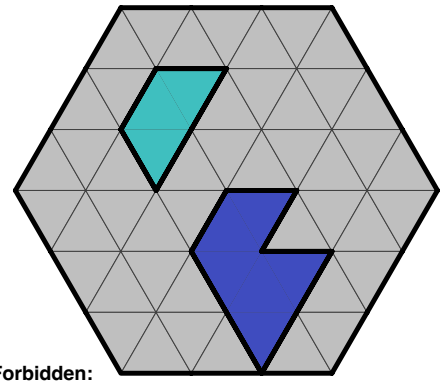
Must Not Touch Corners With Free



Forbidden:

2

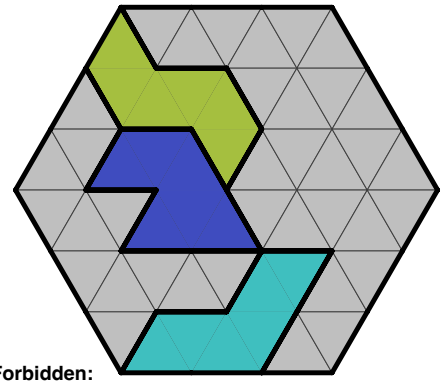
Must Not Touch Corners With Free



Forbidden:

2.58496

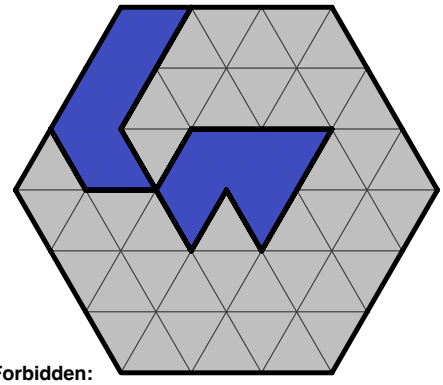
Must Not Touch Corners With Free



Forbidden:

4

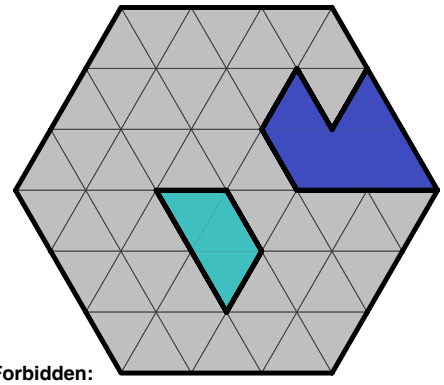
Must Not Touch Corners With Free



Forbidden:

5.58496

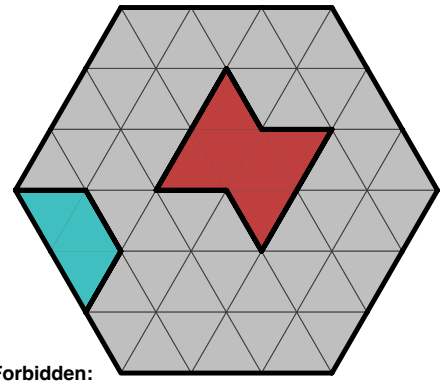
Must Not Touch Corners With Free



Forbidden:

7.33985

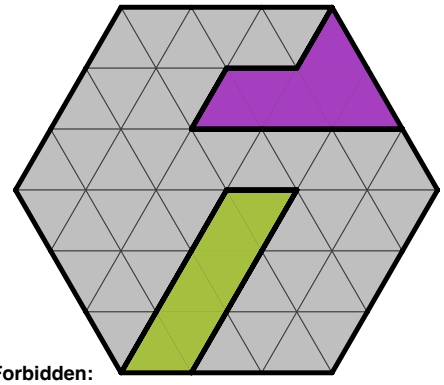
Must Not Touch Corners With Free



Forbidden:

10.9069

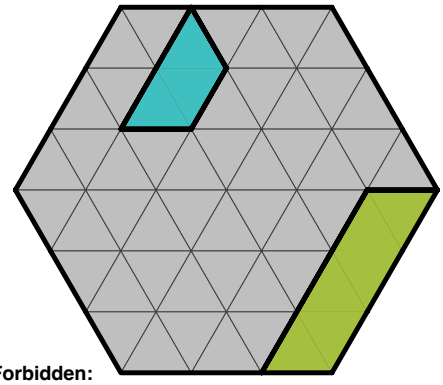
Must Not Touch Corners With Free



Forbidden:

11.9773

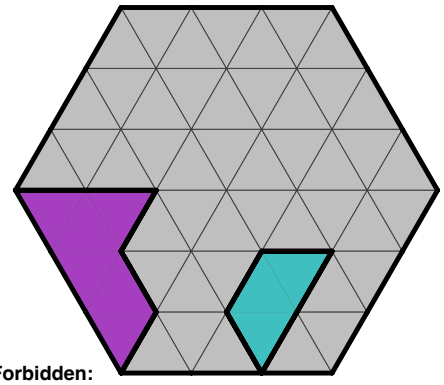
Must Not Touch Corners With Free



Forbidden:

13

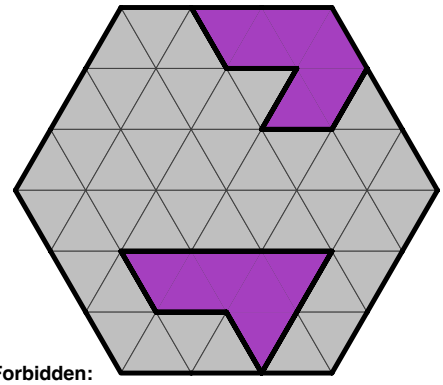
Must Not Touch Corners With Free



Forbidden:

14.5179

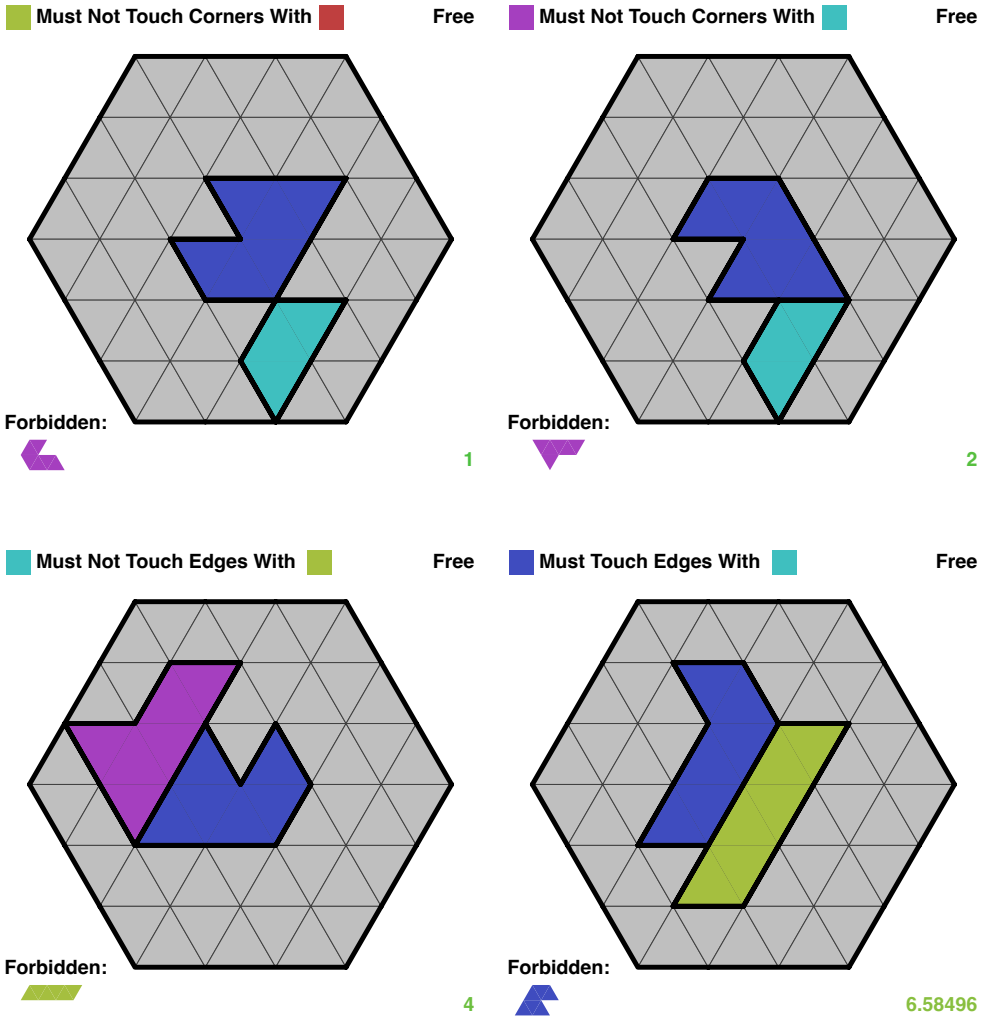
Must Not Touch Corners With Free



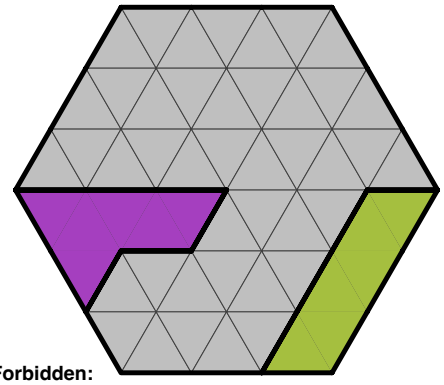
Forbidden:

17.6915

7.1.6 Chapter 6



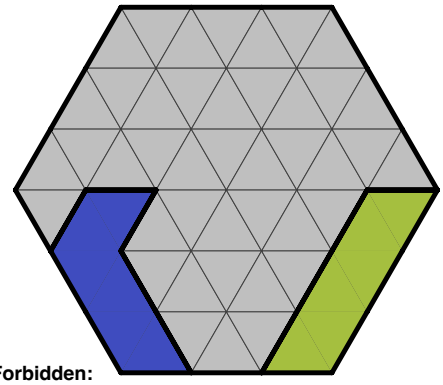
Must Touch Edges With Free



Forbidden:

8.80735

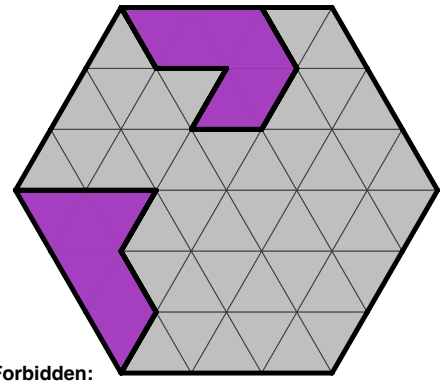
Must Not Touch Corners With Free



Forbidden:

9.32193

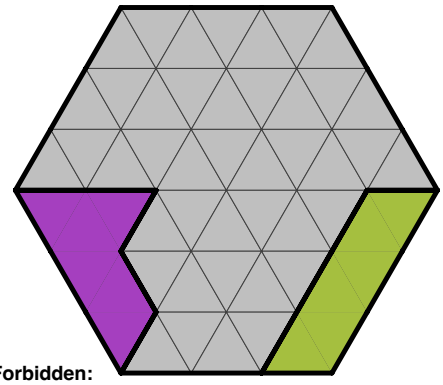
Must Touch Edges With Free



Forbidden:

10.1293

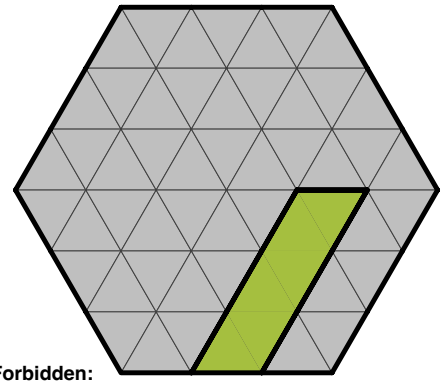
Must Not Touch Corners With Free



Forbidden:

10.6294

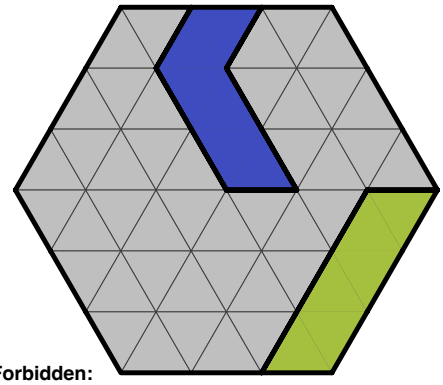
Must Not Touch Corners With Free



Forbidden:

11.585

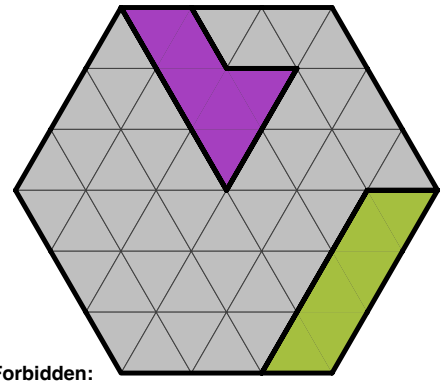
Must Not Touch Corners With Free



Forbidden:

12.9069

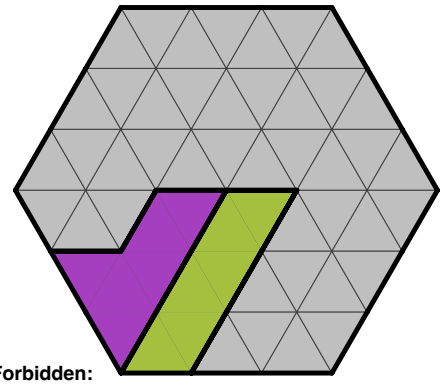
Must Not Touch Edges With Free



Forbidden:

14.1472

Must Not Touch Corners With Free

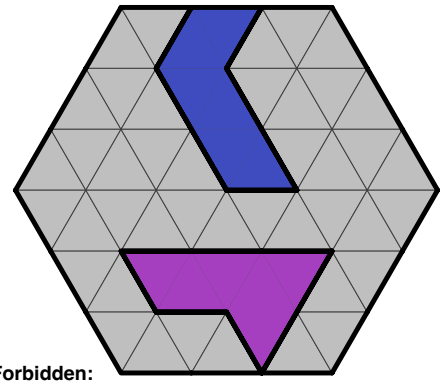


Forbidden:

16.1472

7.1.7 Chapter 7

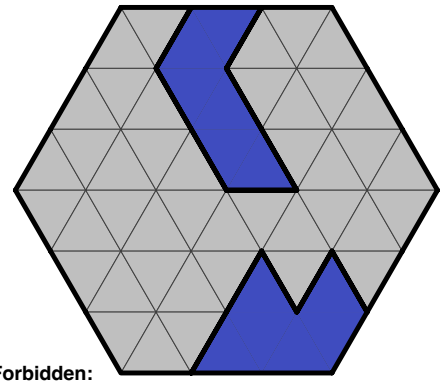
Dots



Forbidden:

3

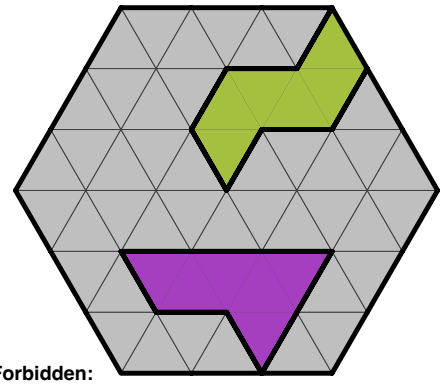
Dots



Forbidden:

4.58496

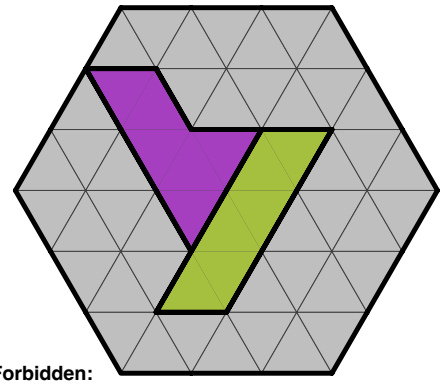
Dots



Forbidden:

5.58496

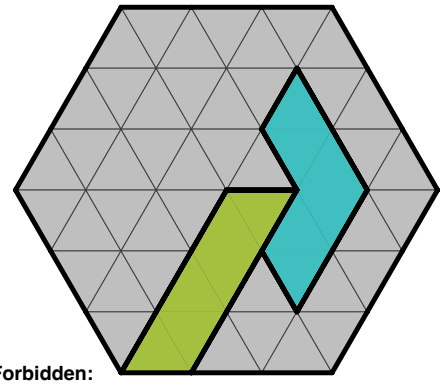
Dots



Forbidden:

6

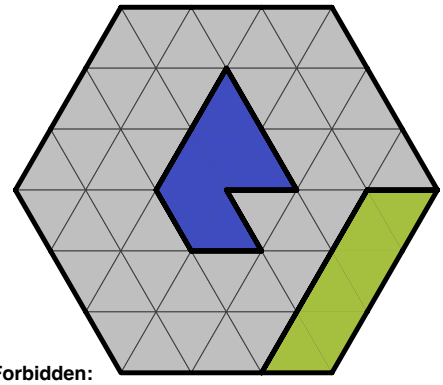
Dots



Forbidden:

7

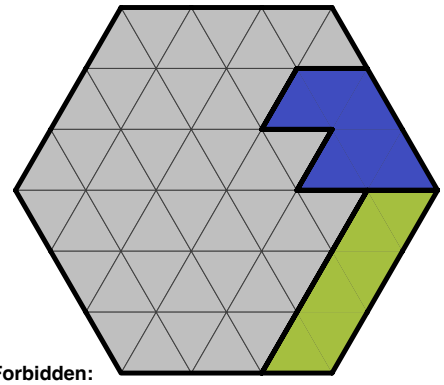
Dots



Forbidden:

8.39232

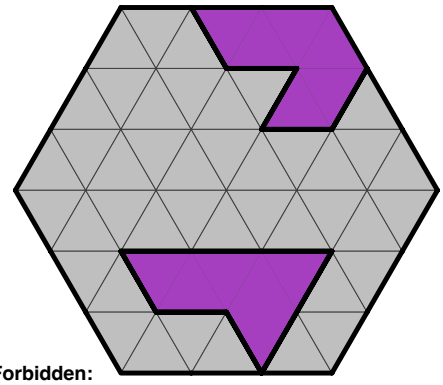
Dots



Forbidden:

9.49185

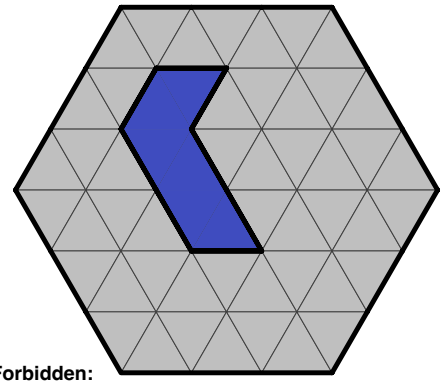
Dots



Forbidden:

10.4919

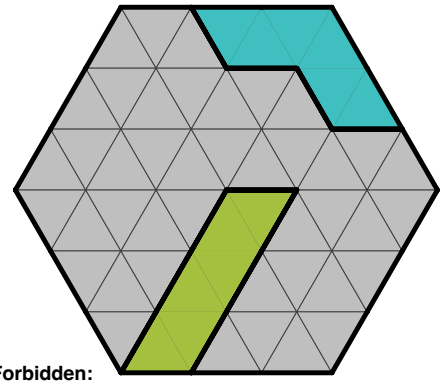
Dots



Forbidden:

12.4594

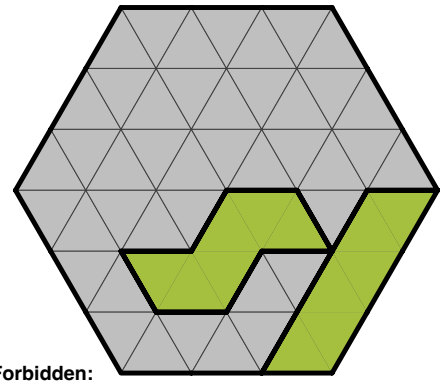
Dots



Forbidden:

13.6618

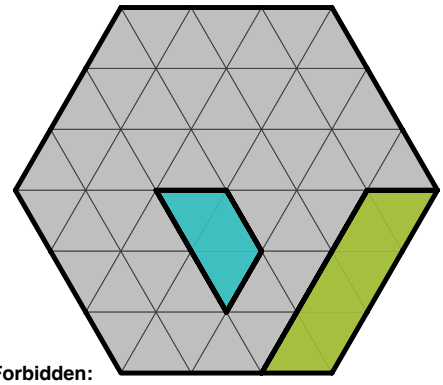
Dots



Forbidden:

15.0362

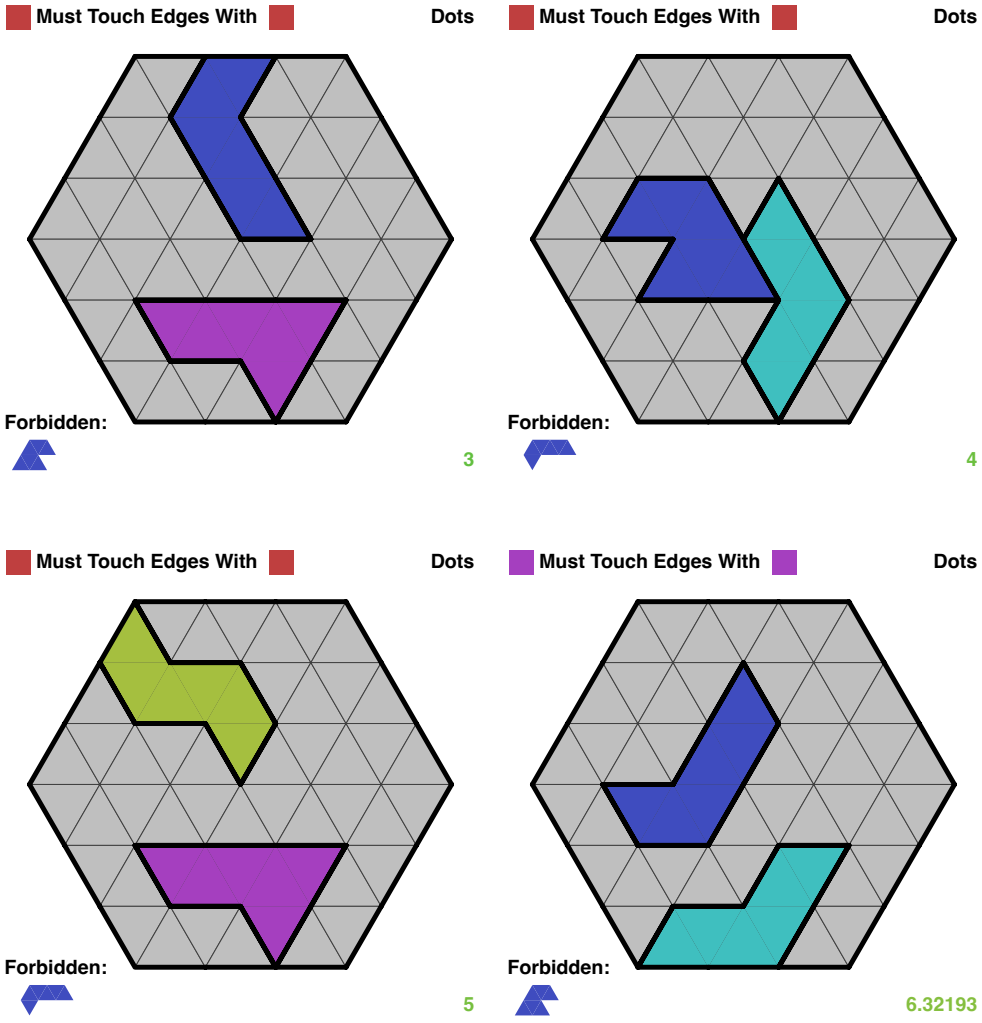
Dots



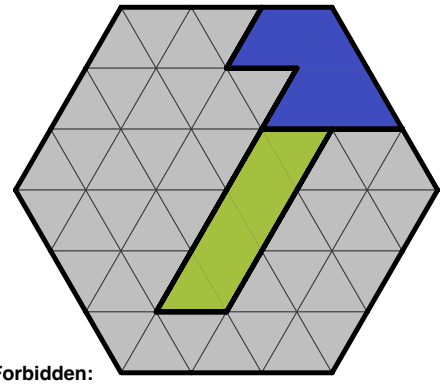
Forbidden:

18.2512

7.1.8 Chapter 8



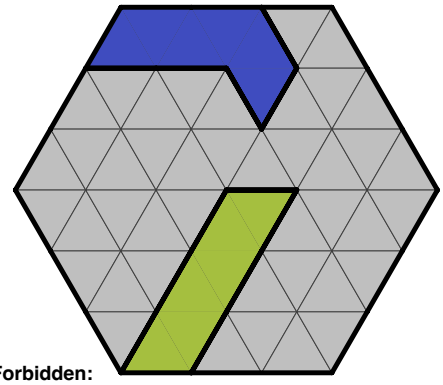
Must Touch Edges With Dots



Forbidden:

7

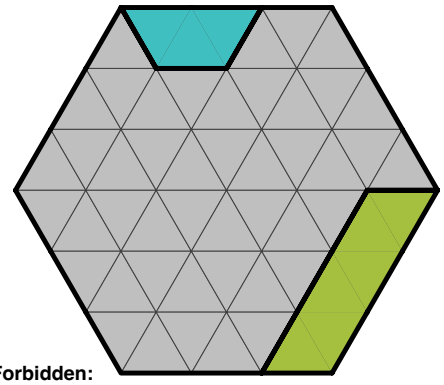
Must Touch Edges With Dots



Forbidden:

8.16992

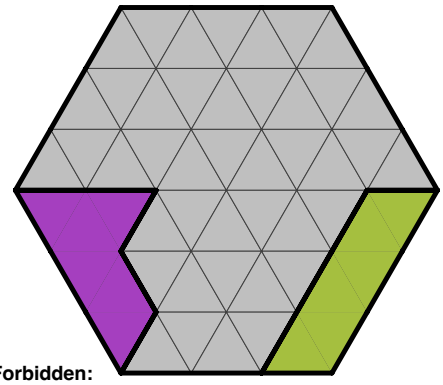
Must Touch Edges With Dots



Forbidden:

10.3399

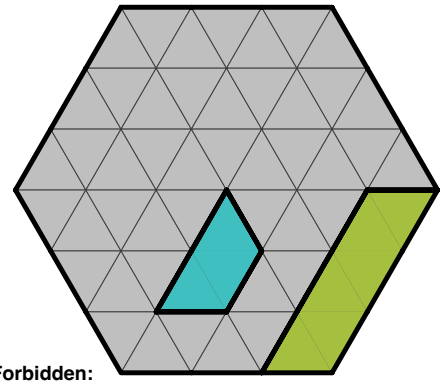
Must Touch Edges With Dots



Forbidden:

11.9513

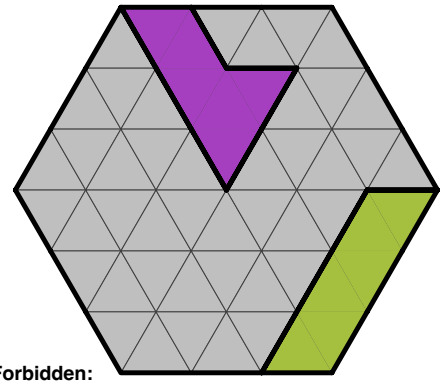
Must Touch Edges With Dots



Forbidden:

12.9513

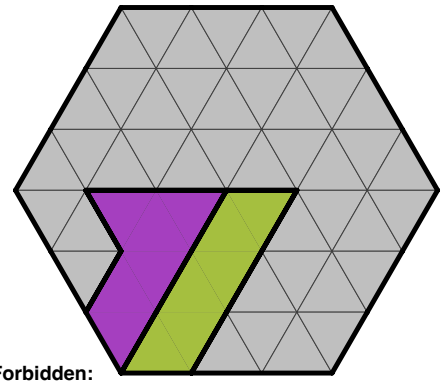
Must Touch Edges With Dots



Forbidden:

14.1472

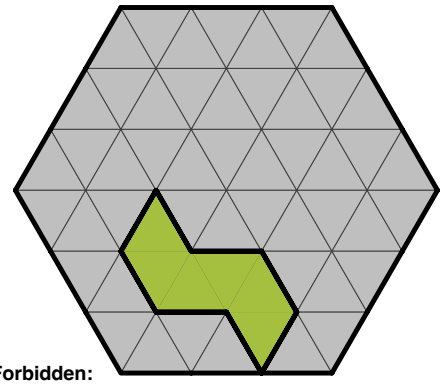
Must Touch Edges With Dots



Forbidden:

16.7993

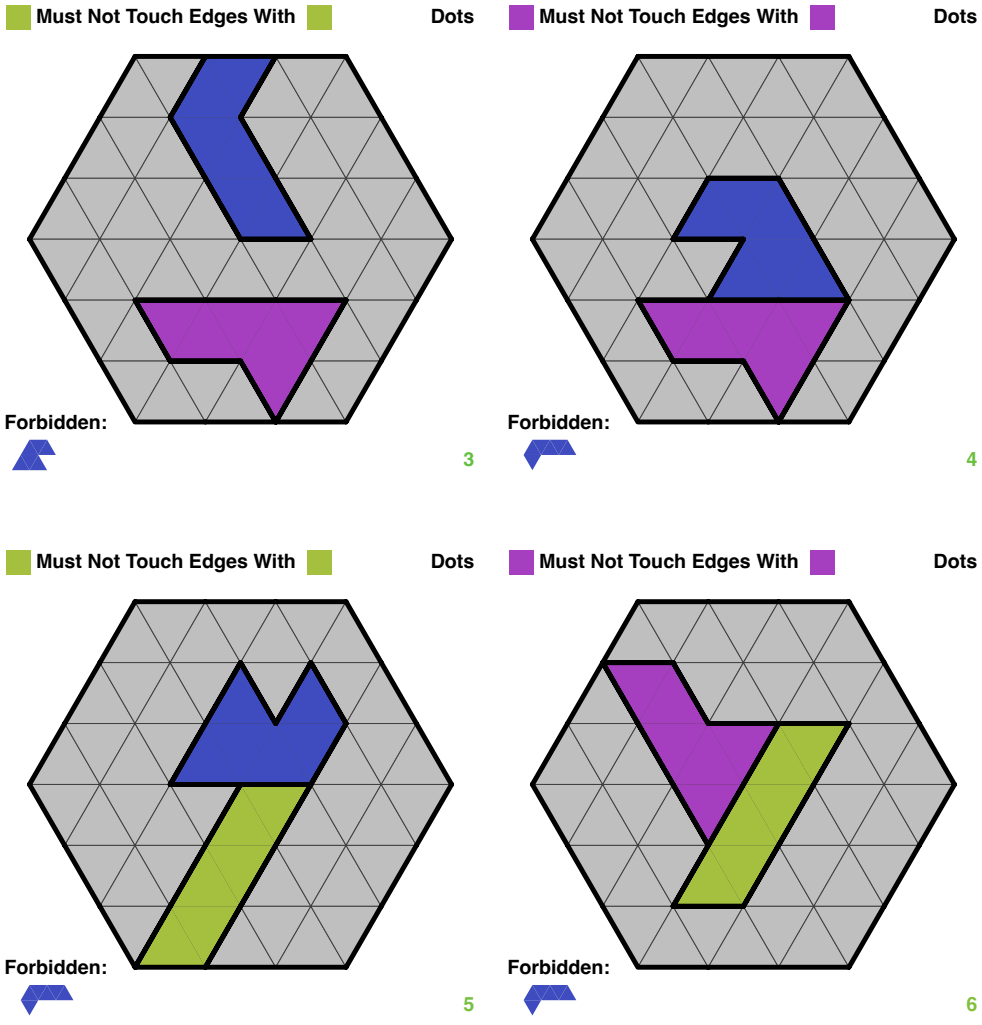
Must Touch Edges With Dots



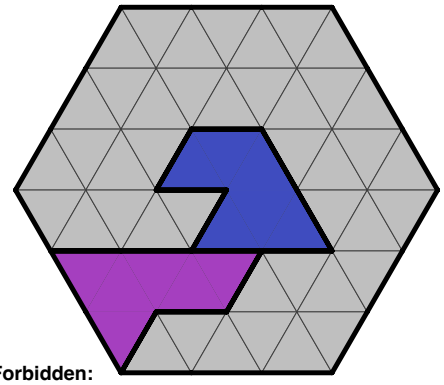
Forbidden:

19.8927

7.1.9 Chapter 9



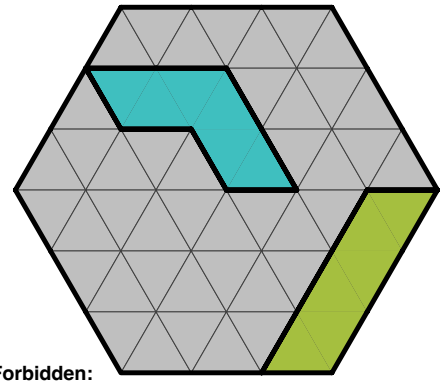
Must Not Touch Edges With Dots



Forbidden:

7

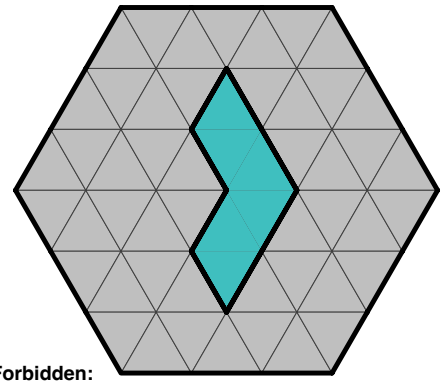
Must Not Touch Edges With Dots



Forbidden:

9.32193

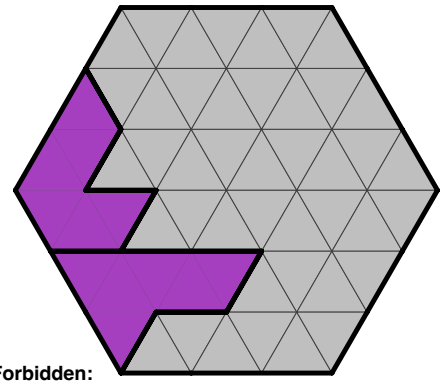
Must Not Touch Edges With Dots



Forbidden:

10.1699

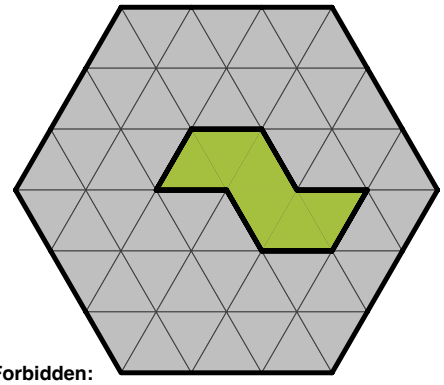
Must Not Touch Edges With Dots



Forbidden:

11.1699

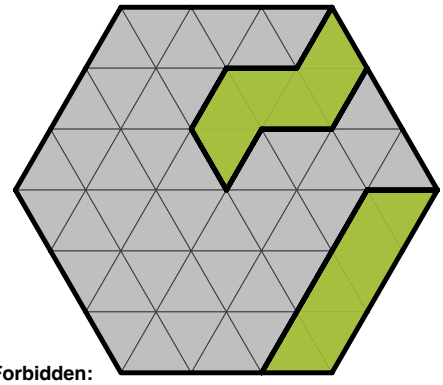
Must Not Touch Edges With Dots



Forbidden:

12.3663

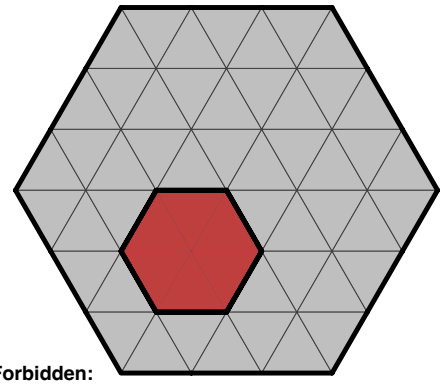
Must Not Touch Edges With Dots



Forbidden:

14.6294

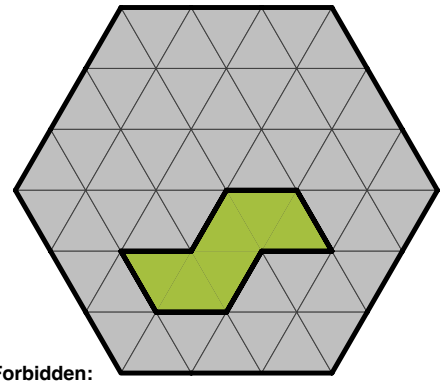
Must Not Touch Edges With Dots



Forbidden:

17.1472

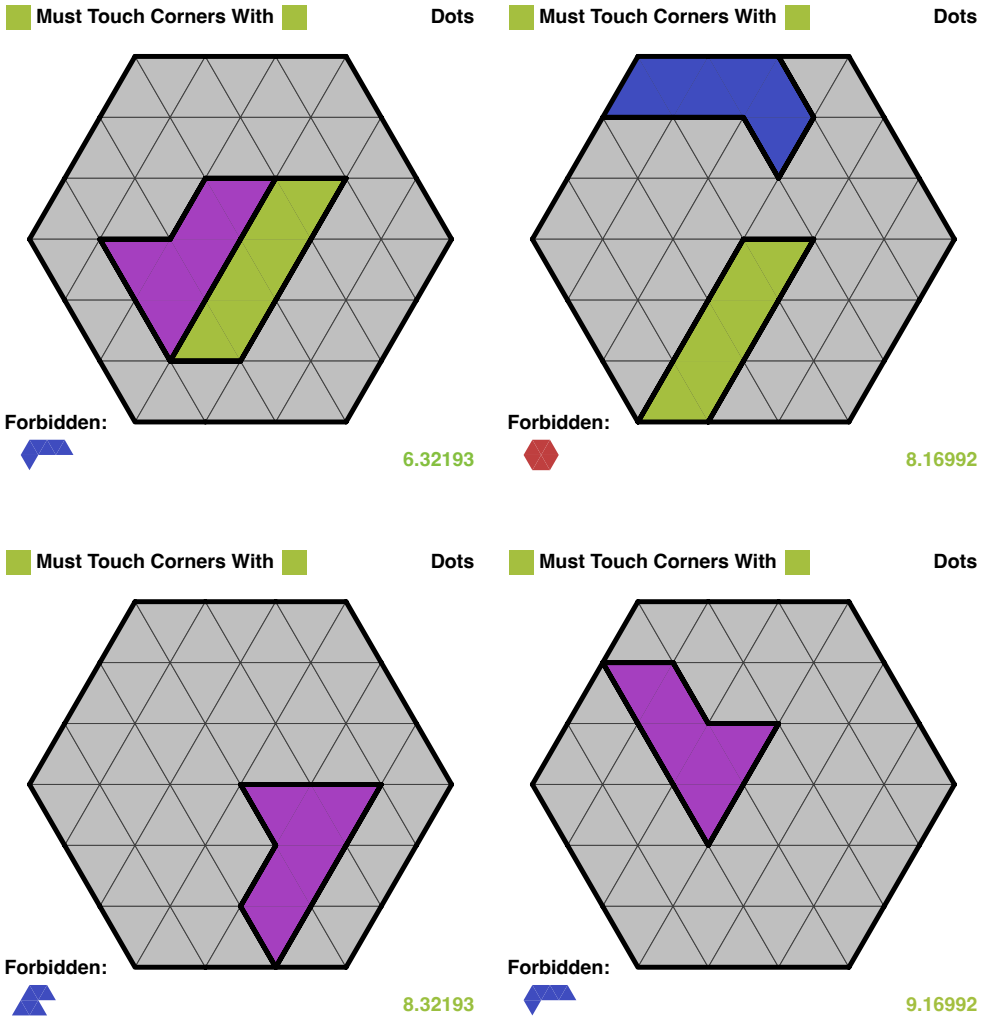
Must Not Touch Edges With Dots



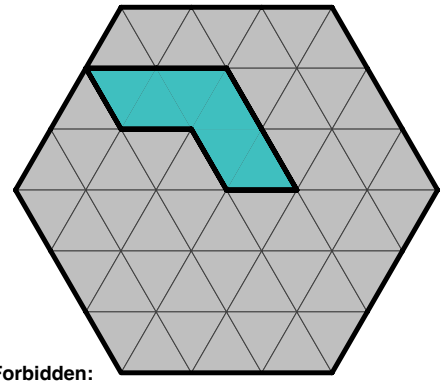
Forbidden:

19.2034

7.1.10 Chapter 10



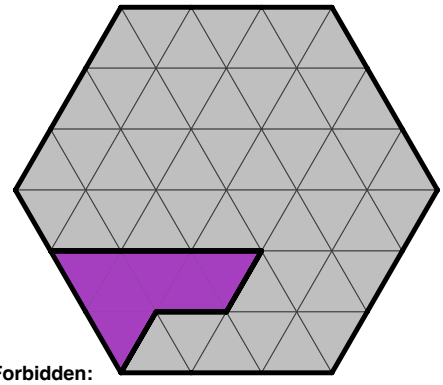
Must Touch Corners With Dots



Forbidden:

10.2288

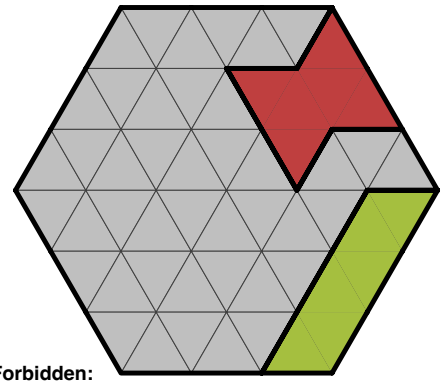
Must Touch Corners With Dots



Forbidden:

11.4553

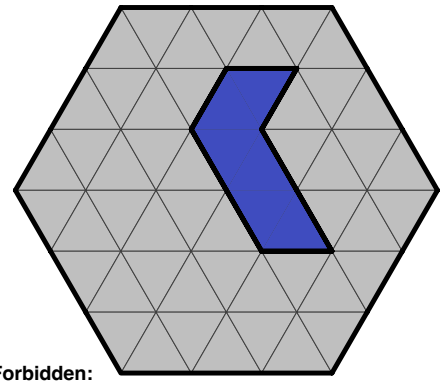
Must Touch Corners With Dots



Forbidden:

11.9069

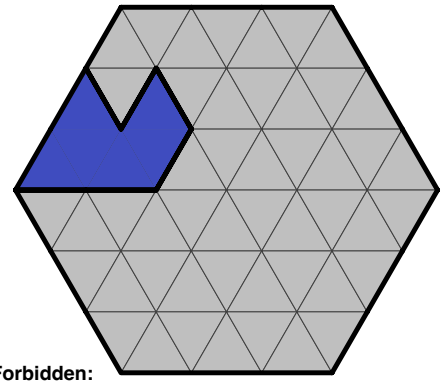
Must Touch Corners With Dots



Forbidden:

12.4367

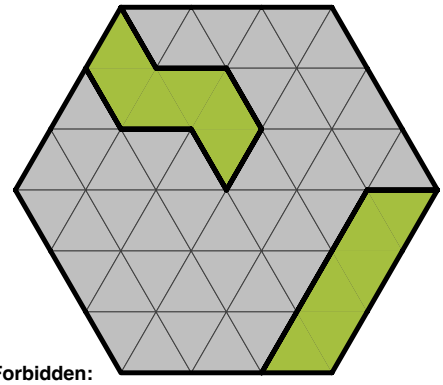
Must Touch Corners With Dots



Forbidden:

13.0224

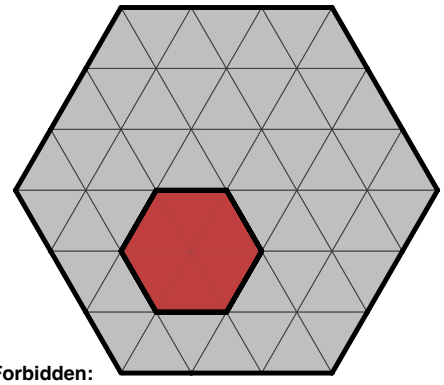
Must Touch Corners With Dots



Forbidden:

14.4367

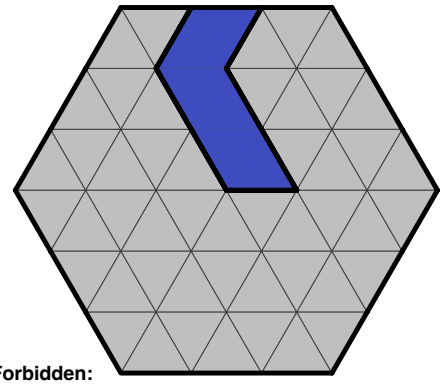
Must Touch Corners With Dots



Forbidden:

17.1472

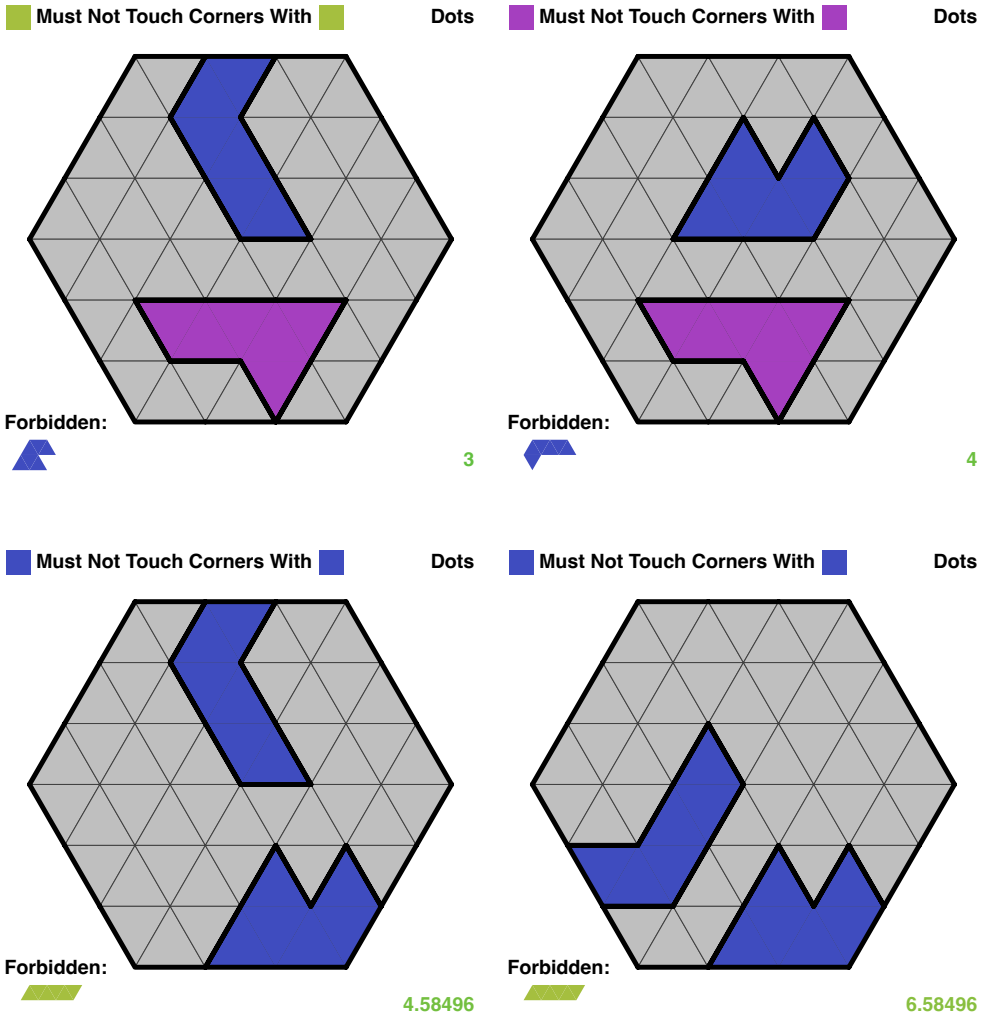
Must Touch Corners With Dots



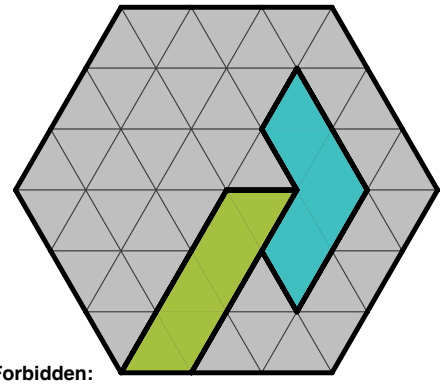
Forbidden:

18.3399

7.1.11 Chapter 11



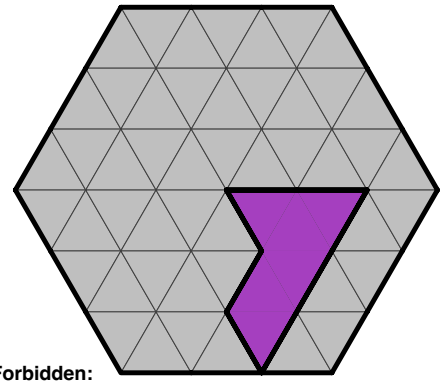
Must Not Touch Corners With Dots



Forbidden:

7

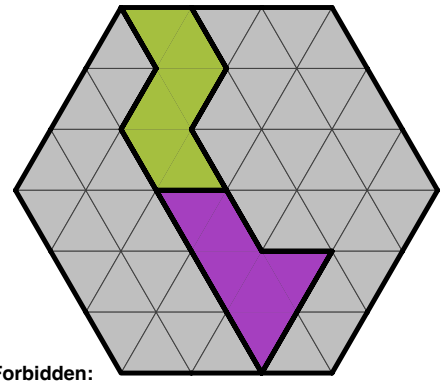
Must Not Touch Corners With Dots



Forbidden:

8.32193

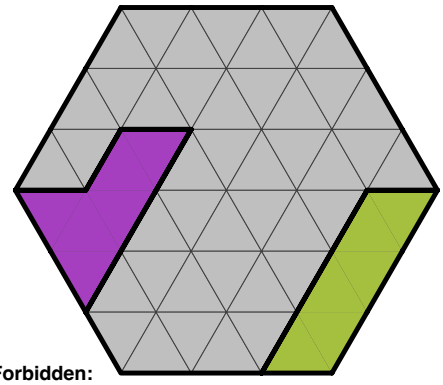
Must Not Touch Corners With Dots



Forbidden:

9.64386

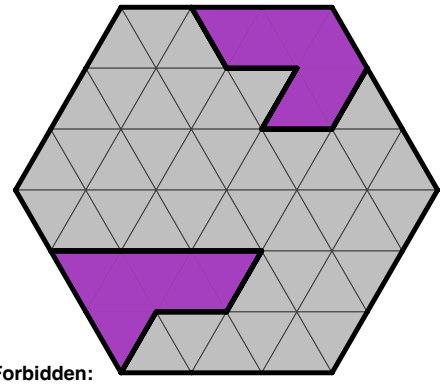
Must Not Touch Corners With Dots



Forbidden:

10.5622

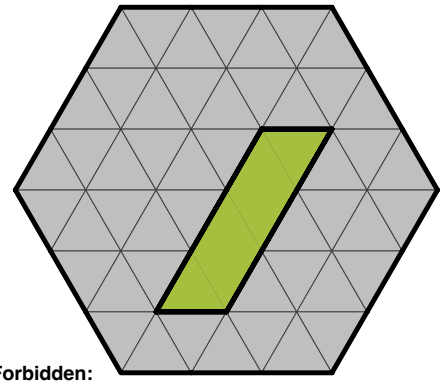
Must Not Touch Corners With Dots



Forbidden:

12.3399

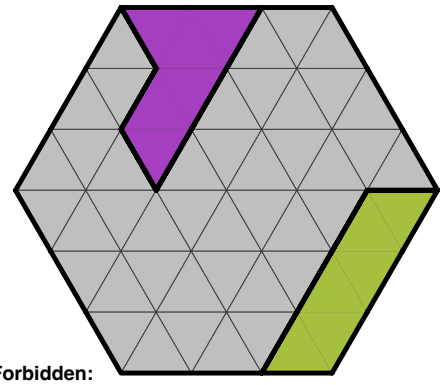
Must Not Touch Corners With Dots



Forbidden:

13.5887

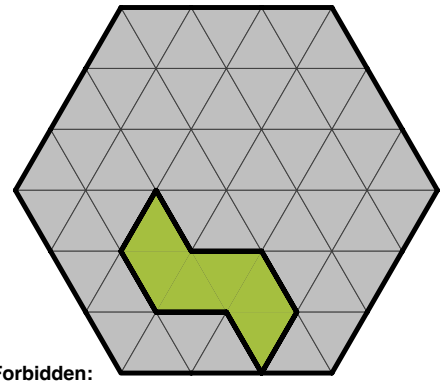
Must Not Touch Corners With Dots



Forbidden:

14.6618

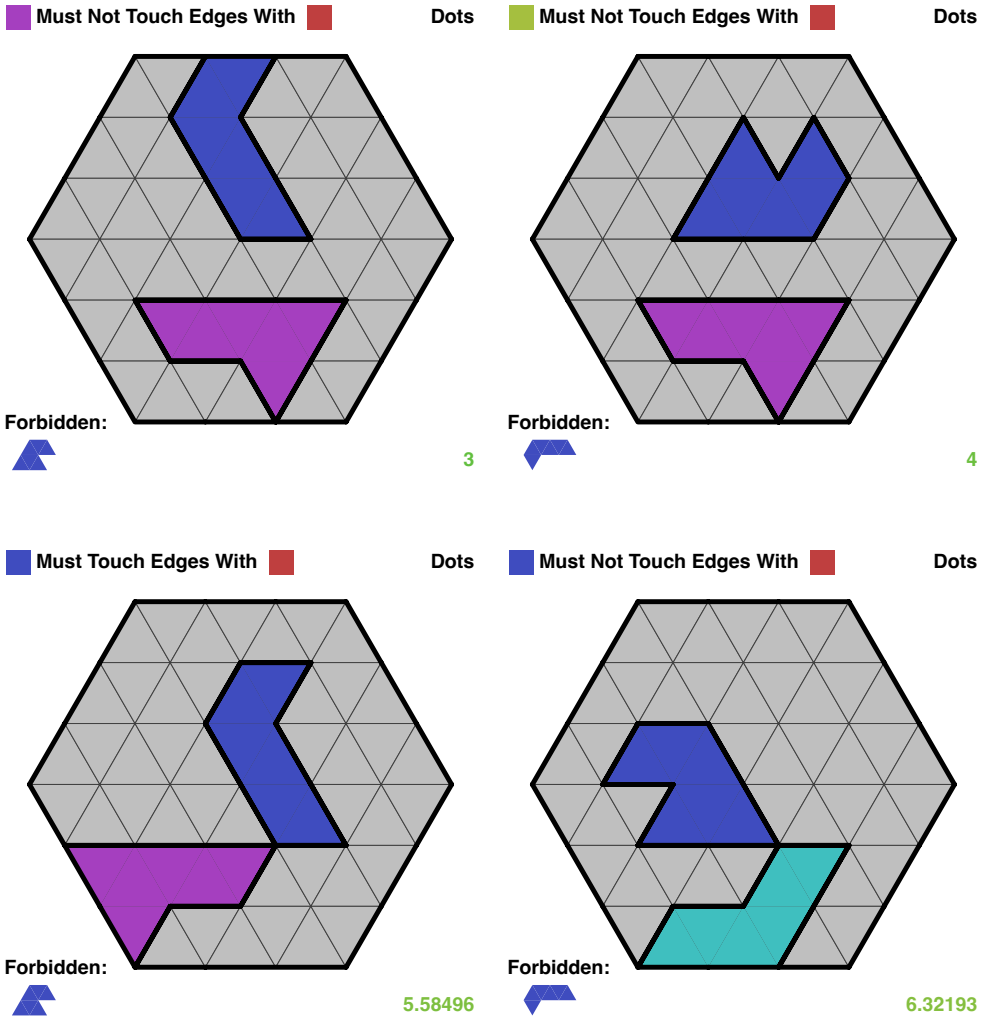
Must Not Touch Corners With Dots



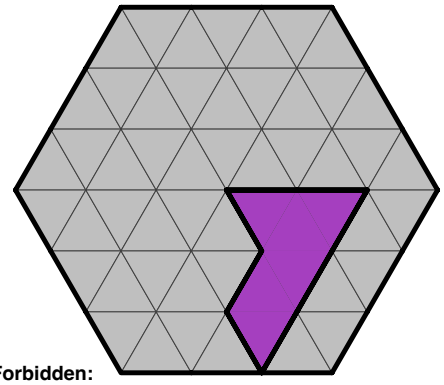
Forbidden:

19.8927

7.1.12 Chapter 12



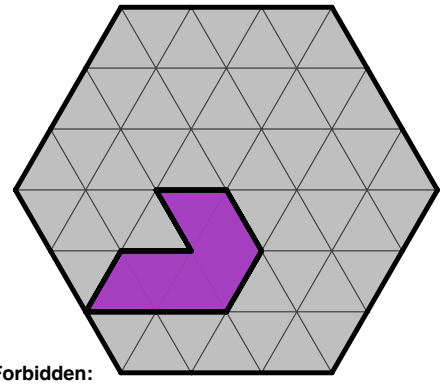
Must Not Touch Corners With Dots



Forbidden:

8.32193

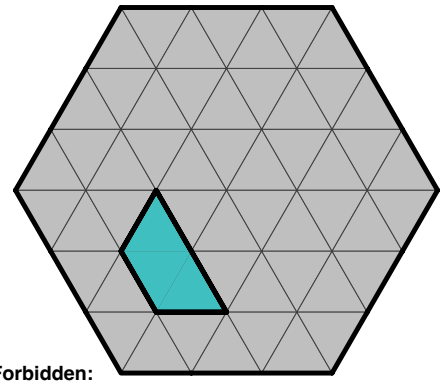
Must Not Touch Corners With Dots



Forbidden:

10.1699

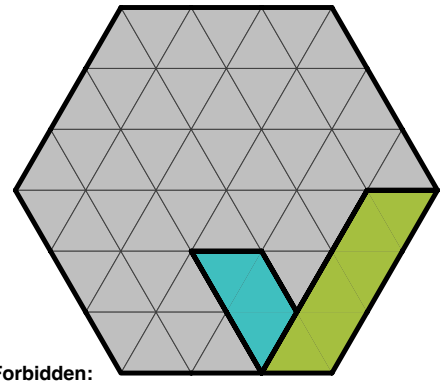
Must Touch Edges With Dots



Forbidden:

11.4798

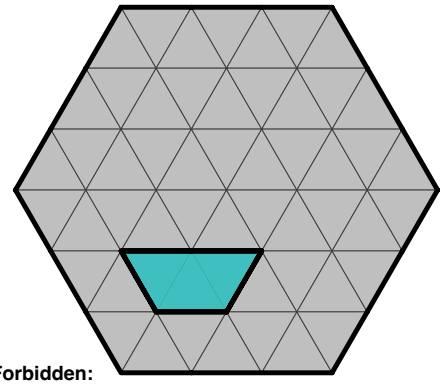
Must Not Touch Corners With Dots



Forbidden:

12.9248

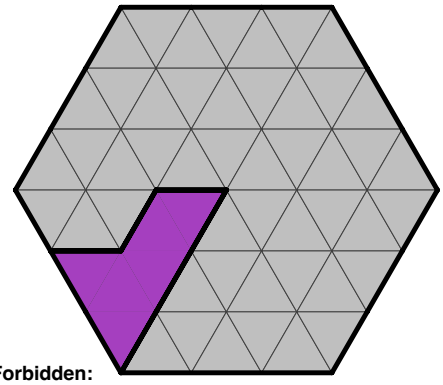
Must Not Touch Edges With Dots



Forbidden:

13.1997

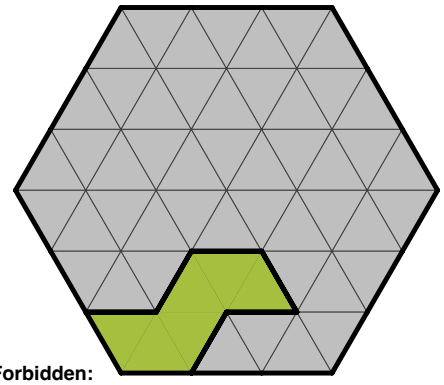
Must Not Touch Edges With Dots



Forbidden:

17.7398

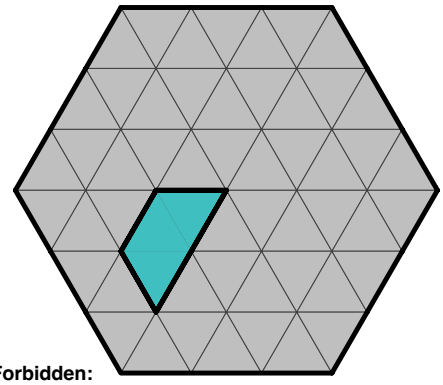
Must Not Touch Edges With Dots



Forbidden:

19.7228

Must Touch Corners With Dots



Forbidden:

20.0496