Quantifying over Play: Constraining Undesirable Solutions in Puzzle Design

Adam M. Smith, Eric Butler, Zoran Popović
Center for Game Science, Dept. of Computer Science & Engineering, University of Washington
{amsmith,edbutler,zoran}@cs.washington.edu

ABSTRACT

Motivated by our ongoing efforts in the development of *Refraction 2*, a puzzle game targeting mathematics education, we realized that the quality of a puzzle is critically sensitive to the presence of alternative solutions with undesirable properties. Where, in our game, we seek a way to automatically synthesize puzzles that can *only* be solved if the player demonstrates specific concepts, concern for the possibility of undesirable play touches other interactive design domains. To frame this problem (and our solution to it) in a general context, we formalize the problem of generating solvable puzzles that admit no undesirable solutions as an $\mathsf{NP}^{\mathsf{NP}}$-complete search problem. By making two design-oriented extensions to answer set programming (a technology that has been recently applied to constrained game content generation problems) we offer a general way to declaratively pose and automatically solve the high-complexity problems coming from this formulation. Applying this technique to *Refraction*, we demonstrate a qualitative leap in the kind of puzzles we can reliably generate. This work opens up new possibilities for quality-focused content generators that guarantee properties over their *entire* combinatorial space of play.

Categories and Subject Descriptors

K.8.0 [**Personal Computing**]: General – Games; I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving – Logic programming

Keywords

game design, procedural content generation, answer set programming, educational games, puzzle games

1. INTRODUCTION

Procedural content generation (PCG) can assist game designers both offline, in the form of design assistance and automation tools, and online, in the form of fully-automated content generators embedded into games. PCG is often motivated as a way to create a larger space of individual artifacts (such as the generated weapons of *Galactic Arms Race* [9] and *Borderlands* (Gearbox Software 2009)) or creating single artifacts with many more details than than would be reasonable to author via traditional means (such as the detailed dungeons of *Diablo 3* (Blizzard Entertainment 2012) or the pseudo-infinite terrains of *Minecraft* (Mojang 2009)). On a quantity–quality tradeoff spectrum, most PCG systems will try to complement human authoring efforts by providing additional *quantity* (more artifacts/details) in exchange for reduced authorial control. By contrast, in this paper, we are interested in developing game design automation systems (starting in the domain of puzzle games) that assist their human collaborators by outperforming them in regards to *quality*: eliminating broad classes of shortcut solutions that would undermine a puzzle's intended purpose. Our primary strategy is to apply exhaustive (but intelligent) search over the entire combinatorial space of play, a space too large to feasibly check by hand. In this paper, we use the term "shortcut solution" to denote any gameplay a designer would find undesirable, even when the artifact in question is not a puzzle (perhaps another kind of level or interactive story) or when makes the shortcut undesirable is not a property literally relating to its length.

1.1 The Refraction Project

Refraction[1] is a puzzle game that targets mathematics education. Because *Refraction* puzzles intentionally entangle spatial and mathematical challenges, they have the capacity to present players with deep challenges in gameplay. This same depth complicates puzzle design, where we would like to carefully control which concepts are being introduced across the game's level progression. As *Refraction*'s level progressions are altered to be appropriate for different audiences in several ongoing game-based education studies, the game continually demands more original puzzle design effort. Thus far, our team has authored at least eight full sequences of levels (of 20 to 50 levels each). To support fine-grained player adaptivity in the upcoming game *Refraction 2*, we will need both a massive increase in the quantity of puzzles designed as well as an increase in the level of quality assurance applied to each puzzle.

In previous work [13], we created a set of in-house design automation tools using answer set programming (ASP) to separate the definition of the *Refraction*-specific design spaces from the domain-independent combinatorial search

[1] `http://games.cs.washington.edu/refraction/`

tools used to generate puzzles and search for alternative solutions. Given high-level gameplay requirements (such as that a level involve using certain pieces, building certain mathematical expressions, or selecting amongst a number of distracting pieces), our previous synthesis tool created a puzzle-and-solution pair where the solution demonstrated the required concept.

Ensuring that a level *could* be solved in a certain way was not enough, however. Often, with the addition of distractor pieces that are not part of an intended solution (and sometimes even without them), there were alternative ways to solve a puzzle that did not demonstrate the target concept. To automatically determine if a puzzle could be solved in a certain alternative way (perhaps never constructing a required fraction), we also created an analysis tool that could reason over all possible ways a given puzzle could be played.

In order to express our interest in solvable puzzles that admit no undesirable solutions, we needed a system that combined the competencies of our previous tools. Simply combining them in a pipeline (where one proposes a stream of solvable puzzles and the other filters them by the presence of shortcut solutions) would be unsatisfactory. First, there would be enormous inefficiencies when the synthesis tool continually proposes variations on a common idea that the analysis tool could determine to always fail for the same reason—the pipeline system would not learn from its mistakes. Secondly, when the concept we wanted to require in all solutions was not actually enforceable (perhaps it requires constructions too large for the game's fixed grid space or was indirectly contradictory in other wise), this lack of feedback would force the synthesis tool to completely enumerate the space of puzzles before reporting that our request was unsatisfiable.

What was needed was a new *problem formulation* that would allow us to express the concerns of *solvable* puzzle generation and *exhaustive* machine playtesting at the same time. This formulation would allow rapidly exploring alternative encodings of design concerns as the nature of high-quality puzzles is iteratively discovered. To solve these problems, we needed new and reusable infrastructure for content generation that would allow us to quickly and concisely express the idea that what makes content appropriate may depend on the *entire* combinatorial space of play.

1.2 Contributions

To address the general challenge sketched above, this paper makes three primary contributions:

- We identify many interactive design problems as being critically sensitive to the presence of shortcut behaviors. This entails a natural formulation of the design automation problem as an $\mathsf{NP}^{\mathsf{NP}}$-complete search problem.
- We offer a translation-based extension to normal ASP that allows elevated-complexity modeling by means of design-appropriate language constructs, allowing the direct reuse of state-of-the-art search techniques.
- We report on an application of this technique to the full complexity of an existing, deep puzzle game, demonstrating the ability to pose interesting constraints quantified over the entire space of play.

2. RELATED WORK

Stepping back from *Refraction*'s need for a particular kind of generated puzzle levels, this section examines how shortcuts are typically addressed in level generation projects. There has been much work in automatically generating levels; however, work that makes strong guarantees that generated levels will be solvable is rare. Work guaranteeing some property over the space of solutions is even rarer. The strategies described in this section all contrast with the standard for manual level design: thinking very carefully about the game's mechanics, playtesting levels thoroughly, iterating across many variations, partially railroading players with obvious checkpoints, and occasionally letting levels with undesirable solutions pass into production.

2.1 Softening with Game Mechanics

The first strategy is that of designing games so that any level is solvable with enough effort or that solving individual levels is less important because there many higher-level routes to success.

In *Spelunky* (Derek Yu 2009), players navigate a maze-like cave populated by critters, traps, and treasures. At a coarse grid level, *Spelunky*'s level generator plans a path between the entrance and exit rooms [17]. To fill in rooms with detail, a random room template is selected and populated. Depending on how the templates are populated, the final level may not be solvable by the particular original coarse path. To mitigate this, the player is given ropes to climb distances that cannot be jumped or bombs to blast through walls that cannot be avoided, and they can backtrack to gather more of these items. In this way, the *mechanics* of *Spelunky* virtually guarantee there is a solution for every generated level.

This clever design solution is not applicable to all game designs. For example, the designer may not want to allow the player to alter the level through destructive actions for technical or artistic reasons. In light of *Refraction*'s educational goals, we want a tighter level of control over solvability conditions than the softening strategy allows.

2.2 Sparse Solutions by Reference Agents

If an algorithm for computing a reference solution is available, we could use this algorithm in the inner loop of a puzzle generator or as a post-filtering process to ensure any generated puzzle has a solution. Uses of this strategy often also assume that properties of the reference solution are indicative of the properties of the puzzle as a whole.

The reference agent strategy was adopted by the developers of the upcoming game *Cloudberry Kingdom* [5], a project that foregrounds the generation of very complex platforming levels with solvability guarantees. By interleaving calls to a reference agent, the outer loop of level design can be sure it places platforms just within the player's expected path and dangerous enemies/traps just outside. In an academic context, Shaker et al. [11] employ a similar strategy to generate *Infinite Mario Bros* (Markus Persson 2006) levels that are personalized according to an estimate of a player's experience of fun on that level. In both of these projects, properties of the sparse set of reference solutions generated for each level are used to estimate the overall appropriateness of the level. Sometimes, a level will also admit alternate

solutions that might be nearly as likely to occur in human play as the reference solutions. When these realistic alternative solutions differ dramatically, they undermine the use of the reference agent as an informative model for a player.

2.3 Minimal Solutions

In games with highly-nonlinear play, knowing the properties of any single solution tells us very little about what it is like to play a level in general. Knowing that a maze has a long solution, for example, does not imply the absence of a shorter one.

The strategy of generating solvable puzzles with a known-minimum solution length enforces one kind of property over all of a puzzle's solutions: they are all at least as long as the minimum. This is commonly done by employing a reference agent that produces guaranteed-optimal solutions. In Ashlock's chess and chromatic maze generators, a dynamic programming algorithm exactly determines a generated puzzle's minimum solution length [1]. The Sokoban level generator by Taylor and Parberry applies this strategy in reverse, searching from a solved game state to find a guaranteed-distant game state and then naming that as the initial state [16]. Attempting to apply this technique to a binary property (e.g. the player takes a critical action or they do not) yields a degenerate metric that lacks the informative gradients on which many optimization algorithms rely when searching large design spaces.

2.4 Declarative Design Space Models

The final strategy, of exhaustive symbolic reasoning over the space of all puzzles and solutions, is not yet a common strategy for producing content generators. The general idea of representing a space of game content artifacts using answer set programming (ASP, a declarative programming paradigm focused on complex combinatorial search and optimization problems [6]) was proposed recently [15]. In addition to enabling our previous tools for *Refraction* [13], this strategy has been applied to real-time strategy maps [2], mini-game rulesets [14], platformer levels [12, sec. 10.5.3], and opponent behavior [3].

This strategy has been most successful for generation problems in the complexity class NP, for which ASP provides an expressive modeling language. However, as we will show in Section 5, the general problem of shortcut-free generation lives outside of NP. Although there are many off-the-shelf tools that solve problems at the required level of complexity (e.g. general theorem provers), the challenge is in figuring out how to encode puzzle design problems in their modeling languages without first becoming an expert in formal logic.

3. REFRACTION

In this section, we briefly review the mechanics and design challenges of our game as a concrete point of reference for our general problem formulation.

Figure 1 shows a typical puzzle and an example piece from *Refraction 2*. The player's goal is to place a given set of puzzle pieces so that laser beams flow from fixed laser sources, through player-arranged pieces (which may split and recombine beams to form different fractions), into fixed laser targets with required power levels. The various types of pieces

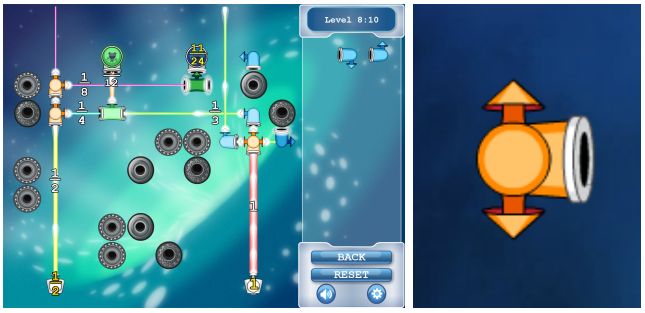

Figure 1: Screenshot of *Refraction 2* and detail of a splitter piece showing input/output ports. Players should drag pieces from the tray onto the board to form a network of laser beams that satisfies targets with the correct fraction. Pieces cannot be rotated.

in the game (sources, targets, benders, blockers, splitters, and combiners) present a the player with a mixture of spatial and mathematical obstacles. Setting up these challenges requires a mastery of these entangled mechanics.

Because the type and input/output port configuration of each piece is fixed by the puzzle's designer, the player must carefully allocate piece usage between the different goals presented by a puzzle. Meanwhile they must ignore extra pieces, called distractors, that a designer may add to a puzzle. When there are multiple sources and multiple targets, we often make careful spatial/mathematical design choices so that players cannot simply connect the nearest sources and targets. Combining dataflow-like mechanics with the compact game board, the potential for deep puzzles is enormous (as is the potential for unforeseen solutions).

What makes a *Refraction* puzzle difficult, interesting, or elegant is not easy to state. Simply heaping on distractor pieces in an attempt to hide a solution often only serves to give players the tools to build a simpler solution. Likewise, scattering many blockers around a puzzle can work to give so many spatial clues to a puzzle's solution that mathematical reasoning is not required to solve it—in simply using available pieces to route around the cloud of blockers, players may find that they have accidentally solved the mathematical challenge in passing. Judging the quality of a *Refraction* puzzle requires a much deeper analysis of the play it affords.

We do not claim to have completely characterized the judgement of a puzzle's quality or appropriateness to a point in a level progression. We have, however, uncovered a very general type of constraint that we would like to place on nearly every generated puzzle: that no detectably-undesirable arrangement of pieces is a valid solution to the generated puzzle. Although our example results in § 6.4 focus on easily describable kinds of shortcuts, our internal use of this idea will define shortcuts with respect to data-derived player models.

4. SHORTCUTS IN THE WILD

Shortcuts (or undesirable solutions generally) can occur in many forms. In *Refraction*, two kinds of shortcuts stand out as the most prominent annoyance in both manual and automated puzzle design. In *spatial* shortcuts, we intend

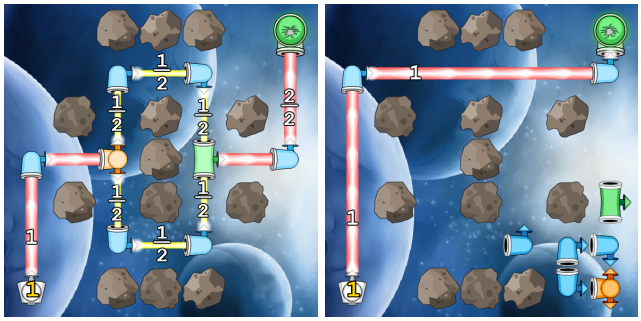

Figure 2: Puzzle with shortcuts. Left: an optimistic reference solution that demonstrates equal partitioning and reassembly (part of the game's educational content). Right: an alternate, shortcut solution using only two bender pieces.

that the player will build a particular chain of benders to route a laser around obstacles; however, in a shortcut solution, there is a way to leave most of those benders off of the board. In *mathematical* shortcuts, we intend that the player will construct certain mathematical expressions, but there is a shortcut solution (which may actually involve *more* pieces) in which the key expression is never realized. See Figure 2 for an example of both types of shortcuts.

Even veteran *Refraction* level designers are susceptible to creating puzzles with undesirable solutions. One example our team encountered was when a level designer, collaborating with a learning scientist, was trying to create a level that tested construction of $^1/_{12}+^1/_{12} = ^2/_{12}$. In order to trap the common error of players thinking that $^1/_6 + ^1/_6 = ^2/_{12}$ (that both the numerator and denominator should add), two $^1/_6$ sources were added to the level. It was not until much later that we realized the shortcut: a player could use just one of the distracting $^1/_6$ sources (which were intended to be left unused in the ideal solution) to solve the puzzle without practicing addition at all.

Shortcuts are hardly unique to *Refraction*. Consider designing a piece of interactive fiction (IF) like the classic *Anchorhead* [8]. IF games can involve several overlapping puzzles that entangle movement, inventory, crafting, and character relationship mechanics. Often, an author will imagine a sparse set of reference solutions to their game that take the player on an interesting tour of the story's plot, setting, and characters. Exacerbated by the non-visual nature of IF games, there may be many shortcut solutions that bypass the key narrative details or allow it to be experienced in an undesirable order. Drama management systems, such as the system Nelson and Mateas [10] applied to *Anchorhead*, make small edits to the story world as the player plays (silently locking or unlocking doors, moving items to alternate locations, etc.) in an effort to guide the player through interesting experiences. However, there are as yet no tools available to help authors manage the properties of the combinatorial space of alternate solutions at design-time.

Many game designs are likely to be critically sensitive to the presence of undesirable solutions. Quality-focused content generators, we claim, should actively seek out their presence.

5. QUANTIFYING OVER PLAY

In this section, we formalize the problem of generating solvable puzzles that are free of shortcuts. Even though this section makes use of the vocabulary of formal logic, the non-formalist reader will still benefit from seeing how the concerns of puzzle design (of puzzle creation and subsequent playtesting) are reflected in the logical setting.

To make it easier to ascribe intent to the various structures to come, we introduce two characters. Elise is a game designer who wants to create a puzzle; she wants to show that there *exists* a puzzle-and-solution pair with desirable properties. Fiona is a design assistant who is asked to examine Elise's puzzles and ensure they are free of shortcuts; she wants to show that *for all* possible ways to solve a puzzle, none are problematic. Later we will show how to automate the work of Elise and Fiona using state-of-the-art search techniques.

Although we focus on puzzles and solutions to stay close to our motivational setting, the reader should feel free to replace these terms with those of their own design domain. For example, platformer or role-playing game world designs would count as puzzles where reaching a final platform or completing the main quest count as the solution conditions. Finishing within a certain time or experiencing a critical narrative sequence are valid concepts.

5.1 Basic Puzzle Generation

Before jumping into reference/alternate solutions, we first consider the basic puzzle generation problem: proposing a puzzle with efficiently detectable properties guaranteed of its form. Symmetric placement of blockers in *Refraction* is an example of this type of property, as are ensuring there are source pieces with certain fraction values. Basic puzzle generation is part of Elise's job. Formally, she is asked to prove (by construction) the following statement, where p is a vector of Boolean variables that define a puzzle:

$$\exists\, p\ \texttt{Form}(p)$$

Assuming evaluating the `Form` predicate on a puzzle is easy (formally, in the complexity class P), the basic puzzle generation problem has complexity NP—it is the problem of non-deterministically guessing a candidate puzzle such that its form can be quickly verified. As some problems in NP are easier than others, it is not surprising that many generation problems can be solved by efficient, feed-forward generation algorithms (often called constructive generators).

5.2 Solvable Puzzle Generation

Although puzzles with guaranteed good form may look pretty, they might not always be solvable. To generate guaranteed-solvable puzzles in a general way, we use the strategy of generating puzzle-and-solution pairs. This is Elise's full job. In the following statement, p describes the puzzle as before, and s describes a reference solution:

$$\exists\, s, p\ [\texttt{Form}(p) \wedge \texttt{Solves}(s, p)]$$

Here, we have *quantified over the space of play* (potential solutions) using an existential quantifier. Deciding, given a candidate solution, whether that candidate solves the puzzle currently under consideration can be non-trivial. In *Refraction*, checking validity of solutions requires simulating

beam flow through the player's arrangement of pieces. For other games with mechanics approaching the expressiveness of general purpose programming, such as *Light-Bot* (Coolio Niato 2008) and *SpaceChem* (Zachatronics Industries 2011), this check approaches a P-complete problem—it may involve simulating a machine of the player's design for a bounded number of steps.

So, assuming evaluating the `Solves` predicate is in P, the problem of generating solvable levels is also in NP. It bears mentioning that solvable puzzle generation problems may be much bigger than their basic counterparts, however, because it takes more bits to describe a puzzle-and-solution pair than it takes to describe a puzzle in isolation.

5.3 Detecting Shortcuts

Given a specific puzzle, such as those Elise has been creating, we now look at the problem of detecting shortcuts. If t describes a candidate solution, the following statement captures the problem of finding a shortcut solution for a given puzzle described by p where shortcuts are defined as solutions that do *not* practice an arbitrary target concept.

$$\text{given } p,\ \exists\, t\ [\texttt{Solves}(t,p) \wedge \neg\texttt{Concept}(t,p)]$$

As before, we assume the `Concept` predicate can be evaluated in P. `Concept` might simply check that the solution used a type of piece that we wanted the player to practice, or it might be more complex, as in checking if the solution would not have been suggested by a set of heuristics that we think the player may be following (ensuring that solving this puzzle will force a re-evaluation of their strategy).

5.4 Verifying Lack of Shortcuts

Being able to ensure a given puzzle has no detectable shortcuts is Fiona's job. That is, we want to state that a playtest could never reveal a shortcut solution.

$$\text{given } p,\ \neg(\exists\, t[\texttt{Solves}(t,p) \wedge \neg\texttt{Concept}(t,p)])$$

Applying De Morgan's laws and rewriting the inner expression as an implication, Fiona should prove that solving a puzzle implies practicing the required concept:

$$\text{given } p,\ \forall\, t\ [\texttt{Solves}(t,p) \implies \texttt{Concept}(t,p)]$$

As the complement (negation of the result) of a problem in NP, this problem has complexity coNP. For this type of problem, there is no single solution that witnesses the truth of the statement, so Fiona must prove it by contradiction (perhaps finding a set of mutually inconsistent constraints or showing that any possible choice gives rise to such a set).

5.5 Shortcut-free Puzzle Generation

Moving to the crux of this paper, we look at how Elise and Fiona can collaborate to create solvable puzzles that admit no shortcut solutions. Stated formally, this is their task:

$$\exists\, s,p\ \forall\, t\ (\texttt{Solves}(s,p) \wedge [\texttt{Solves}(t,p) \implies \texttt{Concept}(t,p)])$$

Problems of the form $\exists e \forall u\, \Phi(e,u)$ (where Φ is any quantifier-free expression) are known as 2QBF problems. 2QBF is the canonical problem for the class $\mathsf{NP}^{\mathsf{NP}}$. Because shortcut-free puzzle generation and 2QBF are mutually reducible (`Concept` could be any Φ), our formulation of the shortcut-free puzzle generation problem is $\mathsf{NP}^{\mathsf{NP}}$-complete. That is to say, any problem other in $\mathsf{NP}^{\mathsf{NP}}$ can be posed as a question about puzzles requiring concept practice.

As easily as we can imagine Elise and Fiona working in a strict pipeline (where Elise enumerates solvable puzzles that Fiona will verify), we can also imagine a scenario where Elise and Fiona work in a more cohesive way, sharing notes and learning from each other's feedback. By formulating their combined task as a single problem, we now have a way to directly automate this second, more sophisticated scenario using the declarative programming techniques in the following section.

6. PUZZLE GENERATION

In this section we walk through an application of the above problem formulation to generating shortcut-free puzzles for *Refraction 2*. First, we review answer set programming, the technology used in our previous generation of design automation tools. Next, we describe the answer set program that models our design space for *Refraction* puzzles, making use of two special constructs to setup the shortcut-free generation problem. Then we step through the very small (and very general) meta-program that automatically transforms our design space model into one with the desired shortcut-free semantics. Finally, we review some example puzzles generated by defining shortcuts in different ways.

6.1 ASP Refresher

In answer set programming [6], programs declaratively specify a space of *answer sets*: sets of statements that are true in different solutions to the program. AnsProlog (the modeling language used by ASP systems, distinct from Prolog) programs are composed of facts and rules. A fact like `type(4,source)` formally states that the `type` relation holds between the objects identified by the symbols `4` and `source`. In the context of *Refraction* puzzle design, however, it expresses the idea that piece #4 will be a laser source.

Three types of rules, employing variables to talk about many facts at a time, allow an answer set solver to infer additional facts from those given. Choice rules allow the solver to *guess* that a fact may be true. Deductive rules force the solver to *deduce* certain facts given the presence of others. Finally, integrity constraints *forbid* solutions with certain properties. Applied in *Refraction*, the following snippet shows a use for one rule of each type:

```
% guess exactly one fraction power level for each source piece
1 { source_power(P,F):fraction(F) } 1 :- type(P,source).

% deduce whether solution practices use of a piece type
practiced(T) :- type_name(T), 2 { active(P):type(P,T) }.

% forbid leaving any target piece unpowered
:- type(P,target), not target_powered(P).
```

Another type of rule is supported only by some answer set solvers: disjunctive rules. Although disjunctive rules are required to model $\mathsf{NP}^{\mathsf{NP}}$-complete problems in ASP, their use is strongly discouraged in practical applications that do not absolutely require them. The creators of the same disjunctive ASP tools we employ characterize disjunctive modeling as "rather involved and hardly usable by ASP laymen" [7]—a gloomy state that, in the interest of the "ASP laymen"

wishing to solve practical design automation problems, we resolve in this paper.

Using a collection of facts and rules, a designer can capture a *design space model* that describes the space of appropriate artifacts without committing to a particular method of generating those artifacts or ever explicitly using logical quantifiers [15]. Although we can imagine answer set solvers repeatedly sampling combination of *guesses*, *deducing* their implications, and emitting the result if it has no *forbidden* properties in a kind of generate-and-test process, this is not how common solvers operate. Instead, they often apply a backtracking (and heuristically-informed) search process that makes one choice at a time, eliminating large subspaces of potential solutions at a time as soon as a forbidden property can be deduced from the partial solution. The particular tools we use employ a search algorithm called conflict-driven nogood learning (CDNL) that augments the skeleton of backtracking search with an analysis of the underlying reasons for dead-ends, allowing the algorithm to lazily learn additional constraints (called nogoods) [6].

Inside the disjunctive answer set solver we use [4], two CDNL-based solvers, called the "generator" and the "tester," collaborate. The responsibilities of these two internal solvers resemble the roles of Elise (the generator) and Fiona (the tester). Counterexamples found by the tester become nogoods for the generator, allowing the generator to internally reject future candidates that would fail for the same reasons. As a result, a vast number of calls to the tester are eliminated compared to a pipelined architecture.

6.2 The Design Space Model

Before tackling disjunctive modeling, we first sketch our design space model that captures the mechanics and design constraints of *Refraction 2* puzzles.

Our problem encoding is comprised of rules that guess the primary structure of a candidate puzzle-and-solution pair, deduce its properties, and forbid unsuitable candidates. In particular, choice rules guess which pieces are used (on the board vs. left in the tray), their locations, their types, their port configurations (input/output directions), and what power level will be emitted by sources. Several integrity constraints forbid stacking pieces, assigning a direction as both an input and output port, producing benders that do not bend, or leaving a target unpowered. Additional rules deduce the properties used in the integrity constraints from those guessed with choice rules (e.g. deducing the laser flow entering each target given the guessed arrangement and configuration of pieces).

The encoding above (defined in only about 120 source lines of code) describes the game in generic terms (e.g. on a d-dimensional grid). To set up the puzzle design problem for the specific 10-by-10 grid used in the public version of our game with about 40 pieces (many with desired types known a-priori), our problem instance description includes facts committing to these values as well as tabulating allowed fraction values the effect of splitters and combiners on those fractions. In a deployed setting, the problem instance description would be emitted by the game's level progression manager.

Feeding the problem instance and encoding to an answer set solver, we have an automated system for generating solvable puzzle-and-solution pairs. As in previous work [13], we could add many constraints on the form of the reference solution. Here, however, we want to focus on shaping the space of play by defining a sense of shortcuts.

Two small additions the answer set program sketched above prepare us for generating shortcut-free puzzles for *Refraction*. In each of the snippets below, we use a double-underscore naming convention for predicates that will be given special semantics under our program transformation (described in the next subsection). The first addition is a set of rules declaring which parts of the puzzle-and-solution pair define the puzzle. This snippet shows how we tag the port configuration of all pieces and the input power for targets as a level design detail (vs. a solution detail):

```
__level_design(port(P,D,S)) :- port(P,D,S).
__level_design(target_power(P,F)) :- target_power(P,F).
```

The second set constructs a challenge for Fiona. It defines the `Concept` predicate needed to formulate the shortcut-free generation problem (assuming that candidate solutions which would not satisfy the `Solves` predicate have already been forbidden by integrity constraints). Here, we define the concept of actively using at least six bender pieces:

```
active_bender(P) :- active(P), type(P,bender).
__concept :- 6 { active_bender(P) }.
```

Interpreted as a normal answer set program, these additions cause no filtering of the set of puzzle-and-solution pairs. They merely advise the game-independent program translator discussed in the next subsection that transforms this program into one with the desired semantics.

6.3 Transforming the Model

The `metasp`[2] project, emerging from research in modeling complex optimization problems in ASP, provides a powerful toolkit for meta-programming in AnsProlog. By providing a reference implementation of the answer set semantics in AnsProlog itself, it is easy to build programs that modestly extend these semantics.

One of the example applications of `metasp` is in building building a subset minimality (also called inclusion-based minimality) constraint. Although subset minimality does not immediately appear to have anything to do with finding shortcuts in the space of play, there is a way to reduce our problem to one of judging subset minimality. Our method, which was inspired by a crucial tip from one of the `metasp` inventors (Martin Gebser) on how to model conformant planning problems, works by constructing a very specific set of literals for use in the minimization.

The meta-program below (itself a normal answer set program) analyzes the rules in our design space model from the previous subsection and instructs the `metasp` library to filter the space of puzzle-and-solution pairs to those satisfying both the subset minimality problem and demonstrating the required concept in the reference solution. Applying the

[2] `http://www.cs.uni-potsdam.de/wv/metasp/`

program below to any normal answer set program employing `__level_design` and `__concept` rules yields a larger, disjunctive answer set program that is consistent with the intended semantics of shortcut-free puzzle design.

```
% extract atoms in the head of __level_design rules
level_design(A) :- rule(pos(atom(__level_design(A))),_).

% metasp: minimize over the set {A, not A, __concept}
wlist(-1,0,pos(atom(A)),1) :- level_design(A).
wlist(-1,0,neg(atom(A)),1) :- level_design(A).
wlist(-1,0,pos(atom(__concept)),1).
minimize(1,-1).

% metasp: use inclusion-based minimality for that set
optimize(1,1,incl).

% metasp: require that __concept is present
:- not hold(atom(__concept)).
```

We put these seven source lines in a file called `metaC.lp` and stored them with the other meta-programs taken from the `metasp` project. Using this collection as a inline translator, we can sample shortcut-free puzzles from our previously described design space model (`refraction.lp`) with the following command. Clingo and Clasp are state-ot-the-art answer set solving tools from the Potassco[3] project (specifically using a version of Clasp supporting disjunctive rules [4]).

```
$ clingo refraction.lp --reify \
      | clingo - meta{,D,O,C}.lp -l | clasp
```

Why does this work? Suppose Elise has found a puzzle-and-solution pair for which the reference solution demonstrates the concept. The `__level_design` decisions and their negations, along with the fact `__concept`, form a set. This is the set our meta-program checks for subset minimality. If there is some shortcut solution for this puzzle, Fiona can find a second puzzle-and-solution pair that agrees with the Elise's on all `__level_design` choices (and their negations) but does not satisfy the `__concept` definition. Because Fiona's pair demonstrates a feasible subset of Elise's, the subset minimality constraint rejects Elise's candidate. The only puzzle-and-solution pairs for which subset minimality is satisfied are those for which either there are no shortcut solutions (these are the ones we want) or those which do not satisfy `__concept` in the first place. The final line of our meta-program instructs the forbids the latter case. Note that this explanation is independent of the particular search algorithms used by the ASP tools.

Although this transformation is expressible by a very small meta-program, it is far from intuitive. We intend of design automation tool developers to use our small utility as a black-box translator. We give them the ability to express any problem in $\mathsf{NP}^{\mathsf{NP}}$ without straying from the normal guess/deduce/forbid mindset beyond the use of our domain-appropriate `__level_design` and `__concept` predicates.

6.4 Example Results

In this subsection, we give some examples of shortcut-free puzzles for *Refraction 2* generated[4] with different required concepts. Our intent is to demonstrate direct control over a major stumbling block in past manual and automatic puzzle design efforts. Previously, it was simply not possible to directly ask for the kind of puzzles we wanted—we demonstrate a design-critical, qualitative leap over previous puzzle generation techniques.

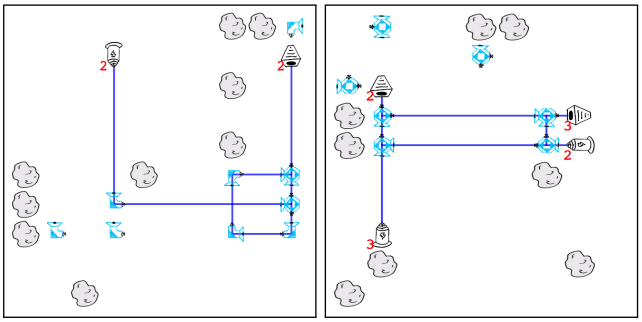

Figure 3: Generated puzzles for the concepts of *using at least one splitter and one combiner* (left) and *using at least two splitters and two combiners* (right). Interestingly, the source and target power requirements do not seem to suggest a need to split or combine, but the need arises from the spatial concerns of port directions. At right, the player must initially split/combine the given beams into beams with power 1 and 4, and then split/combine these to form the beams of power 2 and 3 in the correct directions.

Beginning with an example inspired by Figure 1, we ask the generator to produce levels which always practice using a splitter and combiner together (a typical mid-game challenge). The left side of Figure 3 shows one solution found after 7.26 sec. (2.12 sec. search). In the next example, we require that the player always splits and combines at least twice. The right side of Figure 3 shows a solution found after 18.9 sec. (8.29 sec. search).

We were intrigued by how the previous example involved building a monolithic network to satisfy two source–target pairs that would initially seem to be satisfiable independently. Altering our definition of the required concept to mean building a single network that spans every piece (an unrealistically difficult request designed to stress our tools), we searched for a puzzle that required this property over three source–target pairs and several player pieces. After 152 sec. (140 sec. search), we found the left example shown in Figure 4. This example highlights the generator's mastery of the design for the game's entangled spatial and mathematical mechanics. Where a human designer might use a several blocker pieces to restrict the space of solutions to a size that is feasible to manually check, our tools are capable of designing levels that allow many (but all very complex) solutions without this reasoning crutch. For a clearer example of this phenomena, we asked the generator to find a similar puzzle requiring monolithic solutions without using any blockers at all. This is the source of the right example in Figure 4, which was found in 74.2 sec. (67.2 sec. search).

Our new perspective on the puzzle design problem has al-

[3] `http://potassco.sourceforge.net/`

[4] We report problem construction (single-threaded) and time-to-first-solution search times (with eight threads) using a 2011-era laptop with a 2.2 GHz Intel Core i7 processor.

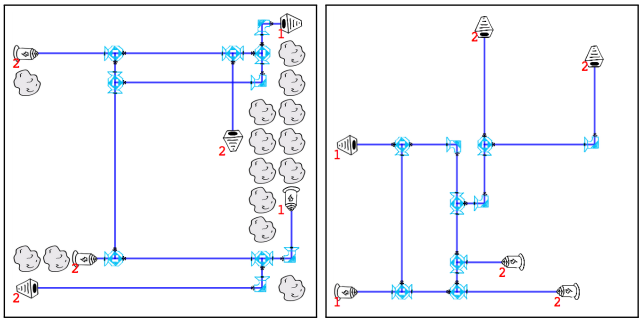

Figure 4: Generated puzzles requiring that the three source–target pairs *can only be satisfied by a monolithic laser network spanning every given piece.* These examples demonstrate the generator's competence in designing intricate puzzles without the need for obvious chokepoints (particularly clear in at right).

lowed us to exactly formulate the key (and previously inexpressible) property sought for most levels: that they *require* the player to practice game mechanics at a depth appropriate to a certain point in the game's level progression. These preliminary results suggest that even concepts that are exceedingly difficult to enforce by hand are readily addressed with off-the-shelf tools. Already, we have begun to learn new puzzle design patterns from the generated outputs.

7. CONCLUSION

In this paper, we have revealed that many game designs are critically sensitive to the presence of undesirable solutions (crystallized in the concept of shortcuts). In response, we have set up a logical foundation for a very broad class of design automation problems. By formulating the shortcut-free generation problem in terms of statements quantified over the space of play, we tap into the availability of state-of-the-art combinatorial search infrastructure. Offering a very small translation utility, we allow designers access to declarative solutions to this entire class of problems ($\mathsf{NP}^{\mathsf{NP}}$) using only two design-appropriate language constructs: `__level_design` and `__concept`. Applying this approach to our own design domain, we report promising feasibility results from building a shortcut-free puzzle generator for *Refraction* puzzles.

In the future, we are intend to integrate player models that can take a candidate (perhaps incomplete) solution as input and report whether the candidate is feasible, likely, or interesting. Using this kind of model, we will express that interesting puzzles should require more subtle properties: perhaps every solution is sufficiently likely for one model while being sufficiently unlikely for another or that the player must place a piece where their model suggests they would tend to do otherwise (requiring concept practice in player-specific contexts).

We encourage others developing *quality*-focused level design tools to account for the distinct responsibilities of the level designer (to synthesize levels with properties like solvability) and the design assistant (to actively seek out and identify potential interactions that might cause the designer to regret her design choices in those levels).

8. ACKNOWLEDGMENTS

This work was supported by the University of Washington Center for Game Science, DARPA grant FA8750-11-2-0102, and the Bill and Melinda Gates Foundation.

9. REFERENCES

[1] D. A. Ashlock. Automatic generation of game elements via evolution. In *Comp. Intel. and Games (CIG), IEEE Conf.*, 2010.

[2] M. Brain and F. Schanda. DIORAMA (warzone map tools), 2009. `http://warzone2100.org.uk/`.

[3] K. Compton, A. M. Smith, and M. Mateas. Anza island: Novel gameplay using ASP. In *FDG Workshop on Procedural Content Generation (PCGames'2012)*, 2012.

[4] C. Drescher, M. Gebser, T. Grote, B. Kaufmann, A. König, M. Ostrowski, and T. Schaub. Conflict-driven disjunctive answer set solving. In *Proc. Intl. Conf. on Principles of Knowledge Representation and Reasoning (KR'08)*, pages 422–432, 2008.

[5] J. Fisher. How to make insane, procedural platformer levels. *Gamasutra*, May 2012. `http://www.gamasutra.com/view/feature/170049/How_to_Make_Insane_Procedural_Platformer_Levels_.php`.

[6] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012.

[7] M. Gebser, R. Kaminski, and T. Schaub. Complex optimization in answer set programming. *Theory and Practice of Logic Programming*, 11(4-5):821–839, 2011.

[8] M. S. Gentry. Anchorhead, 1998. `http://www.ifarchive.org/if-archive/games/zcode/anchor.z8`.

[9] E. J. Hastings, R. K. Guha, and K. O. Stanley. Automatic content generation in the galactic arms race video game. *Comp. Intel. and AI in Games, IEEE Trans.*, 1(4):245–263, 2009.

[10] M. J. Nelson and M. Mateas. Search-based drama management in the interactive fiction anchorhead. In *Artif. Intel. and Interactive Digital Entertainment Conf. (AIIDE)*, 2005.

[11] N. Shaker, G. Yannakakis, and J. Togelius. Towards Automatic Personalized Content Generation for Platform Games. In *Artif. Intel. and Interactive Digital Entertainment Conf. (AIIDE)*, 2010.

[12] A. M. Smith. *Mechanizing Exploratory Game Design*. PhD thesis, University of California, Santa Cruz, 2012.

[13] A. M. Smith, E. Andersen, M. Mateas, and Z. Popović. A case study of expressively constrainable level design automation tools for a puzzle game. In *Intl. Conf. on the Foundations of Digital Games*, 2012.

[14] A. M. Smith and M. Mateas. Variations Forever: Flexibly generating rulesets from a sculptable design space of mini-games. In *Comp. Intel. and Games (CIG), IEEE Conf.*, 2010.

[15] A. M. Smith and M. Mateas. Answer set programming for procedural content generation: A design space approach. *Comp. Intel. and AI in Games, IEEE Trans.*, 3(3):187–200, 2011.

[16] J. Taylor and I. Parberry. Procedural generation of Sokoban levels. Technical Report LARC–2011–01, Laboratory for Recreational Computing, Dept. of Computer Science & Engineering, Univ. of North Texas, February 2011.

[17] D. Yu. The full spelunky on Spelunky, 2011. `http://makegames.tumblr.com/post/4061040007/the-full-spelunky-on-spelunky`.