A Case Study of Expressively Constrainable Level Design Automation Tools for a Puzzle Game

Adam M. Smith[*†], Erik Andersen[†], Michael Mateas[*], Zoran Popović[†]

[*]Center for Games and Playable Media, University of California, Santa Cruz
[†]Center for Game Science, Dept. of Computer Science & Engineering, University of Washington
{amsmith,michaelm}@soe.ucsc.edu {eland,zoran}@cs.washington.edu

ABSTRACT

Some problems in procedural content generation for games involve hard constraints (e.g. that a generated puzzle is necessarily solvable). Common techniques for generator design lack a way to specify crisp (yes/no) constraints on what counts as a valid content artifact and guarantee these constraints are satisfied in the generator's output. In this paper we present two independent implementations of three diverse level design automation tools for the popular online educational game *Refraction*. All of the systems guarantee key properties of their output. Applying a constraint-focused generator design perspective in depth, we found that even emergent aesthetic style properties were straightforward to directly control. Our results with *Refraction* provide further concrete evidence for the claim that the expressive power of constraints and the ease with which they can be incorporated into suitably designed generative processes makes them a powerful tool for producing reliably-controllable generators for game content.

Categories and Subject Descriptors

K.8.0 [**Personal Computing**]: General – Games; I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving – Logic programming; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search – Backtracking

Keywords

procedural content generation, backtracking search, answer set programming, educational games, puzzle games

1. INTRODUCTION

In procedural content generation (PCG) for games, the topic of what guarantees a generator makes about its output often goes unaddressed. When seeking to apply PCG to a future, player-adapting version of the popular online educational game *Refraction*[1], we encountered a strong need for assurances regarding generated content. *Refraction* is a Flash puzzle game in which players arrange devices on a grid to construct networks of laser beams. By requiring the player to construct beams of varying power levels, the game aims to teach mathematical skills, such as proportional reasoning, while exercising spatial problem solving abilities.

With regard to guarantees, we require that puzzles generated for a given player be not just solvable, but solvable under conditions appropriate for that player's progress: with precisely dictated size, complexity, and required mathematical and spatial skills (such as being able to understand fraction simplification or use three left-turns to make a right-turn). Further, we want to prescribe aesthetics of the visual composition to continue the standards of control used in the creation of the game's current hand-authored puzzle sequence. While some of these requirements flow from our game's educational goals, it should be clear that methods for addressing these requirements will have use well outside of the realm of educational games.

Content generators are often designed as either directly constructive processes or generate-and-test systems [15]. Constructive processes guarantee some properties of their outputs *by construction*, however other properties can only be enforced by carefully hand-picking from sampled outputs. Generate-and-test systems attempt to automate this process, however they are often implemented as open-ended optimization processes (such as genetic algorithms) which still require human intervention to decide precisely when generated artifacts are sufficiently fit for use in gameplay. Crisp thresholds (sharp boundaries defining what content is acceptable or not) are not defined in the problem formulation used by these system because picking a single acceptance threshold on artifacts' computed fitness (usually a single scalar value) is difficult to impossible.

Ideally, we would have many examples of generators with crisply defined output spaces to draw from when designing new systems. These example generators should handle the full complexity of well-known games to provide realistic references for familiar problems. Towards generality, they should demonstrate direct control over a wide variety of features of interest (e.g. low-level structural validity, user-specified control parameters, and high-level aesthetic con-

FDG '12, May 90–June 1, 2012, Raleigh, NC, USA.
Copyright (c) 2012 ACM 978-1-4503-1333-9/12/05... $10.00

[1]As of March 2012, *Refraction* had over 490,000 plays at `http://kongregate.com/games/GameScience/refraction`

cerns). Schanda and Brain's Diorama, a highly controllable map generator for *Warzone 2100* (Pumpkin Studios 1999), is one such system in the domain of real-time strategy games, but it has only been briefly reviewed in the literature [12].

In this paper we describe six systems that guarantee key output properties. We consider three diverse level design automation problems in *Refraction*: generation of high-level missions (under educational and gameplay constraints), transforming missions into spatially-realized puzzles (which must be solvable in particular ways), and producing alternative solutions to pre-existing puzzles (allowing us to probe the requirements of our hand-made levels and those generated with different goals). For each problem, we provide two system implementations. Our initial implementations were based on either constructive techniques or familiar complete-search techniques (bounded depth-first search). Exploring recently proposed techniques [12], our subsequent implementations used answer set programming (ASP), a declarative programming paradigm that targets difficult combinatorial search problems with state-of-the-art algorithms.

Because all of our systems are correct by design (in that they always produce content conforming to input requirements upon termination when it is logically possible to do so), we focus our analysis of these systems on their uncontrolled aspects: code size, running times, accidental style features, and authoring affordances. We found that our ASP-based tools often produced example outputs which directly drove refinement of our problem formulations, causing us to better understand the deeper issues of puzzle design in games like *Refraction*. Because we were able to rapidly iterate on the specification of constraints, our later ASP-based tools are significantly more controllable and thus more useful in the face of our design automation problems.

The primary finding of our case study is the unexpected expressive power that resulted from the in-depth application of a constraint-focused generator design perspective. Our results provide further evidence for the claim that declarative languages with first-class constraints such as those available in ASP are powerful tools for producing *expressively constrainable* generators, systems that accept a wide range of hard constraints as part of their input while providing theoretical guarantees for the production of that content if it is feasible. By treating aesthetic failures (e.g. poor compositional balance of a puzzle) as equivalent to gameplay failures (e.g. an unsolvable puzzle), we not only raise the stakes on a question Togelius et al. [15] identify as a major research challenge in search-based PCG (*How can we avoid pathological failures?*), but provide multiple real-world examples as answers.

2. RELATED WORK

In previous research into automatically generating puzzles, there is often a search algorithm in the core of systems that works to separate broken or uninteresting puzzles from those that are well formed and elegant. Colton [3] identified puzzle generation as a creative task, requiring a designer to produce an artifact (the puzzle) that would cause the solver (the player) to make a personal discovery (finding an interesting solution). Much of the search in Colton's system is dedicated to ensuring that the puzzle's intended solution is derivable from the clues provided and that are no simpler solutions.

Focusing specifically on the problem of controlling the complexity of a puzzle's simplest solution, Ashlock [1] demonstrated an evolutionary algorithm for generating Chromatic and Chess mazes (both are spatial puzzles) with preferentially long shortest-path solutions. Oranchak's Shinro puzzle generator [10] also used an evolutionary algorithm. However, instead of optimizing solution length, Oranchak's system optimized a metric that balanced structural validity (which is non-trivial for Shinro, involving global agreement of puzzle clues) with closeness to a set of user-specified parameters that expressed a target number of pieces and solution steps. Though these measures of a puzzle's fitness provided informative evolutionary pressure to guide the search process in the direction of desirable puzzles, they alone do not guarantee eventual generation of suitable puzzles by these systems (an inherent property of metaheuristic optimization techniques [17] that also applies to the feasibility constraints of FI-2Pop [9]).

Gebser's Sudoku puzzle generator [5], by contrast, provides strict theoretical guarantees. This 38 source-line generator, based on answer set programming (described later), defines the structural properties of desired puzzles (including the minimality of clue sets with respect to ensuring a unique solution) and then uses an off-the-shelf answer set solver to deterministically enumerate all (and only) those puzzles with the required properties.

A number of related systems have explored content generation with the application of hard constraints. To our knowledge Diorama[2] is the only one to enforce these constraints through without the need to have defined custom algorithm. Tanagra [13] (a platformer level generator that uses an off-the-shelf numerical constraint solver to enforce reachability for all of a map's platforms), SketchaWorld [11] (a declarative 3D modeling tool for terrains that foregrounds constraints in its user interface), and the layout solver described by Tutenel et al. [16] (rule-based system for arranging building-interior scenes under layout constraints) are highly relevant projects whose applicability to a deployed game remains to be seen.

3. ANSWER SET PROGRAMMING

While a best-first heuristic search algorithm such as A* is likely to be familiar to game developers, the search algorithms[3] underlying some of the core tools in our systems are less so. We have made extensive use of answer set programming (ASP), a logic programming paradigm that borrows syntax from Prolog and search algorithms from solutions to the Boolean satisfiability (SAT) problem [2].

Like regular expressions for string matching or structured query languages (SQL) for retrieval from databases, AnsProlog (the common language for answer set solving systems) is a highly declarative language for solving combinatorial search and optimization problems, not a general-purpose

[2]`http://warzone2100.org.uk/`

[3]The answer set solver we used employs conflict-driven nogood learning (CDNL), a state-of-the-art, complete, backtracking, heuristic search algorithm loosely inspired by the Davis-Putnam algorithm for Boolean satisfiability [7].

programming language such as C++ or Java. Most ASP systems work by translating the programmer-provided problem definition into a low-level, domain-independent representation through the process of grounding (also called instantiation). Then the ground problem is solved by a high-performance combinatorial search algorithm.

One of the goals of ASP is to allow programmers to construct solutions to complex search problems without the need to develop and maintain advanced combinatorial search infrastructure. The imperative details of the answer set solver's underlying algorithm (which are easily reconfigured with command-line settings) are unimportant so long as suitable outputs are produced in an acceptable amount of time.

3.1 ASP for PCG

In a recent journal article [12], we described the general approach of applying ASP to PCG problems, offering a tutorial introduction to answer set programming and a review of four existing applications using the technique.

Regarding the software engineering practices around using ASP for PCG, we noted that properties of artifacts produced during the development of a generator will often inspire changes to the design space definition, motivating the need for flexible generation systems which admit sculpting the space of outputs without an overall redesign of the generator. Some of these changes involve "zooming in" on content exhibiting patterns of interest or rejecting content with easily describable flaws.

In contrast with modern multi-paradigm languages (e.g. Python), the structure of answer set programs is relatively simple. These logic programs contain two constructs: facts and rules. Facts are statements (akin to data literals or documents in a data language like XML) that can be used to describe bulk configuration or the properties of an input problem instance. Three types of rules control the production of new facts. Choice rules specify how to *guess* a description of a candidate solution. Deductive (Prolog-like) rules specify how to *deduce* the properties of a guessed solution. Finally, integrity constraints *forbid* solutions exhibiting or not exhibiting certain deduced properties.

For the purposes of high-level design, the programmer can imagine the answer set solver runs a generate-and-test process, repeatedly *guessing* solution candidates, *deducing* their properties, and then testing if they should be *forbidden*. In actuality, solvers will propagate constraints forwards and backwards through the rules in a non-trivial manner that further includes learning of new constraints (called nogoods) on the fly from dead-ends discovered by the live search process. Further, whole spaces of partial solutions that exhibit forbidden substructures are often eliminated before any are completely assembled.

When building a content generator with ASP, the programmer focuses almost exclusively on how the content design space is declaratively defined, treating the solver as an uninteresting black box.

4. REFRACTION PUZZLE DESIGN

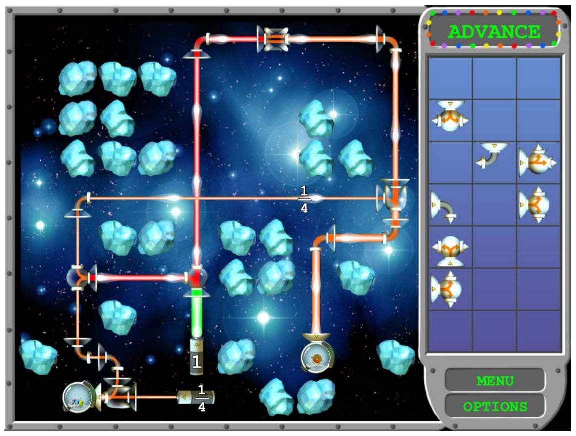

Figure 1: A screenshot of gameplay in *Refraction* depicting a puzzle solution involving benders, splitters, combiners, and an expander.

The premise of *Refraction* is that the player must arrange devices to form a network of laser beams that will restore power to animals stranded in underpowered spaceships. The power of laser sources and the power required by targets are mismatched, so the player must split and recombine beams to provide power in the correct proportion, indicated by a fraction. Figure 1 shows an example puzzle and solution.

When play begins, the 10-by-10 grid is clear except for laser sources, laser targets (animals in spaceships), and blockers (asteroids or other space debris). The position and orientation of these pieces are fixed. Additionally, sources and targets are annotated with the fractional power that they emit or require to be satisfied.

All player-movable pieces (beam manipulating devices with a fixed rotation) start in the panel on the right. There are four broad piece categories. Benders simply apply a 90-degree deflection to a beam without changing power. Splitters consume one input beam and produce two beams at one half of the input power (or three beams at one-third power depending on the number of outputs on the splitter). Combiners (which come in two-input and three-input varieties) produce an output beam with a power that is the sum of all of the input beams (but only if all inputs are filled and the input fractions share the same denominator). Expanders (which do not deflect) facilitate combining of unlike fractions by multiplying the numerator and denominator by a common factor. Expanders are available with factors of 2, 3, and 5 such that applying a 3-expander to the fraction 1/2 results in the (unreduced) fraction 3/6.

Not all of the pieces provided to a player are necessary to form a solution. Most puzzles will include extra pieces that are intended to distract the player. Distractors are not always useless because they may be used to construct alternate (often more elaborate than necessary) solutions.

The designer's challenge is to produce a progression of puzzles that incrementally introduces the player to the spatial and mathematical reasoning challenges of the game and eventually prepares them for the game's full complexity, requiring fluent use of many types of pieces simultaneously. In the context of this progression, it is clear that what makes a level acceptable (as the product of generation) depends far more on its relevance to the player's progress than any simple measure of its solution length or the like.

The bare-minimum challenge for deployable level design automation in *Refraction* is to produce a generator that can recreate levels in the style and complexity of each point in the current, hand-authored linear progression. Even this requires generators with highly controllable output sufficient to express what makes a puzzle appropriate for the very beginning, the very end, or, say, the first level that introduces expanders. Beyond this, we are interested in enabling nonlinear, player-specific concept and difficulty progressions.

5. PROBLEM FORMALIZATION

To make the challenge of level design automation for *Refraction* more concrete, we have broken it down into three artifact generation problems. To structure our puzzle generator, we have adopted Dormans and Bakkes' [4] distinction between missions and spaces. A mission is a logical order of the goals a player must accomplish to complete the level, and a space is the actual physical layout of the level. Our first two problems are concerned with producing missions for *Refraction* and subsequently embedding those missions in a puzzle grid. The final problem is concerned with seeking alternative solutions to existing puzzle designs, regardless of the mission for which it was originally designed.

5.1 Mission Generation

The intent of the mission generation problem is to capture the high-level design concerns of a *Refraction* puzzle: Which pieces are active? How big is the imagined solution? What level of mathematical knowledge will be involved? Because fractions are integral to the game's educational goals, mission generation includes working out which fractions should be constructed and how the construction might proceed.

The primary input to our mission generators is a set of mathematical expressions that the player should construct during play. The set $\{(1/2) + (1/4), (1/4) + (1/4), (((1/2)/2)/2)\}$ suggests the need for adding twice (once with the use of an expander to build a common denominator) and repeated splitting by half. The mission generator is also given a target number of blockers (24), benders (7), and distractor pieces (7) to modulate difficulty. These were the inputs to the mission generation process that eventually resulted in the puzzle and solution shown in Figure 1.

In the ASP-based mission generator, an upper bound and optional lower bound on piece counts are specified for all piece types, along with style constraints affecting the presence or absence of arbitrary mission subgraphs. These constraints are not expressible with the initial mission generator (described later) without a major redesign.

The output of mission generation is an annotated directed acyclic graph (DAG) where there is a node for every piece in the imagined puzzle and an edge for every solution-critical laser beam connecting pieces. Nodes describe a piece's mathematical type (such as 2-splitter or 5-expander) but not its spatial type (such as having an input from the west and an output to the north). Source and target nodes are labeled with the fraction power they emit or require. Figure 2 shows an example mission satisfying the constrains above.

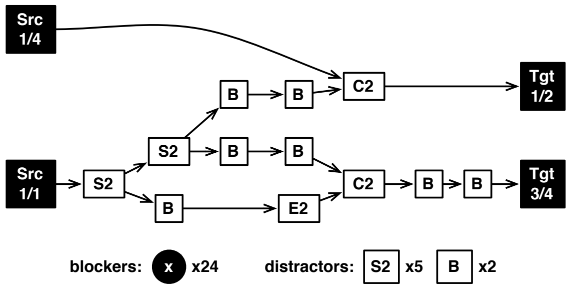

Figure 2: A mission DAG containing several 2-splitters (S2), benders (B), 2-combiners (C2), and a 2-expander (E2). Several blockers (x) and distracting pieces will also be present in any spatial realization of this mission.

5.2 Grid Embedding

In the grid-embedding problem, the intent is to realize a puzzle with sufficient detail to be played in the live game. That is, embedding resolves the spatial concerns ignored in the mission generation problem. While the current version of *Refraction* is played on a discrete, rectilinear, two-dimensional grid, a version for play on, say, continuous spaces, hex maps, or three-dimensional grids would not disrupt our high-level problem formulation or solution methods.

The primary input to the embedding problem is the same as the output of the mission generation problem. These mission DAGs may come from either of our mission generation systems, extraction from the hand-designed levels, or original human authoring effort. The ASP-based embedder also accepts additional style constraints describing spatial properties such as symmetry, compositional balance, and piece spacing.

The output of grid embedding is the (x/y) position and (north/south/east/west) input/output port configuration of all pieces such that one example solution is constructed that realizes the input mission DAG. Figure 3 shows an alternate embedding, to the one depicted in Figure 1, for the mission shown in Figure 2. Generally, there may be astronomically many valid embeddings of a mission, but we are concerned with producing only a single one.

5.3 Puzzle Solving

Finally, the intent of the puzzle-solving problem is to simply construct alternative reference solutions (at the spatial grid level). In addition to revealing which pieces and patterns are required in a solutions to a puzzle, a fully automated puzzle solver can be used to provide feedback to players by telling them if their solution-under-construction can be extended to a complete solution without removing any piece already placed. This same type of partial solution feasibility checking can be used in offline analysis of recorded game data to track how much time different players spend in infeasible regions of the game's state space.

The input to the puzzle-solving problem is a complete def-

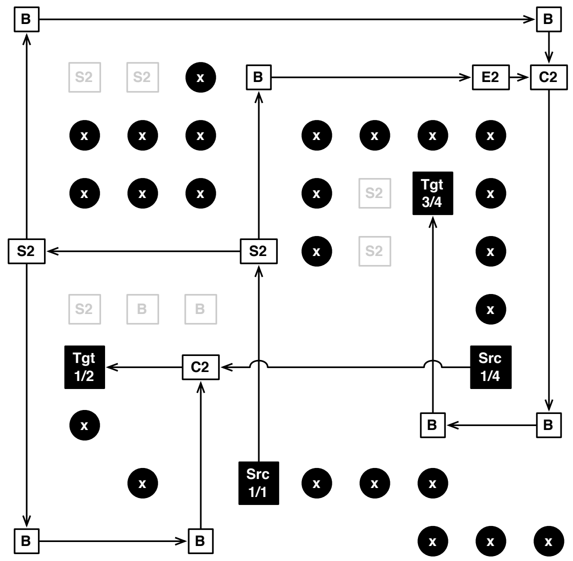

Figure 3: An embedding of the mission DAG from Figure 2 into *Refraction*'s 10-by-10 spatial grid under no special style constraints.

inition of available pieces and their mathematical and spatial configurations (excepting the positions of player-placed pieces, of course). The ASP-based puzzle solver also takes additional style constraints as input: requirements to use or avoid a certain piece or grid cell, to construct or not construct certain sub-networks of laser flow, etc.

The output of puzzle solving is simply the position of every piece such that the resulting configuration satisfies all laser targets or an assertion that the puzzle has no solution of the required style.

6. SYSTEM DESCRIPTIONS

In this section we describe the two implementations of each of our three level design automation tools. We review them in the order they were developed to help convey the idea that each was a legitimate best-effort research solution to the stated problems given the knowledge at hand. Each system was created with the intent of use in a production setting, not specifically for the purposes of comparison.

6.1 Feed-Forward Mission Generation

Addressing the first problem, that of generating high-level missions, our initial implementation adopted a constructive approach consisting of a seven-step pipeline:

1. **Expression Translation**: Mission graph fragments corresponding to the required input expression trees were generated though straightforward compiler techniques. For example, $\{(1/4)+(1/4)\}$ becomes a three-node graph with one 2-combiner node (to represent the "+") linked by edges labeled 1/4 from two untyped nodes.
2. **Opportunistic Combination**: In a randomized fashion, nodes are unified so that the output of one required expression could be the input to another. This process proceeds, avoiding cycle creation, until exhaustion.
3. **Target Completion**: To motivate (though admittedly not to guarantee) the player to construct the imagined graph so far, target nodes are added to consume all laser outputs that are not consumed by another piece already.
4. **Expander Insertion**: Expanders are inserted so that the inputs to combiners all have the same denominator.
5. **Bender Insertion**: The requested number of bender pieces are randomly inserted into the graph on paths between sources and targets.
6. **Distractor Selection**: A number of randomly typed pieces are also added to the graph without adding edges.
7. **Obstacle Insertion**: Similarly to distractors, the required number of disconnected blocker pieces are added.

By construction, generated mission DAGs will describe feasible solutions (at least at the network level) that involve the required mathematical construction and the requested number of blockers, benders, and distractors. This system serves as an example how to guarantee certain properties of outputs through bespoke algorithm design. Note, however, that the above algorithm is carefully adapted to just those design requirements known at the time of its creation.

6.2 Grid Embedding with DFS

The problem of grid embedding immediately appeared to us as a highly constrained search problem (unlikely to be fruitfully addressed with feed-forward or simple generate-and-test approaches). While the problem somewhat resembles the place-and-route problem from electronic design automation (EDA) [14], the particular mechanics of *Refraction* made a hand-rolled complete-search implementation seem the most approachable solution at the time.

Our randomized depth-first search (DFS[4]) algorithm was configured as follows:

- **States**: list of embedded pieces and their positions; graph of remaining pieces to be embedded; list of outgoing beams with their directions
- **Successor Function**: place a piece with no un-embedded inputs from the mission somewhere along the beams to which it must connect and assign its input directions as necessary; if the piece has output directions, then assign them randomly at this time according to the piece's type (ensuring benders deflect the laser, etc.)
- **Goal**: no pieces remain to be embedded

When our DFS terminates at a goal state, that state necessarily represents a valid embedding of the mission DAG with respect to *Refraction*'s rules. While this implementation is sufficient to solve the problem, we later back-ported the use of a geometric restart policy (a common technique for boosting combinatorial search [8]) from our ASP-based embedder, resulting in observed performance improvements of up to four orders of magnitude for realistic problems.

[4]Note that DFS is complete for search spaces with bounded diameter. In the grid-embedding problem, no paths have a length that exceeds the total number of pieces in a puzzle.

6.3 Puzzle Solving with DFS

Based on the initial success with DFS as an implementation strategy for complete and correct embedding, we decided to address the problem of producing reference solutions with this algorithm as well. Puzzle solving involves a similar spatial search to the embedding problem. However, in embedding, a plausible solution graph (the mission DAG with fully-resolved mathematical concerns) is given and the piece input/output port configurations are flexible (to be generated). In solving, port configurations are constrained as part of the input and no solution sketch is provided (dramatically complicating the problem).

Our DFS for puzzle solving was configured as follows:

- **States**: list of pieces placed so far and their positions; list of outgoing beams with their direction and power
- **Successor Function**: select an unused piece that has an input port that can accept an existing, unconsumed outgoing beam and place it somewhere along that beam; decide if the new piece will produce new beams, and compute their direction and power
- **Goal**: the simplified sum of beam powers entering every target matches its required value

As before, correctness with respect to input requirements on successful termination is assured by well-known results for DFS. The geometric restart policy was also back-ported to the DFS-based puzzle solver after our experiments with ASP, yielding solutions for previously intractable puzzles.

6.4 Grid Embedding with ASP

Having assembled and tested the previous three systems and integrated them into a research version of the game, we identified the existing grid embedding system as the biggest bottleneck for runtime performance and expressive control in our plans for a player-adapting revision to the game. Seeking to replace this system with a simpler (towards better adaptability) and potentially faster implementation, we adopted the following organization for our ASP-based grid embedding system.

- **Choice Rules**:
  - Guess absolute (x/y) positions for pieces.
  - Guess port configurations based on piece type.
- **Deductive Rules**:
  - Deduce relative (north/south/east/west) positions from absolute positions.
  - Deduce free paths from relative positions.
  - Deduce realization of beams (embedding for mission edges) from paths and guessed port configurations.
  - Deduce presence of style patterns (compositional balance, symmetry of blockers, etc.).
- **Integrity Constraints**:
  - Forbid two pieces overlapping.
  - Forbid lack of embedding for mission edges.
  - Forbid illegal port configurations (benders must bend, expanders must not, etc.).
  - Forbid violation of style policy (reject if balance or symmetry not detected, etc.).

Correct enumeration of all and only those embeddings that are valid is assured by the correctness of the answer set solver. The source code for our ASP-based embedder did not contain any descriptions of search algorithms, only a declarative description of the search space, artifact analysis methods, and goal conditions.

6.5 Mission Generation with ASP

Success in using the ASP-based embedder as a drop-in replacement for the previous embedder was enticing, so we next looked at replacing the mission generator with an ASP-based variant as well.

Thus far, the mission generator and grid embedder had been run with an overall generate-and-test architecture (because some mission DAGs are formally impossible to embed, such as those containing triangular undirected cycles). ASP held promise for the ability to run the mission generator and grid embedder as an integrated whole under constraints that jointly bound both phases of generation. With the goal of upgrading our embedder into a complete puzzle generator, our ASP-based mission generator was designed as follows.

- **Choice Rules**:
  - Guess which pieces will be active.
  - Guess power level for laser sources.
  - Guess presence of edges between pieces.
- **Deductive Rules**:
  - Deduce a piece's emitted laser power from the power of all edges into it (using a recursive definition).
  - Deduce simplified power for all targets.
  - Deduce set of pieces that are upstream of active targets.
  - Deduce the presence of mathematical or other graph properties ("half_plus_quarter", "triple_bending", etc.).
- **Integrity Constraints**:
  - Forbid directed edges above port limits (only one edge into a splitter, only one edge out of a combiner, etc.).
  - Forbid edges to nodes not on a path to a target.
  - Forbid presence or absence of particular mathematical and style patterns.

To simplify the AnsProlog definition of mission generation logic, we used an auxiliary answer set program to precompute a table of all ways of manipulating beams of different powers. Parts of this program (notably Euclid's algorithm used in fraction simplification) were expressed in Lua, a scripting language made available for performing arbitrary transformations of logical terms with imperative code.

The final ASP-based mission generator can be run standalone, as a drop-in replacement for the previous mission generator, or it can be combined with the ASP-based grid embedder (by simply concatenating the source for the two programs) to form a monolithic puzzle generator.

6.6 Puzzle Solving with ASP

With the accumulated logical modeling experience of producing the mission generator and grid embedder, creating a styleable puzzle-solver using ASP was straightforward.

- **Choice Rules**:

– Guess piece positions (the player's only responsibility).

- **Deductive Rules**:

– Deduce relative positions from piece positions.
– Deduce free paths from relative positions.
– Deduce beam flow from paths and port configurations.
– Deduce emission fractions from beam flow (recursively).
– Deduce target power from beam entrance.
– Deduce presence of solution-style patterns.

- **Integrity Constraints**:

– Forbid two pieces overlapping.
– Forbid leaving targets unpowered.
– Forbid incorrectly powering targets.
– Forbid violation of style policy.

In the puzzle solver, a piece's effect on fractions was again expressed in Lua. However, no table was pre-computed because, when pieces are fully specified, a much smaller and puzzle-specific space of fractions is encountered.

In addition to correctly reporting whether a puzzle is solvable under stylistic restrictions (yes/no), this puzzle solver is radically reusable for online and offline analysis of partial solutions and queries as to whether a particular piece type or placement is essential to solving a puzzle (regardless of the originally imagined mission for that puzzle).

7. ANALYSIS

Our comparative analysis of the two sets of level design automation tools breaks down into a quantitative comparison of the software systems and a qualitative comparison of the inputs and outputs for each tool.

7.1 Quantitative Comparisons

When comparing our respective generator implementations side-by-side, the most apparent difference is in their language distribution and code size. The original tools consist of a moderate amount of Java code whereas the newer tools consist of a smaller amount of code in AnsProlog and Lua. For the three tools, here are the code size[5] distributions:

- **Mission Generation**:
  1,145 Java lines — 194 AnsProlog, 38 Lua lines
- **Grid Embedding**:
  987 Java lines — 75 AnsProlog, no Lua lines
- **Puzzle Solving**:
  988 Java lines — 83 AnsProlog, 61 Lua lines

In another numerical comparison, we looked at the running time of the tools on a fixed set of inputs derived from the example shown in Figure 1, a high-complexity puzzle in the class of late-game challenges that involve several pieces from every major piece category. Configuring the tools as equivalently as possible (applying no style constraints for the ASP-based tools), we averaged times[6] for 1,000 runs with different random seeds.

- **Mission Generation**:
  Feed-forward algorithm: $<$ 1 ms. total
  ASP: $<$ 1 ms. search (530 ms. grounding/preprocessing)
- **Grid Embedding**:
  DFS: 650 ms. search (negligible overhead)
  ASP: 110 ms. search (630 ms. grounding/preprocessing)
- **Puzzle Solving**:
  DFS: did not find solution within 1-hour timeouts
  ASP: 350 ms. search (340 ms. grounding/preprocessing)

Generally, for difficult search problems, the advanced search algorithms of the answer set solver (Clingo 3.0.3 [6]) are better suited than the hand-rolled search in the original tools (dramatically more so before the post-hoc addition of a geometric restart policy). In all cases, the ASP-based solutions spend a significant amount of time on non-search activities (predominantly in propositional grounding). This grounding cost need only be paid when input requirements change if the grounded program is cached. Although the time required for grounding grows with problem size (our current encodings are cubic in player-controlled piece count), the fact that *Refraction* is played on a *constant*-bounded scale (with no more than a handful of player-controlled pieces) means this growth is a theoretical curiosity so long as results are swiftly found at the scale of interest. That the solver's worst case running time is bounded only by an exponential in the size of the grounded problem is similarly uninteresting for realistic problems.

7.2 Qualitative Comparisons

Resulting from the *target completion* phase used in the constructive mission generator's pipeline, every mission DAG generated by this implementation describes a puzzle solution that does not involve laser wasting (the situation where a beam emitted by one of a piece's outputs goes unused in the solution). While (in concert with distractor pieces) players are often capable of wasting lasers if they choose, this style quirk of the original mission generator is an interesting secondary effect of attempting to motivate players to construct a particular network. Knowing of the laser wasting aversion in the original generator, the ASP-based mission generator intentionally includes hooks for requiring or forbidding the presence of laser wasting at the mission level. Similarly, the greedy nature of the *opportunistic combination* phase meant that mathematical expressions would only be realized in a subset of all feasible ways, prompting the subsequent development of arbitrary mission subgraph constraints which could control how expressions were realized more generally.

In the ASP-based embedder, we found that running the answer set solver with a fast-but-simplistic heuristic often led to embeddings that compacted all of a puzzles pieces into a cluster near the one corner of the grid. Crudely addressing this concern by telling the answer set solver to use more randomness in its search or to use a different heuristic was not a reliable solution. While the compacted embedding was in strict conformance with the definition of the embedding problem, the result was aesthetically unacceptable. Analysis of these compacted solutions prompted us to define a basic model of compositional balance: after deducing active hemispheres (e.g. "laser flows through a piece in the west half of puzzle"), we forbid configurations that leave any hemisphere

[5]We counted (non-blank, non-comment) source lines. These line counts are intended to record all code needed to support each tool assuming the others already existed (thus, shared utilities such as parsers and printers for the XML level file format are not counted).

[6]Experiments were performed on a 2006-era ("Dempsey") Intel Xeon CPU at 3.0 GHz. DFS times record search-time for the implementation *with restarts*.

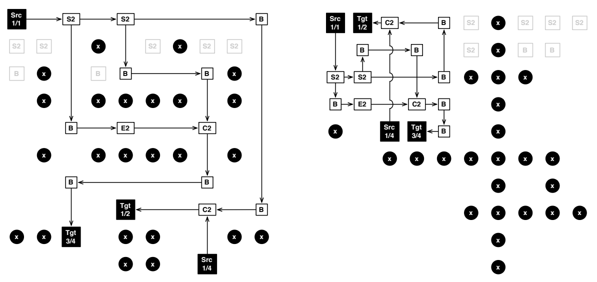

Figure 4: Two oppositely styled embeddings: the left exhibits compositional balance, blocker symmetry, beam spacing, and a lack of crossings while the right exhibits forced imbalance on both axes, asymmetry, piece abutting and multiple beam crossings.

inactive. The pattern of "abutting" (where laser flows between two immediately adjacent pieces without revealing the beam) could also be controlled to produce cleanly spaced reference solutions (meaning that players, particularly novices, would not be required to abut their pieces to complete the puzzle). Finally, the pattern of "crossing the beams" represented yet another feature of exactly-controllable output style in the ASP-based embedder. Figure 4 demonstrates driving the embedder in opposite stylistic directions.

When no style constraints are provided for the ASP-based embedder, the space of embeddings it might generate is identical to that of the DFS-based embedder running on the same inputs. Adding style constraints results in a generative space that is a strict subset of the unconstrained space. A similar situation applies to adding style constraints to the ASP-based puzzle solver.

While the original puzzle solver was intended primarily to produce a simple (yes/no) answer, the inclusion of style constraints in the input to our other tools naturally prompted our brainstorming of the potential alternate uses of an expressively constrainable puzzle solver mentioned previously. The ability to attach novel constrains to the input of our ASP-based tools, even when the existing definitions were not designed with these constraints in mind, represents a major qualitative difference between our two sets of tool implementations owing to ASP's architectural affordances.

7.3 Conclusion

We have described six examples of level design automation systems that make hard guarantees on key properties of their output. Covering three diverse level design automation challenges for *Refraction*, we have demonstrated that such guarantees can be made for the full complexity of a popular online game (further, one that was not designed around future design automation). In achieving this for an initial set of constraints, we made use of a constructive generator (which guarantees properties of its output by careful construction) and a familiar complete-search algorithm (DFS in a bounded space). To quickly produce generators for a wider variety of output guarantees, we explored emerging PCG results suggesting the use of answer set programming to declaratively capture exactly the design space we required.

Our results suggest the developers of procedural content generators should not shy away from working with hard constraints. By applying a constraint-focused generator design perspective in depth, it is possible to not only produce reliably controlled generators with attractive performance measures, but to also come to better understand design automation problems through iterative exploration of constraints and generated output

8. ACKNOWLEDGMENTS

This work was supported by the University of Washington Center for Game Science, DARPA grant FA8750-11-2-0102, the Bill and Melinda Gates Foundation, NSF grants IIS-0811902 and IIS-1048385, and a NSF Graduate Fellowship.

9. REFERENCES

[1] D. Ashlock. Automatic generation of game elements via evolution. In *IEEE Symposium on Comp. Intel. and Games (CIG'10)*, pages 289–296, 2010.

[2] C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving.* Cambridge Univ. Press, 2003.

[3] S. Colton. Automated puzzle generation. In *Proc. of the AISB'02 Symposium on AI and Creativity in the Arts and Sciences*, 2002.

[4] J. Dormans and S. Bakkes. Generating missions and spaces for adaptable play experiences. *IEEE Trans. on Comp. Intel. and AI in Games*, 3(3):216–228, 2011.

[5] M. Gebser. gen_sudoku.gringo, 2010. `http://asparagus.cs.uni-potsdam.de/encoding/show/id/12739`.

[6] M. Gebser, B. Kaufmann, R. Kaminski, M. Ostrowski, T. Schaub, and M. T. Schneider. Potassco: The Potsdam answer set solving collection. *AI Commun.*, 24(2):107–124, 2011.

[7] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-driven answer set solving. In *Proc. of the 20th Intl. Joint Conf. on Artif. Intel.*, pages 386–392, 2007.

[8] C. P. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *Proc. of the 15th Natl. Conf. on Artif. Intel.*, pages 431–437, 1998.

[9] S. O. Kimbrough, G. J. Koehler, M. Lu, and D. H. Wood. On a feasible-infeasible two-population (FI-2Pop) genetic algorithm for constrained optimization: Distance tracing and no free lunch. *European Journal of Operational Research*, 190(2):310–327, October 2008.

[10] D. Oranchak. Evolutionary algorithm for generation of entertaining Shinro logic puzzles. In *EvoApplications 2010*, pages 181–190, 2010.

[11] R. Smelik. *A Declarative Approach to Procedural Generation of Virtual Worlds.* PhD thesis, TU Delft, 2011.

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

[13] G. Smith, J. Whitehead, and M. Mateas. Tanagra: Reactive planning and constraint solving for mixed-initiative level design. *IEEE Trans. on Comp. Intel. and AI in Games*, 3(3):201–215, 2011.

[14] P. Suaris and G. Kedem. An algorithm for quadrisection and its application to standard cell placement. *IEEE Trans. on Circuits and Systems*, 35(3):294 –303, 1988.

[15] J. Togelius, G. Yannakakis, K. Stanley, and C. Browne. Search-based procedural content generation: A taxonomy and survey. *IEEE Trans. on Comp. Intel. and AI in Games*, 3(3):172–186, 2011.

[16] T. Tutenel, R. Bidarra, R. M. Smelik, and K. J. de Kreker. Rule-based layout solving and its application to procedural interior generation. In *Proc. of the CASA Workshop on 3D Advanced Media in Gaming and Simulation*, 2009.

[17] X.-S. Yang. *Nature-Inspired Metaheuristic Algorithms.* Luniver Press, 2008.