Probabilistic Logic Programming Semantics for Procedural Content Generation

Abdelrahman Madkour, Chris Martens, Steven Holtzen, Casper Harteveld and Stacy Marsella

Northeastern University

{madkour.a, c.martens, s.holtzen, c.harteveld, s.marsella}@northeastern.edu

Abstract

Research in procedural content generation (PCG) has recently heralded two major methodologies: machine learning (PCGML) and declarative programming. The former shows promise by automating the specification of quality criteria through latent patterns in data, while the latter offers significant advantages for authorial control. In this paper, we propose the use of probabilistic logic as a unifying framework that combines the benefits of both methodologies. We propose a Bayesian formalization of content generators as probability distributions and show how common PCG tasks map naturally to operations on the distribution. Further, through a series of experiments with maze generation, we demonstrate how probabilistic logic semantics allows us to leverage the authorial control of declarative programming and the flexibility of learning from data.

Introduction

Over the past two decades of procedural content generation (PCG) research, two key authorship problems have emerged: on the one hand, when the author of a generative system has only a vague notion of their general intent, but a plethora of examples (e.g., level data for existing game levels, whose properties they want to replicate in the generator), a data-driven approach (cf. PCGML (Summerville et al. 2017)) can lighten the authorial burden of describing criteria for generation in formal terms. On the other hand, if the author's goal is to generate content for which few examples exist (or that deviates from existing example data), but the author has a clear sense of constraints and generation criteria, they may reach for a tool that is based on logic programming or rewrite grammars. The latter emphasizes human authorial control over specifying criteria for generated content, and the former, automatic inference of those criteria via statistical patterns in data. Across this spectrum, there are many formulations of PCG in the literature (Yannakakis and Togelius 2018). Each formulation stipulates the content generation process based on the needs of the underlying computational framework, e.g., "PCG as search" in the case of evolutionary algorithms (Togelius et al. 2011).

In practice, however, PCG developers face both of these authorship needs—direct modeling and inference—in tandem, though potentially at different points in the development process. In this paper, we propose using *Probabilistic Logic* as a general framework for PCG in game design that unifies these two needs.

Our central contributions are (1) a formal notion of **content generators as probability distributions**, and (2) an analysis of an existing probabilistic logic programming language, Problog, in terms of its suitability for PCG authoring tasks. Our analysis entails a small benchmark suite we devised based on maze generation, over which we run several experiments corresponding to common tasks in an iterative PCG development workflow. We conclude with a discussion of how well Problog fits the needs of PCG and what can be improved in this context.

Why Probabilistic Logic?

To be able to allow designers to come up with complex generators that remain controllable and understandable, we argue that we need a solid framework for dealing with randomness. In every formulation of PCG, we define a possibility space and draw instances from it with some level of randomness. The degree to which this randomness is specified is often implicit in the techniques themselves; e.g., search-based techniques search stochastically and prune based on some fitness criteria. The effect of doing so is obfuscation of the way randomness is controlled to other aspects of the generator specification, which in turn makes it difficult to reason about the generator. One must have a solid understanding of the underlying generator to aptly control its randomness, and even then the association is frequently only implicit. Diversity is an important component of many PCG systems, and though there are many means by which one can enforce diversity, they all entail a form of selecting an artifact with some form of uncertainty. Otherwise, the same artifacts will be generated repeatedly. In our view, randomness is important in understanding and specifying content generators that strive for diverse content. A main challenge in developing tools for PCG is empowering designers to define and control this randomness.

Bayesian reasoning affords us with mechanisms we require to do accomplish our goal. It lies at the bedrock of techniques that formally reason about uncertainty (Murphy 2012). In PCG via machine learning, researchers formalize content generation in terms of a latent distribution over the content and assume that one can learn it from data (Summerville et al. 2017). Some formalize it in Bayesian terms and doing so explicitly captures their uncertainty, and permits us to reason about and control their distributions via Bayesian conditioning. However, the lack of high-quality data makes learning the generative design space purely from data challenging (Liu et al. 2020). This constitutes the need for the specification of priors that restrict the learning space by incorporating domain expertise in either the structure of the model or its parameters. Defining this structure via probabilistic graphical models does not directly capture the nuances of game design and requires a great deal of domain expertise in both game design and Bayesian modeling (Summerville and Mateas 2015). Thus, there is a need for a system that provides authorial control to designers that is more expressive than graphical models.

Approach

We adopt a programming-language perspective; that is, we view the specification of this system as a language design problem. Therefore, we leverage existing work on probabilistic programming languages (PPLs), a tractable means by which non-AI experts can model complex phenomena. PPLs have explicit Bayesian semantics and thus allow for sophisticated Bayesian modeling. One can leverage ideas and formalizations common to analyzing programming languages to define a language that allows the separation of modeling the domain from reasoning about it(inference). Thus, designers need not worry about how Bayesian reasoning takes place and just focus on modeling the game design domain. PPLs provide a rich expressive language that can specify a wide range of probabilistic models that previous approaches, such as probabilistic graphical models, cannot. Though young, the field has garnered interest in many domains, especially those that require transparent and explainable AI models. As far as we are aware, we are the first to explore PPLs' use in game design.

There are many approaches to probabilistic programming. In this work, we limit ourselves to ones whose semantics have emerged from the domain of logic programming. Probabilistic logic programming allows us to define language semantics that can specify these models in a high-level language and allow us to learn from examples. They additionally allow us to incorporate constraints that are easier to express in declarative logic statements. The use of declarative programming in the form of Answer Set Programming (ASP) is widespread in PCG research and has proven fruitful as a means by which designers can incorporate their intent (Smith and Mateas 2011). Its ability to facilitate exploration of the generative space through observing facts facilitates a feedback loop that is essential in game design. However, it is difficult to control its stochasticity and thus the shape of its generative space. Designers have limited control over how often each of the possible outputs shows up. In contrast, formulating generators in Bayesian terms allows us to reason about the probabilities explicitly and control how often each output is sampled. This in turn gives designers another knob to control that can be tuned by training, the data for which can come in the form of samples that pass a certain metric or through examples selected by the designer. In essence, this allows us to utilize both domain knowledge and to learn from data; utilizing authorial intent for the former and example artifacts for the latter.

Background

Declarative Generation in Games

Smith and Mateas 2011 first proposed using ASP for designing content generators for games. They argue that ASP allows designers to formulate design problems in a declarative manner and explore the design space via observation of facts. Since then, researchers have explored using ASP in the generation of a variety of different artifacts. Smith and Bryson 2014 demonstrated how dungeon generators can be specified in ASP. Inspired by Dormans and Bakkes (2011)'s work on separating dungeon generation into multiple generation tasks, Smith, Padget, and Vidler (2018) showed how ASP can be used for specifying a high-level graphical representation of levels. Abuzuraiq, Ferguson, and Pasquier 2019 take this a step further and use a graph partitioning algorithm implemented in ASP that converts this graph representation of levels into tile-map levels. Dabral and Martens 2020 presented a narrative planner using ASP where the ability to observe certain narrative events allows the exploration of narrative spaces.

All of these systems built on top of ASP do not specify an explicit parameterization on randomness. This lack of explicit randomness is a challenge due to the issue of deterministic outcomes: the implicit randomness in ASP is only dictated by which model the underlying solver selects first. Although in ASP which stable model gets selected can be altered through optimization functions, ASP solvers do not allow for specific alterations to the underlying distribution and do not curtail the tendency to find the same stable model repeatedly.

PCG via Machine Learning

Procedural content generation via machine learning (PCGML) defines the generation process explicitly in terms of probabilities (Summerville et al. 2017). PCGML models attempt to learn a generative distribution from data to explore and generate artifacts from the generative space. Deep learning models, most commonly a variant of Generative Adversarial Networks (GANs) or Variational Autoencoders (VAEs), are often employed to learn this distribution (Liu et al. 2020). These deep learning models can then be sampled from to support interesting exploration of the latent space of the data such as blending levels from datasets that include levels from multiple games (Snodgrass and Sarkar 2020). However, these models struggle with consistently generating playable levels. Playability often requires hard constraints, e.g., a traversable path must exist from the beginning of the level to the end, which requires strict enforcement. Deep learning models struggle with inference that forces hard constraints to be met (Liu et al. 2020; Xu et al. 2018). Models that attempt to address this by taking into account some hard constraints only steer away the learned distribution from invalid artifacts. These methods do not guarantee excluding unplayable levels (Liello et al. 2020). Thus, today's deep learning for content generation strategies is dependent on post-generation checks for playability and hard constraints.

Probabilistic graphical models can encode hard-constraints (Koller and Friedman 2009), and were also considered as techniques for PCGML (Summerville et al. 2017; Summerville and Mateas 2015). However, using probabilistic graphical models tends to result in domain-specific generators. They require a great deal of modeling expertise to develop and thus difficult for a non-expert to design. This is in part due to the limited expressiveness of graphical models as a modeling language. It is difficult to encapsulate the nuances of design knowledge in such a representation. Developing high-quality graphical models that are true to expert intent and knowledge requires a great deal of understanding of the underlying mechanisms of how graphical models capture the distributions. One can liken developing generators using graphical models to programming in machine language: it is possible but requires a great deal of technical knowledge of the semantics of the representation. For such models to be more widespread, a more expressive language that allows for more abstract descriptions of the problem is required; akin to high-level programming languages such as Python and Rust.

Probabilistic Logic Programming

We can loosely classify systems that combine logic programming with probability theory into two categories: those that follow Sato 1995's distribution semantics, and those that follow Knowledge Base Model Construction (KBMC) (Riguzzi 2018). In systems that follow distribution semantics, probabilistic logic programs define a distribution over possible worlds in the space of normal logic programs. KBMC systems on the other hand are means by which a user can encode a large probabilistic graphical model. We consider systems that follow the distribution semantics since they are most amenable to circuit compilation. Circuits are a powerful data structures that encode Boolean formula (Darwiche and Marquis 2002). Compilation to circuits can amortize the cost of inference and sampling and thus are an attractive representation for PCG. There are probabilistic logic formulations beyond Problog that can utilize circuits such as probabilistic answer set programming (PASP) (Cozman and Mauá 2020). However, their credal semantics are more expressive than Sato 1995's semantics, and thus current inference techniques are much slower than the already slow Problog.

There are various programming features that are needed for PCG that current PLP systems, specifically Problog allow for, including flexible probabilities, stochastic memoization and negation as failure (De Raedt and Kimmig 2015). However, there are crucial features that no existing system supports, such as limiting the space of possible worlds of the programming via a first-order sentence. We will discuss this issue further in the section where we outline Problog's features. We also show how this can be partially emulated by using Problog's Bayesian conditioning operator, the evidence primitive.

A Bayesian Framework for PCG

In this section, we will formulate content generation in a Bayesian framework. We will start with a few background definitions, present an example to ground the discussion, and then discuss the details and features of this framework.

Basic Definitions

We mentioned earlier that we formulate content generators as probability distributions. We start off with a few definitions to clarify this formulation:

Definition 1 (Probability distribution)**.** Let Ω be a set. A probability distribution on Ω is a function $P : \Omega \to [0, 1]$ such that $\sum_{\omega \in \Omega} P(\omega) = 1$.

Definition 2 (Events)**.** We call subsets $E \subseteq \Omega$ *events*. The probability of an event is the sum of the probabilities of each element of the sample space in the event, i.e., $P(E) = \sum_{\omega \in E} P(\omega)$.

Definition 3 (Random Variable)**.** A *random variable* is a function $F : \Omega \to D$ for some set D. We define the probability that a random variable takes on a particular value $d \in D$ as $P(F = d) = \sum_{\{\omega | F(\omega) = d\}} P(\omega)$.

Grid Example

To ground the discussion in this section, we will begin with an example. Suppose that we are trying to write a generator that creates a level represented by a 2D grid where the player has to get from the start of the level to the end. The player can move up, down, left, or right from and to any cell that is traversable and cannot move to or from any cell that is an obstacle. For simplicity, let's assume that the start of the level is the top left of the grid and the end of the level the bottom right of the grid. We can represent such a grid as a binary string where each cell in the grid is either a 1 or a 0; indicating whether it is traversable. Figure 1 shows all such 2×2 grids, where white cells are traversable, black cells are obstacles, the green cell is the start of the level, and the red cell is the end of the level. Let's refer to the set of all possible artifacts a generator can generate as its "universe" and denote it by Ω. Thus, in our example:

$$\Omega = \{grid_0, grid_1, grid_2, grid_3, \ldots, grid_{15}\}.$$

Without any prior knowledge of what elements of Ω we would like to generate, we have no preference for any particular outcome; thus, for illustration let's assume that initially our generator is uniformly distributed over all possible artifacts.

Valid Generator

We use the term *artifact* to refer to an object a designer wishes to generate. The term is purposely vague as much of what we discuss here applies to different domains of interest in game design, such as levels, art assets, and music, that may be generated. In this paper, we limit our scope to artifacts that require a strong logical constraint to be satisfied. For example, for platform game levels to suit their purpose, there must exist a traversable path from the start of the level to the end. We refer to such hard constraints, ones that

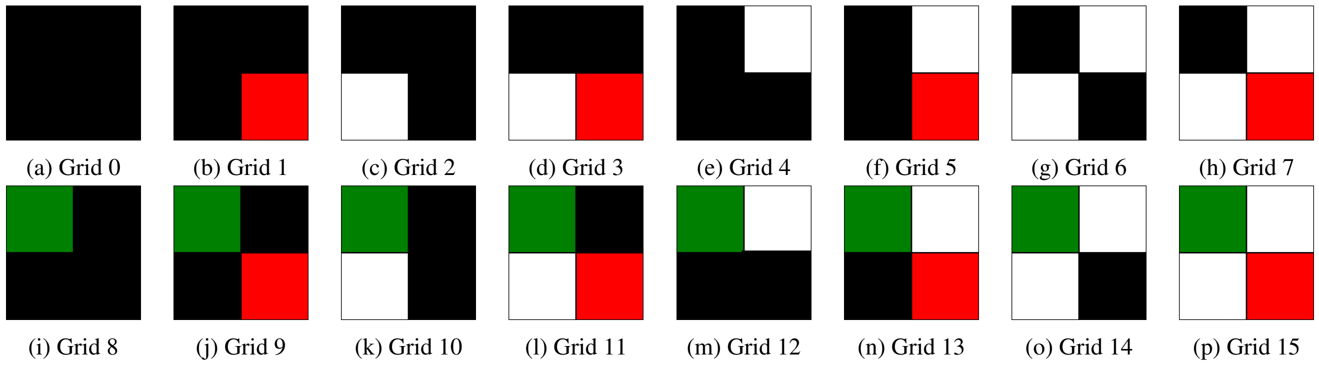

Figure 1: All possible instances of the example 2×2 grid generator. The start is colored with green and the end is colored with red

are required for an artifact to be usable in a game, as *validity constraints*. Thus, validity constraints are *hard evidence* constraints that require a definite true or false observation. For a given generation task, we refer to the set of possible artifacts as the universe or the *generative space* Ω. The task of authoring a valid generator is restricting Ω to a subset whose elements satisfy validity constraints. We refer to this subset as the *valid generative space* V. Formally we view a valid generator as a probability distribution on Ω.

Definition 4 (Valid generator)**.** Let Ω be the set of possible artifacts, and $V \subseteq \Omega$ be the set of valid artifacts. Then a valid generator is a probability distribution P such that for all $\omega \notin V$, $P(\omega) = 0$.

Thus a valid generator meets the minimum requirement of generating artifacts that a designer can use; it is technically usable and fulfills the necessary constraints. There may be multiple ways of generating V from Ω, for example in ASP we can specify rules in the program that eliminate answer sets that do not satisfy the rule. A natural way to express this operation under Bayesian semantics is through conditioning. To understand that, let us first define conditioning:

Definition 5 (Conditional Probability)**.** Let $A, B \in \Omega$ be events in Ω, and P is a distribution over Ω. Then the conditional probability $P(A \mid B)$ is defined as:

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)}. \qquad (1)$$

Similarly, let X and Y be random variables, we can define a conditional distribution on X given Y as:

$$P(X = x \mid Y = y) = \frac{P(X = x \cap Y = y)}{P(Y = y)}, \qquad (2)$$

where $P(X = x \cap Y = y) = \sum_{\omega \mid X(\omega)=x, Y(\omega)=y} P(\omega)$.

Therefore we can define a valid generator through conditioning as:

Definition 6 (Bayesian Valid generator)**.** Let P be a probability distribution on possible artifacts Ω. Let $V' : \Omega \rightarrow \{true, false\}$ be a random variable such that for some $\omega \in \Omega$, if ω is invalid then $V'(\omega) = true$ otherwise it is $false$. Then a valid generator is $P(\Omega \mid V' = true)$.

We refer to the conditioning listed above as conditioning on *hard evidence*; there is no uncertainty about what is valid. The effect of such conditioning is giving events that are not in V zero probability, which matches our initial definition of a valid generator.

Example In our grid example, a valid constraint would be the condition that a player must be able to get to the end of the level; i.e., there must exist a path from the beginning to the end. Thus, we are left with Grid 11, 13, and 15 as possible artifacts, i.e., $V = \{grid_{11}, grid_{12}, grid_{13}\}$. Without any other information about how the generator is enforcing these constraints, we can just assume that the distribution over V in our example is uniform. Table 1 shows the result of this operation and removing zero probability outcomes from Ω.

Grid ID	cell(1)	cell(2)	cell(3)	cell(4)	$P(\omega)$
11	1	0	1	1	0.333
13	1	1	0	1	0.333
15	1	1	1	1	0.333

Table 1: Valid grid configurations, i.e., valid generative space V and assuming a uniform distribution over it.

Quality Constraints

Though not all elements of V are desirable, many valid levels are bad levels. There are *quality constraints* that are difficult to articulate yet are relevant in generating high-quality artifacts. Some would argue that this is where the artistic value of game design comes in; all levels in commercial games are valid levels but not all of them are considered to be of high quality. These constraints may come in the form of hard logical constraints or a function that assesses the quality of an artifact. In the PCGML literature, one assumes that a model can learn these quality constraints from data. In our formulation we do not make any assumptions as to how these constraints are given, thus they can be directly defined or learned from data. If the constraint is a hard logical constraint we consider it hard evidence against the artifact, that is, we eliminate the possibility of its generation.

This will in turn affect the resulting distribution by eliminating more artifacts. If the constraint is not a strong logical constraint, then we consider it to be *virtual evidence*, and it can be used to influence the generator by adjusting the probabilities of valid artifacts to be higher or lower depending on the strength of the certainty of the evidence. Following the definition given by Pearl (1986), we formally specify virtual evidence on a random variable as a distribution over the co-domain of the random variable:

Definition 7 (Virtual Evidence)**.** Let P be a distribution and X and D be random variables. Let $P(D = d \mid X = x)$ be a conditional probability distribution on values of X. Conditioning P on the event $D = d$ is called *virtual evidence* of X.

A common way to think about virtual evidence is to think about them as noisy sensors; the sensors make observations with a level of uncertainty. The degree of belief we attribute to those sensors, the random variable D, is the way we model the lack of definiteness in quality constraints. Thus, we can incorporate many possible techniques that can model noisy readings, such as Bayesian networks and neural networks. Virtual evidence can become hard if we collapse the distribution to 0 or 1, that is, we can assign the evidence a value d, and set $P(D = d \mid X = x) = 1$ and $P(D = d' \mid X = x') = 0$ for all $d' \neq d \in D$.

Constraint Types Readers familiar with constraint programming and optimization might be wondering what the distinction is between quality/validity constraints and soft-/hard constraints. In that literature hard constraints are defined as constraints that restrict the solution space, i.e., they **have** to be satisfied. While soft constraints are defined as constraints that are optional but preferred; in other words, they **ought** to be satisfied but do not have to be. Solutions are eliminated by using hard constraints and penalized by using soft constraints.

Validity constraints can be considered as hard constraints; as they need to eliminate non-playable levels, for example, to ensure validity. However, quality constraints can be hard constraints or soft constraints; or an arbitrary combination of both. We make this distinction for two main reasons: the first is a focus on the separation of concerns between the validity of an artifact and an assessment of its quality. That is, we can separate the concrete constraints for the artifact to be considered workable (e.g., a level must be completable) and other attributes of the artifacts whose value may be more subjective (e.g., how many ways to complete the level). The second reason is that this formulation allows us to have the same validity generator to make multiple quality generators depending on different quality constraints, which reduces the workload needed to adapt content generators for multiple games.

Ultimately, this is a terminology convention, and one can make the formulation using hard and soft constraints, whereby hard quality constraints are folded into validity constraints.

Quality Generator

We can now formalize the notion of a *quality generator* that takes into account a set of quality constraints.

Definition 8 (Quality Generator)**.** Let Q be the set of artifacts consistent with a set of quality constraints F, $P(Q)$ be the probability that a generator P generates Q, where Q contains assignments to the virtual evidence defined by F, V be a set of valid artifacts that P can generate and $P(V)$ the probability that P generates V. Then a *quality generator* $P(V|Q)$, is defined:

$$P(V|Q) = \frac{P(Q|V)P(V)}{P(Q)}, \tag{3}$$

where $P(Q|V)$ is the probability of generating quality artifacts given that the generator generated valid artifacts V.

Readers may have already noticed that this is an example of Bayes' rule. In Bayesian statistics, we refer to $P(V|Q)$ as the posterior, $P(Q|V)$ the likelihood, $P(V)$ the prior, and $P(Q)$ as the marginal probability of evidence. Thus, the quality generator is the posterior and the valid generator is the prior. The likelihood captures how likely the generator is to generate quality artifacts given that it has generated a set of valid artifacts V. The probability of evidence is the probability over quality artifacts, and in many applications of Bayesian statistics is often just used as a normalizing constant.

The main mechanism that designers have to control the quality of the generator is specifying the likelihood. In machine learning, this likelihood is learned from data. This can happen under many forms including parameterized model (Summerville and Mateas 2015) or deep learning (Liu et al. 2020). But as we mentioned earlier, this likelihood can also be specified by the designers directly in the form of hard or virtual evidence.

Example For illustration purposes we will consider two different quality constraints; to show how the same valid generator can be used to construct multiple quality generators and how we can use both logical and non-logical constraints. Our grid example is small, and thus our quality generators will have some overlap, but that need not be the case. First, let's consider the case of the hard constraint that there must be exactly one path from the start to the end. Thus, our F will only output 1 or 0; 1 when there is one path from the start cell to the end cell and 0 otherwise. A quality generator consistent with this constraint would have non-zero probabilities for Grid 13 and Grid 11 and zero for all other cases in V, which is just Grid 15 in this example. Without any other information regarding how to weigh the remaining grids, we will assume a normal distribution. The table for this generator is in Table 2.

Grid ID	cell(1)	cell(2)	cell(3)	cell(4)	$P(\omega)$
11	1	0	1	1	0.5
13	1	1	0	1	0.5

Table 2: Grid configurations for the first quality generator

Suppose we have another generator that does allow for multiple paths and, in fact, weighs grids that allow for multiple paths more. Concretely, we can define F to return the number of traversable paths multiplied by the prior probability of the grid; then, we have to normalize to get a valid distribution by dividing each output of F to the sum of all of its outputs. This quality generator will have non-zero probability for all of our V, and only adjust the distribution, which can be seen in Table 3:

Grid ID	cell(1)	cell(2)	cell(3)	cell(4)	$P(\omega)$
11	1	0	1	1	0.25
13	1	1	0	1	0.25
15	1	1	1	1	0.5

Table 3: Grid configurations for the second quality generator

Iterative Conditioning

The definition given for quality generators obscures, for simplicity, one fact that proves to be a great advantage of this approach; the prior here is itself a posterior. If we consider the Bayesian way of obtaining $P(V)$, then it is a posterior on a $P(\Omega)$, i.e., $P(\Omega|V' = true)$. Herein lies the power of Bayesian modeling as an underlying formalism for PCG; we can allow for iterative conditioning that is non-monotonic. We can have more than one validity constraint or quality constraint, observe their results, and still have a valid distribution. Therefore, the iterative nature of designing content generators is handled naturally by Bayesian conditioning.

Problog as a Language

How do we model quality and validity constraints in a way that is understandable and controllable whilst keeping the computational process of conditioning, also known as inference, tractable? In this section, we discuss how Probabilistic Logic Programming can address this challenge and describe a specific language, Problog, and its features. Problog is a probabilistic extension of Prolog, where in addition to rules and deterministic facts, a user can specify a fact with an associated probability (Dries et al. 2015). Its semantics are based on Sato (1995)'s distribution semantics and thus Problog programs define a distribution over possible worlds in the program. Problog converts the program into a weighted boolean formula which is then compiled to circuits, on which inference amounts to a well-known problem in the knowledge-compilation literature, *weighted model counting* (Dries et al. 2015). Thus, Problog programs can be compiled and then sampled from efficiently.

Example Program

We demonstrate Problog's conditioning ability by writing a simple program to generate mazes. A more complicated version of this program will also be used in the next section. A maze can be modeled by a graph where each node models a cell in a grid. Each node is connected to its adjacent cells in the grid. We can then remove edges from the node to get a maze representation where if an edge exists from one node to another, then one can traverse from that cell in the grid to that adjacent cell. We can model this by starting off with initializing a 2×2 grid, in the form of a probabilistic graph:

```
0.5::edge(1,2,0);0.5::edge(1,2,1).
...
0.5::edge(3,4,0); 0.5::edge(3,4,1).
```

where `edge(X,Y,0)` indicates that an edge between grid cell X and grid cell Y is not connected, and `edge(X,Y,1)` indicates that it is. Each of the two possibilities has a probability of 0.5.

For a maze to be valid, we must ensure that a path exists from the start to the end. Here we assume that the start is cell 1 and the end is cell 4. To do so we tell Problog to observe the fact that `path(1,4)` is true. We do this by using the evidence primitive:

```
evidence(path(1,4)).
```

This statement allows us to condition any predicate we defined earlier in the program and adjust the distribution accordingly.

Finally, we query for the active edges in the grid that remain to get our maze.

```
query(edge(_,_,1)).
```

From this sample we can construct a graph of the level, removing nodes that do not appear in any edge.

Declarative System

Problog's declarative nature positions it as a good starting point for a generalized probabilistic framework for PCG. A non-technical end user would require some form of GUI component, but as a first step, it is fruitful to consider what features the underlying system needs to support.

Specifying generators is difficult in part due to the difficulty of specifying authorial intent through construction. As we mentioned earlier it takes a lot of expertise and effort to come up with a generator that is both suited to the design task and has diverse output. Though not all constraints are easily specifiable, Problog's is expressive enough to encompass at least Definite Clause Grammars, which are more expressive than context-free grammars and we can show that they can express context-sensitive grammars (Pereira and Warren 1980).

It remains to be shown, however, if this is a means of generator specification that makes sense to designers. As a first step, it would be useful to run a study with technically minded designers and ask them about their experience with Problog. Through that, we may encounter other desirable language features we have not considered.

Problog Ecosystem

The Problog system is composed of a modular pipeline; each function is relatively self-contained. The system first grounds the program with respect to the given query, it then converts the ground program to a conjunctive normal form formula which is then compiled to a circuit, an efficient data structure for Boolean formula. One benefit of this pipeline is the access to the circuit used for inference. This allows Problog to save models and sample from them efficiently.

Additionally, extensions to Problog exist for continuous distributions (Dries 2015), and deep learning (Manhaeve et al. 2018). Thus, Problog supports many kinds of quality constraints beyond the evidence primitive.

Shortcomings

Though Problog shows promise as a possible modeling language for PCG, given how useful Bayesian semantics can be for specifying distributions over artifacts, it leaves a lot to be desired in terms of usability. Compared to AnsProlog and popular implementations of Prolog (e.g., SWI-Prolog), it lacks sufficient syntactic sugar. Writing plain Problog is a chore as many useful shorthands, such as generating a list of integers of a certain range, do not exist.

Problog's implementation has some inconsistent behaviors when it comes to well-formedness checking. For example, erroneous Problog programs often result in null pointer errors, stack overflow, and non-termination, where the semantics of Prolog would otherwise present an error message. These can be triggered by semantic errors such as terms that were unexpectedly not grounded, annotated disjunctions that do not sum to one, and impossible worlds. As a result, it is much easier to use an alternative Prolog implementation first and then port it to Problog, which is far from ideal from a usability standpoint.

Furthermore, Problog's Bayesian conditioning language construct, the evidence primitive, only allows for literals. Thus, any first-order observations that one might want to ensure have to be enumerated manually before grounding; otherwise, Problog has inconsistent behavior.

Finally, Problog inherits Prolog's lack of strong or logical negation, and only supports negation by failure. This makes removing undesirable features from our generators much more difficult. ASP allows for the use of such negation.

Maze Generation in Problog

To show some of the benefits of utilizing a Bayesian framework, we conduct some experiments with both ASP and Problog in maze generation. For ASP we use the popular implementation Clingo (Gebser et al. 2019), and for Problog we use the most recent public revision. For these experiments, we are considering the task of generating a 5×5 maze. We arrived at this size as it was large enough to see interesting maze patterns but compact enough for both ASP and Problog to generate mazes in a reasonable time frame. Note that long computation times are to be expected as generating mazes in this manner is NP-Hard. Problog's performance is much slower than ASP and does not scale beyond 5×5 mazes. However, the current Problog implementation is implemented in Python, as opposed to the C++ based clingo, and does not leverage performance optimizations, such as TP-compilation (Vlasselaer et al. 2016). Thus it is difficult to accurately make performance claims on the Probabilistic Logic approach without implementing at least some of these optimizations. We leave that assessment to future work. To generate our mazes, we reason about a 5×5 graph in the shape of a grid and find a set of edges that make a maze with

```
#const size = 5.
cell(1..size*size).
adjacent(X,Y) :- cell(X), cell(Y),
    X-1 = Y, Y \ size > 0.
adjacent(X,Y) :- cell(X), cell(Y),
    X+size = Y.
adjacent(X,Y):- adjacent(Y,X).
1 {edge(X,Y):adjacent(X,Y)} 2*size*size.
edge(X,Y) :- edge(Y,X).
path(X,Y) :- edge(X,Y).
path(X,Y) :- edge(X,Z), Y!=Z, path(Z,Y).
:- not path(1,size*size).
:- not path(1,Y),cell(Y).
#show edge/2.
#minimize{
edge(X,Y) : cell(X), cell(Y)
}.
```

Figure 2: AnsProlog code for 5×5 maze generation.

```
0.5::edge(1,2,0);0.5::edge(1,2,1).
...
38 more lines
...
0.5::edge(24,25,0); 0.5::edge(24,25,1).
edge(X,Y,V) :- edge(Y,X,V).
path(X,Y) :- edge(X,Y,1).
path(X,Y) :- edge(X,Z,1), Y\=Z, path(Z,Y).
evidence(path(1,2)).
...
21 more lines
...
evidence(path(1,25)).
query(edge(_,_,1)).
}.
```

Figure 3: Problog code for 5×5 maze generation.

desirable qualities, namely, a path exists between the entrance and exit. We consider only mazes where each node in the graph is reachable from every other node, i.e., the graph is strongly connected. To achieve this, however, we had to enforce that all the nodes in the original 5×5 graph grid are in the final maze. This is due to the difficulty we found in enforcing strong connectivity without strong negation.

The code for both ASP and Problog was relatively small, as the compactness of expressing graph reachability in Prolog-like syntax makes the task of specifying our maze-structure straightforward. However, ASP was much more pleasant and compact to use due to the ease by which one can specify constants and primitives with ranges of value. Compare the code for maze generation in AnsProlog (Figure 2) against the code for Problog (Figure 2).

In Problog we used annotated disjunction, a rule of mutually exclusive outcomes with annotated probability, for the edges as opposed to using choice rules in ASP. Also, we had to write out each edge. We could have used a first-order variable, but we would then have to link it to some other primitives that would need to be enumerated. Similarly, we needed to enumerate all the evidence statements to ensure there is a path from node 1 to all other nodes. This led us to use Python [1] to generate the Problog program, as the lack of syntactic sugar to express enumeration made writing the program by hand a chore. In contrast, with ASP it was much easier to express the set of edges. We used the listed programs to generate 1000 5×5 mazes. In Problog, we simply used the sample feature, which samples a world accord-

[1]https://github.com/a3madkour/plp-mazes

ing to its probability in the program. For ASP we asked it to generate a single answer set with a random seed each time. Since the randomness is implementation specific, the diversity is highly sensitive to the input parameters (the parameters tested and more empirical results are in the linked Github repository). With enough ASP knowledge, an expert gets more diverse output from ASP, however, specifying that diversity is much more difficult. For these experiments, we consider the scenario where we only adjust the available input parameters that are in the Clingo documentation.

Evaluation

Diversity For our evaluation, we considered the diversity of the output for both ASP and Problog. To do this we took the generated mazes from each technique and ran the Weisfeiler Lehman graph hashing on them. Weisfeiler Lehman is guaranteed to give a unique hash for isomorphic graphs (Shervashidze et al. 2011). Out of the 1000 graphs we sampled using ASP, there were only **99** unique hashes, while all (**1000**) Problog graphs had a unique hash. Though this may seem like a surprising result, given that Problog did not generate a single isomorphic graph, it is worth considering that the space of possible graphs is very large. Thus, the probability of sampling the same graph is very unlikely, whilst ASP is likely to arrive at similar Answer Sets that the solver finds first. We conducted further analysis on the diversity of the mazes according to the metrics proposed by Kim et al. (2019b); see Figure 5 for the node types used in the metrics. Our findings from this analysis show that Problog's distribution over the metrics was less tightly peaked and more even as seen in Figure 4.

Adjusting via learning We also conducted a small experiment to test Problog capabilities for learning parameters. To do so we add the $t()$ to each annotated probability in the Problog program, which corresponds to the edge probabilities. The annotation t stands for tuneable, and this is what lets Problog know that these are parameters to learn. Note that since our program indicates that edge weights are independent we do not expect to learn parameters that massively impact the metrics we collect. The metrics we collected rely on relationships between multiple edges, and, therefore, our independence assumption severely limits Problog's ability to learn how to maximize the metrics. Nevertheless, we sought to see how well it fares despite this limitation.

We generated a set of graphs from the 1000 that the original Problog program generated, and picked out all graphs that have less than 4 decision nodes in the solution path. The solution path is the shortest path from the beginning to the end of the maze, and decision nodes are either T-junction or Cross-Junction nodes, see Figure 5. We ended up with a total of **195** graphs as our training data. This condition is arbitrary but serves as a possible design scenario a designer might wish to adjust the generator towards. We generated a new Problog program with learned parameters and used it to generate 1000 more mazes. Out of these mazes **246** had less than 4 decision nodes in the solution path. Moreover, the learned Problog program maintains the hash diversity of the original Problog program, as each of its graphs also has a unique hash and as seen in Figure 4, the learned program still maintains good diversity across the other metrics. Thus even with this ability to learn simple parameters we are able to move the generator towards a design scenario via learning and maintain a diverse set of outcomes.

PCG Tasks with PLP

In this section, we discuss common PCG tasks, analyze their mapping into our formulation and identify opportunities for future work. Further, we discuss how they can be formulated under Probabilistic Logic semantics and use the Problog system to demonstrate how they can be employed.

Specifying the Valid Space

In our formulation, this amounts to specifying the set V. Under Probabilistic Logic, this would require the user to write down the logic program that specifies the valid space. Probabilistic logic allows us to do this via Bayesian conditioning, which could reduce the authoring burden of such a task. However, this is largely a human authorial task, which makes it a fruitful avenue for future work.

Computing the Valid Space

If the logic program encodes the validity constraints by construction, that is, we leverage the closed world assumption and assume that any world that is not implied by our program is false, then computing the valid space is simply sampling from the ground program. Therefore, in this instance we do not require any explicit Bayesian inference. However, if the program requires some form of observation, e.g., using the Problog evidence primitive, then computing the valid space becomes more computationally taxing. Specifically, we require the system to compute the posterior distributions over the queried variables; in our example whether the edge is in the graph or not. In general, we can categorize this task as the probabilistic query as MAR, or compute the marginal distribution which corresponds to the well-known MAJ-SAT problem (Darwiche 2020). Thus, in general this task is PP-complete, however, compilation amortizes this cost and allows us to compute this query in polynomial time in the size of the compiled circuit (Darwiche and Marquis 2002). Doing so also guarantees an encoded distribution that the author can easily specify as part of the program.

Adjusting the Valid Space

Designers often need to adjust the validity constraints as they go about specifying their intent in an iterative manner. A naive way to accommodate this is to re-compile the program with each alteration. As far as we are aware, the current Problog system does this when adjustments are made to the program, however, this need not be the case. Circuits allow efficient transformations, and so it is tenable to adjust the valid generator without needing a full re-compilation (Darwiche and Marquis 2002). Therefore, future work can consider how to efficiently and minimally adjust a compiled Problog program without needing to go through the entire pipeline again.

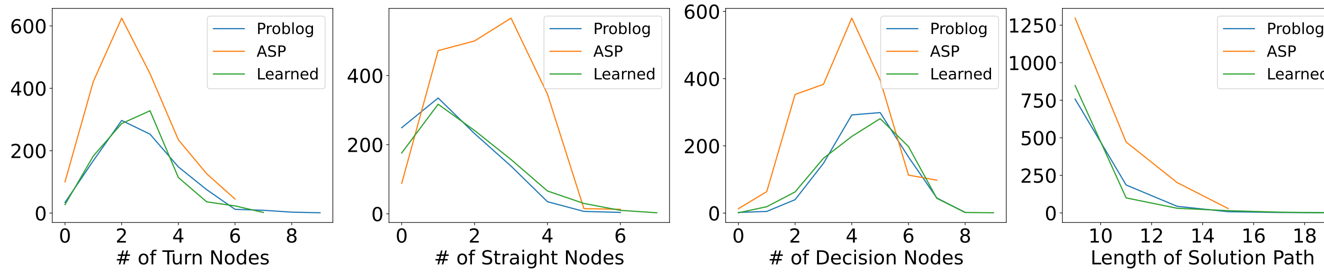

Figure 4: Solution Path metrics proposed by (Kim et al. 2019a) for ASP, Problog and Learned Problog generated mazes

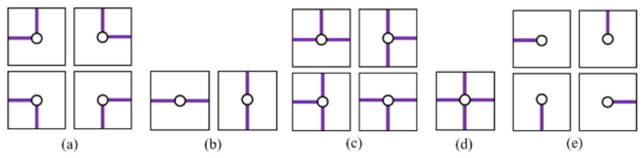

Figure 5: Node types used in the metrics proposed by Kim et al. (2019b). (a) shows Turn nodes, (b) shows Straight nodes, (c) shows T-junction nodes, (d) shows a Cross-Junction node and (e) shows Terminal nodes. Image from Kim et al. (2019b).

Generating a Valid Artifact

Once a Problog program is compiled, generating an artifact amounts to sampling from the compiled circuit. A major benefit of this approach is how efficient this task ends up being. Sampling requires a linear pass through the circuit and thus generation becomes a linear time operation. However, the circuit can increase in size significantly if the program is complex. This can be ameliorated with approximate circuits that are smaller but closely resemble the distribution of the program (Jiang et al. 2020). This approximation effort is best guided when a distribution is explicitly given, which Probabilistic Logic allows us to do.

Specifying Quality Constraints

If a quality constraint is considered to be hard evidence, then the process of its specification and computation is the same as the validity constraint. If it is virtual evidence, then we can tackle adjusting it in multiple ways. The first is to model it as part of the program and tell Problog to learn it from data, the details of which are outlined in the section where we discuss our experiments in maze generation. Another is to leverage the extensions of Problog to specify this evidence, for example, *Deep Problog* (Manhaeve et al. 2018).

Analyzing the Generative Space

A designer may wish to analyze the distribution of the artifacts as they are developing the program to specify it. If the program is sufficiently small then one simply needs to query for the variable of interest and Problog will compute the relevant posterior probabilities. Again this amounts to the MAR probabilistic query. If, however, the space of artifacts is too large for exact inference to be tractable then this becomes no longer possible. One way to deal with this is to limit the query to specific elements of the generator, for example, observe the distribution over a certain edge in our maze. Another is to sample a set of artifacts and use Expressive Range Analysis to visualize the generative space (Smith and Whitehead 2010).Though not currently attempted, one can imagine leveraging the compiled circuit for analysis of the underlying distribution directly, though the details of how to do so remain an open question.

Summary

In summary, PCG tasks map quite naturally to common queries in PLP. Due to the explicitly Bayesian nature of PLP it is more tenable for a non-system expert to glean out desirable statistical features. Moreover, since we encode the distribution of the generator in a compact data-structure, there are many avenues for future work on generalized operations for analysis and specification of PCG.

Conclusion

In this paper, we argued for the formulation of PCG as probability distributions in Bayesian terms and for the use of Probabilistic Logic as a language for the specification and control of these distributions. We showed how this can be employed in the existing Probabilistic Logic language Problog and through some experiments in maze generation showed some of the benefits of this approach over ASP. Finally, we discussed some of the limitations of Problog as a modeling language for PCG; namely, lack of quality-of-life features and support for strong negation. There are many avenues for future work, namely, tackling the issue with strong negation in Problog, visualizing the distribution that is represented by the underlying circuit, and incorporating quality-of-life features for game development, such as debugging tools and integration with existing game engines. Such efforts would get us closer to addressing the authorial needs of PCG practitioners in terms of specifying hard constraints and statistical patterns and making the dream of designer-friendly general-purpose tools for specifying content generators a reality.

Acknowledgements

This work was supported by the Northeastern (TIER) 1: Seed Grant/Proof of Concept Program.

References

Abuzuraiq, A. M.; Ferguson, A.; and Pasquier, P. 2019. Taksim: A Constrained Graph Partitioning Framework for Procedural Content Generation. In *2019 IEEE Conference on Games (CoG)*, 1–8.

Cozman, F. G.; and Mauá, D. D. 2020. The Joy of Probabilistic Answer Set Programming: Semantics, Complexity, Expressivity, Inference. *International Journal of Approximate Reasoning*, 125: 218–239.

Dabral, C.; and Martens, C. 2020. Generating Explorable Narrative Spaces with Answer Set Programming. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 16(1): 45–51.

Darwiche, A. 2020. Three Modern Roles for Logic in AI. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, 229–243. Portland OR USA: ACM. ISBN 978-1-4503-7108-7.

Darwiche, A.; and Marquis, P. 2002. A Knowledge Compilation Map. *Journal of Artificial Intelligence Research*, 17: 229–264.

De Raedt, L.; and Kimmig, A. 2015. Probabilistic (Logic) Programming Concepts. *Machine Learning*, 100(1): 5–47.

Dormans, J.; and Bakkes, S. 2011. Generating Missions and Spaces for Adaptable Play Experiences. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3): 216–228.

Dries, A. 2015. Declarative Data Generation with ProbLog. In *Proceedings of the Sixth International Symposium on Information and Communication Technology*, 17–24. Hue City Viet Nam: ACM. ISBN 978-1-4503-3843-1.

Dries, A.; Kimmig, A.; Meert, W.; Renkens, J.; Van den Broeck, G.; Vlasselaer, J.; and De Raedt, L. 2015. ProbLog2: Probabilistic Logic Programming. In Bifet, A.; May, M.; Zadrozny, B.; Gavalda, R.; Pedreschi, D.; Bonchi, F.; Cardoso, J.; and Spiliopoulou, M., eds., *Machine Learning and Knowledge Discovery in Databases*, volume 9286, 312–315. Cham: Springer International Publishing. ISBN 978-3-319-23460-1 978-3-319-23461-8.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2019. Multi-shot ASP solving with clingo. *Theory and Practice of Logic Programming*, 19(1): 27–82.

Jiang, H.; Santiago, F. J. H.; Mo, H.; Liu, L.; and Han, J. 2020. Approximate Arithmetic Circuits: A Survey, Characterization, and Recent Applications. *Proceedings of the IEEE*, 108(12): 2108–2135.

Kim, H.; Lee, S.; Lee, H.; Hahn, T.; and Kang, S. 2019a. Automatic Generation of Game Content Using a Graph-based Wave Function Collapse Algorithm. In *2019 IEEE Conference on Games (CoG)*, 1–4.

Kim, P. H.; Grove, J.; Wurster, S.; and Crawfis, R. 2019b. Design-Centric Maze Generation. In *Proceedings of the 14th International Conference on the Foundations of Digital Games*, 1–9. San Luis Obispo California USA: ACM. ISBN 978-1-4503-7217-6.

Koller, D.; and Friedman, N. 2009. *Probabilistic Graphical Models: Principles and Techniques*. Adaptive Computation and Machine Learning. Cambridge, MA: MIT Press. ISBN 978-0-262-01319-2.

Liello, L. D.; Ardino, P.; Gobbi, J.; Morettin, P.; and Teso, S. 2020. Efficient Generation of Structured Objects with Constrained Adversarial Networks. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS'20. Red Hook, NY, USA: Curran Associates Inc.

Liu, J.; Snodgrass, S.; Khalifa, A.; Risi, S.; Yannakakis, G. N.; and Togelius, J. 2020. Deep Learning for Procedural Content Generation. *Neural Computing and Applications*.

Manhaeve, R.; Dumančić, S.; Kimmig, A.; Demeester, T.; and De Raedt, L. 2018. DeepProbLog: Neural Probabilistic Logic Programming. *arXiv:1805.10872 [cs]*.

Murphy, K. P. 2012. *Machine Learning: A Probabilistic Perspective*. Adaptive Computation and Machine Learning Series. Cambridge, MA: MIT Press. ISBN 978-0-262-01802-9.

Pearl, J. 1986. Fusion, propagation, and structuring in belief networks. *Artificial intelligence*, 29(3): 241–288.

Pereira, F. C.; and Warren, D. H. 1980. Definite Clause Grammars for Language Analysis—A Survey of the Formalism and a Comparison with Augmented Transition Networks. *Artificial Intelligence*, 13(3): 231–278.

Riguzzi, F. 2018. *Foundations of Probabilistic Logic Programming: Languages, Semantics, Inference and Learning*. River Publishers Series in Software Engineering. Gistrup: River Publishers. ISBN 978-87-7022-018-7.

Sato, T. 1995. A Statistical Learning Method for Logic Programs with Distribution Semantics. In Sterling, L., ed., *Logic Programming, Proceedings of the Twelfth International Conference on Logic Programming, Tokyo, Japan, June 13-16, 1995*, 715–729. MIT Press.

Shervashidze, N.; Schweitzer, P.; Van Leeuwen, E. J.; Mehlhorn, K.; and Borgwardt, K. M. 2011. Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research*, 12(9).

Smith, A. J.; and Bryson, J. J. 2014. A logical approach to building dungeons: Answer set programming for hierarchical procedural content generation in roguelike games. In *Proceedings of the 50th Anniversary Convention of the AISB*.

Smith, A. M.; and Mateas, M. 2011. Answer Set Programming for Procedural Content Generation: A Design Space Approach. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3): 187–200.

Smith, G.; and Whitehead, J. 2010. Analyzing the Expressive Range of a Level Generator. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, 1–7. Monterey California: ACM. ISBN 978-1-4503-0023-0.

Smith, T.; Padget, J.; and Vidler, A. 2018. Graph-Based Generation of Action-Adventure Dungeon Levels Using Answer Set Programming. In *Proceedings of the 13th International Conference on the Foundations of Digital Games - FDG '18*, 1–10. Malm&#246;, Sweden: ACM Press. ISBN 978-1-4503-6571-0.

Snodgrass, S.; and Sarkar, A. 2020. Multi-Domain Level Generation and Blending with Sketches via Example-Driven BSP and Variational Autoencoders. *arXiv:2006.09807 [cs].*

Summerville, A.; and Mateas, M. 2015. Sampling hyrule: Multi-technique probabilistic level generation for action role playing games. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 11, 63–67.

Summerville, A.; Snodgrass, S.; Guzdial, M.; Holmgård, C.; Hoover, A. K.; Isaksen, A.; Nealen, A.; and Togelius, J. 2017. Procedural Content Generation via Machine Learning (PCGML). *arXiv:1702.00539 [cs].*

Togelius, J.; Yannakakis, G. N.; Stanley, K. O.; and Browne, C. 2011. Search-Based Procedural Content Generation: A Taxonomy and Survey. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3): 172–186.

Vlasselaer, J.; Van den Broeck, G.; Kimmig, A.; Meert, W.; and De Raedt, L. 2016. TP-Compilation for inference in probabilistic logic programs. *International Journal of Approximate Reasoning*, 78: 15–32.

Xu, J.; Zhang, Z.; Friedman, T.; Liang, Y.; and Broeck, G. 2018. A semantic loss function for deep learning with symbolic knowledge. In *International conference on machine learning*, 5502–5511. PMLR.

Yannakakis, G. N.; and Togelius, J. 2018. *Artificial Intelligence and Games*. Cham: Springer International Publishing. ISBN 978-3-319-63518-7 978-3-319-63519-4.