Answer Set Programming for PCG: the Good, the Bad, and the Ugly

Ian Horswill[1]
Northwestern University, Evanston IL
ian@northwestern.edu

Abstract

Declarative languages allow designers to build procedural content generation systems without having to design and debug specialized generation algorithms. Instead, the designer describes the desired properties of the objects to be generated, and a general-purpose constraint-solver constructs the desired artifact. Answer-Set Prolog (Gebser et al., 2012; Lifschitz, 2008b) is a popular family of languages and solvers used in procedural content generation research. Answer set programming is very powerful, with mature implementations and a significant user base outside the PCG community. However, ASP uses stable-model semantics (Gelfond & Lifschitz, 1992), which is subtle and difficult. In this paper, I will present some of the history and motivation underlying stable model semantics in as non-technical manner as I can manage, and discuss its advantages and disadvantages. I will argue that while it is appropriate for some very difficult PCG tasks, the simpler semantics of classical monotonic logic may be preferable for tasks not requiring ASP's non-monotonicity.

Executive Summary

This is basically a one-paragraph position paper intended to spark conversation, followed by several pages of tutorial on to explain this paragraph to those not steeped in stable model semantics. Readers who are steeped in ASP may wish to read just the first three paragraphs of the paper, while those unfamiliar with logic programming, may wish to skip to the next section (Declarative PCG Example), and those unfamiliar with formal logic may wish to skip to the section after that (Introduction).

Answer-Set Programming (Gebser et al., 2012; Lifschitz, 2019, 2008b) is an extremely useful tool for procedural content generation (A. M. Smith, 2017; A. M. Smith & Mateas, 2011, 2010; Summerville et al., 2018). It can express and solve problems that are beyond the capability of other declarative languages. However, it is also notoriously tricky. This paper is an attempt to work through the sources of its difficulty and power in the hopes of teasing them apart.

The four claims of the paper are that (1) ASP's difficulty comes in part from its nonmonotonicity: the ability to draw inferences based on the *absence* of information. Nonmonotonicity is important for domains such as legal reasoning. But to the extent that procedural content generation is generally a perfect-information problem, default reasoning is of lesser value. I would argue that (2) being simpler, classical monotonic logic is preferable when applicable. That said, ASP is more powerful than classical first-order logic insofar as it can express concepts such as transitive closure. I would argue that (3) the source of that power is ASP's use of minimization in its semantics. Since minimization does not require nonmonotonicity, (4) I suggest we explore logics that support minimization while retaining monotonicity, such as FO(TC), first-order logic with the addition of a transitive closure operator.

In my experience, the difficulty of ASP is not especially controversial. For the skeptical, I submit as evidence that my one-paragraph position statement is packaged with eight pages of tutorial. One can also compare the standard textbooks on ASP (Gebser et al., 2012; Lifschitz, 2019) to textbooks for other languages. It's not quite as bad as if *How to Design Programs* (Felleisen et al., 2018) or *Structure and Interpretation of Computer Programs* (Abelson et al., 1996) began with the Church-Rosser theorem, but they require dramatically more mathematical sophistication than those texts or Clocksin and Mellish (Clocksin & Mellish, 2003), the canonical Prolog text. While I do teach classes in which students without programming experience make PCG systems using constraint programming (Horswill, 2018b) and logic programming (Horswill, 2020), in my experience students find ASP much more challenging.

One of the appeals of constraint programming for PCG is that constraints operate relatively independently of one

another. Something is a model of a program if and only if it satisfies each constraint in the program. If your generator generates a problematic model, you can add a constraint to rule it out. If it fails to generate what you believe to be a valid model, that model must violate at least one constraint. You can find it and fix it. These properties do not hold in ASP: adding a rule can add models, remove them, or both. I once spent 20 minutes with a Ph.D. student working on an ASP thesis, one doing an SMT thesis, and one doing a PL semantics thesis, puzzling over why a 12-line ASP program was unsatisfiable. We were able to write ASP programs that did the job, but we were never able to determine why that particular one didn't.

Declarative PCG Example

The core idea of declarative PCG is to describe the objects to generate in terms of a set of choices or degrees of freedom for the objects, together with constraints on how those choices interact. It's difficult to generate an example that is compact, gets at the issues, is fair to both classical logic and ASP, and doesn't require having first read the rest of the paper. But here's my attempt.

Let suppose we're generating personalities for characters, by selecting traits, as is common in commercial games such as *Dwarf Fortress* (Adams & Adams, 2006), *The Sims* (Evans, 2009; Maxis, 2009), and *City of Gangsters* (Zubek et al., 2021; Zubek & Viglione, 2021), which does use a randomized SAT solver (Horswill, 2018a). To keep things simple, we'll assume there are two personality traits, ebullient and depressive. A character can be either of these but not both at once. And in homage to *One Night Ultimate Werewolf* (Alspach, 2009), all tanners are depressive. This gives us two constraints:

$C1$: ¬(ebulluent ∧ depressive)
C2: tanner → depressive

This has three different propositions for eight different truth assignments, and two different constraints that may be satisfied or contradicted by any given truth assignment:

Ebullient	Depressive	Tanner	Contradicts
F	F	F	None
F	F	T	C2
F	T	F	None
F	T	T	None
T	F	F	None
T	T	F	C1
T	T	T	C1

Under classical logic, the models are the ones that don't contradict any constraints. That gives us four different models (the ones in green), and so four possible characters that can be generated.

A naïve, line-by-line translation of the constraints above into ASP would read:

```
:- ebullient, depressive.
depressive :- tanner.
```

The first line says ebullient and depressive together form a contradiction. The second says that tanner implies depressive. However, this program only has one solution under ASP (one stable model): the model in which all propositions are false. The technical description in terms of stable-model semantics is complicated, but in this case it boils down to our not having provided any rules for concluding ebullient or tanner are true. In the absence of such rules, ASP requires them be false. And indeed ASP solvers such as *clingo* (Gebser et al., 2010) issue warnings when reading such a program. Depressive can't be true either because it can only be true when tanner is true, and tanner is never true. Thus, all three propositions are false.

This is unfair to ASP because an ASP programmer would know not to write it this way. They would more likely write it as:

```
0 { ebullient; depressive } 1.
{ tanner }.
depressive :- tanner.
```

The first line here says that both ebullient and depressive can be freely chosen, but at most one can be true. The second says that tanner can be freely chosen. These lines are syntactic sugar for more complicated systems of rules, which are outside the scope of this paper, see (Gebser et al., 2012). For this desugared ASP program, all four classical models are stable.

Again, this is an unfair comparison for ASP, both because it makes ASP look more verbose, and also because experienced ASP programmers wouldn't make this specific mistake. But it is indicative of the *kind* of mistake that it's easy to make as one develops a program, such as forgetting to explicitly include a choice rule, or being misled by different ways of saying something that one might naively assume to be equivalent, or simply forgetting they are different.

In the remainder of the paper, I'll explain the background underlying the position statement. I want to emphasize that ASP is unquestionably useful. I am questioning only whether it makes sense to adopt the restriction to stable models when working on problems like the one above that don't require it.

Introduction

Declarative programming languages (e.g. constraint programming, Prolog, ASP) are very attractive for PCG. They allow a designer to describe the choices involved in generating an artifact, along with constraints on the relationships between those choices, leaving it to the system to find examples (models) that satisfy those constraints. They have been used successfully both by commercial game developers (Zubek & Viglione, 2021) and non-programmers (Horswill, 2018b) for generation of characters, in-game items, character relationship maps, and prompts for tabletop role-playing games.

Answer-Set Prolog (Lifschitz, 2008b) is an extremely powerful family of declarative languages and solvers. It has been used for a number of procedural content generation tasks (A. M. Smith & Mateas, 2011), ranging from level generation (A. Smith, 2011) to generation of complete games and their critiques (Summerville et al., 2018).

ASP provides a number of clear advantages:

- It provides an expressive and concise first-order language for describing problems.
- It supports pseudo-Boolean constraints, which allow natural expression of taxonomic constraints, build-point systems, and "pick N from a menu" constraints.
- It works in part by transforming programs to SAT problems, allowing users to leverage the considerable progress in high performance SAT solving.
- It offers mature, highly optimized implementations with a sizable user base (Gebser et al., 2010, 2012)
- It is a non-monotonic logic, which makes it natural for planning and default reasoning tasks.

The first four of these, I think are non-problematic. However, the last of these, its non-monotonic semantics based on "stable" models (Gelfond & Lifschitz, 1992), while powerful, is both subtle and difficult. Here I will try to unpack those semantics in a way that is more accessible those that those that are often given (Gebser et al., 2012; Lifschitz, 2008a).

I will assume here that the reader has a basic undergraduate familiarity with set theory, (classical) propositional logic, and first-order logic, but will not assume the reader knows or remembers any of the details of Tarskian model theory. I will begin by briefly reviewing the model-theoretic semantics of classical logic, the semantics of positive logic programs (which are different), and the semantic difficulties encountered by traditional logic programming languages that motivated the development of stable-model semantics. Section headings are provided to help the reader to skip over material they're already familiar with. I will then present stable-model semantics and discuss the kinds of situations in which it is invaluable. I will also discuss why it is so difficult to understand and some possible alternatives.

Semantics of logic

A formal **logic** is a compositional language for formulating truth statements. It is a set of symbol strings, referred to as **sentences** or well-formed formulae (WFFs), defined by some recursive grammar, together with some set of semantic rules that relate the meaning of a sentence to the meaning of its constituents so that, for example, the meaning of $A \wedge B$ is derived from the meanings of A and B.

A **theory** in a logic is simply a set of sentences, all of which are asserted to be true. In most logics, this will be equivalent to the conjunction of those sentences. Thus, we will treat the theory $\{A, B\}$ and the sentence $A \wedge B$ as interchangeable.

Formal logics largely divide into **classical** and **intuitionistic** logics,[2] with the distinction being roughly that classical logics commit to every sentence having a simple truth value, while intuitionistic logics do not. Semantic systems for intuitionistic logics often identify the meaning of a sentence with the possible proofs of the sentence, while those for classical logics identify it with the possible mathematical objects for which it is true (its models).[3] Although intuitionistic logics are more relevant for much of CS, logic programming semantics is somewhat surprisingly grounded in classical logic.

Important syntactic categories

In most logics, an **atom** is a either a proposition symbol such as A or B, or the application of a predicate to some arguments, such as $P(a, b)$ or $Q(f(a), b))$. A **literal** is either an atom (a positive literal) or its negation (a negative literal). A is a positive literal, $\neg B$ is a negative literal. An atom/literal containing no variables is said to be a **ground atom/literal**.

In classical propositional logic, a (disjunctive) **clause** is a disjunction of literals, e.g. $A \vee \neg B$. Any theory has an equivalent form as a set of clauses. Any clause is equivalent to one or more **implications**, in which one disjunct is

[2] There are a number of more exotic logics, particularly substructural logics, that don't fit cleanly into these distinctions. For example, linear logic seeks to combine properties of both. However, these come more naturally as explorations of proof theory, and their model theories are their semantic theories are well outside the scope of this paper.

[3] Various non-Tarskian model theories exist for intuitionistic logics, such as those based on Heyting algebras. In particular, Heyting showed that particular kinds of subsets of the real line can be used as a kind of model for intuitionistic logic. However, these are more technical devices for studying proof systems than useful systems for relating a theory in the logic to some outside system it's trying to describe.

implied by the conjunction of the negations of the others. Thus $A \vee \neg B \vee C$ is equivalent to $\neg A \wedge B \rightarrow C$, $B \wedge \neg C \rightarrow A$, and $\neg A \wedge \neg C \rightarrow \neg B$. A **Horn clause** is a clause with at most one positive literal. It is therefore equivalent to an implication with no negative literals. For example, $\neg A \vee \neg B \vee C$ is a horn clause equivalent to $A \wedge B \rightarrow C$.

Clauses in first-order logic are more complicated because of quantifiers. Automated reasoning systems typically focus on theories composed of universally quantified clauses, that is, clauses of the form $\forall x_1, \ldots, x_n . L_1 \vee \ldots \vee L_m$ where the L_i are literals over the variables x_j.

Satisfaction

In abstract model theory (Chang & Keisler, 2012), the semantics of a logic is defined in terms of a **satisfaction relation**:

$$M \vDash S$$

stating that the mathematical object[4] M **satisfies**, or is a **model** of, S. To be a valid satisfaction relation, $\vDash$ must obey some obvious compositionality requirements, such as M being a model of $A \wedge B$ iff it's a model of both A and B, as well as the requirements that it is closed on the left under isomorphism (an object isomorphic to a model of S is itself a model of S) and on the right by variable renaming (the models of S don't depend on our choice of variable names).

If we fix a particular domain D of objects we want to describe, then we can define an extensional **meaning for S** as the set of its models:

$$\text{models}_{\text{D}}(S) = \{ M \in D \mid M \vDash S \}$$

The models of a theory are simply the intersection of the models of its sentences.

A sentence or theory is **satisfiable** if it has a model, i.e. if at least one object satisfies it. In the logics we're concerned with, a **contradiction** can be proven from a theory iff it is **unsatisfiable**.

Entailment, inference, and proof

Model theory was originally developed as a way of validating different proof systems for a given logic. A sentence is (semantically) **entailed** by a theory iff it is true in all models of the theory, i.e. the models of the theory are a subset of the models of the entailed sentence. A sentence is syntactically entailed from a theory within a proof system iff it is provable from the theory in that system. The proof system is sound and complete (i.e. good) iff syntactic entailment and semantic entailment are the same.

Minimal Herbrand models

We will focus primarily on one particular domain of models, the Herbrand base, which consists of all of the ground atoms constructible from the symbols appearing in the theory. Many logic programming systems take D, the domain of possible models, to be the set of possible subsets of the Herbrand base. These **Herbrand models** are thus just sets of ground atoms. In particular, they are the set of ground atoms taken to be true within the model.

If a theory has a model at all, it has a Herbrand model, so it's sufficient for a logic programming system to restrict its attention to Herbrand models.

As we shall see, logic programming semantics are defined in terms of **minimal Herbrand models**: models of which no subset is also a model:

$$\text{minimal}(T) = \{ M \vDash T \mid \nexists M' \subset M \,.\, M' \vDash T \}$$

Minimal Herbrand models are a kind of approximation to entailment. If a theory only has one minimal Herbrand model, then that model's atoms must be true in all models. That model is then exactly the set of atoms entailed by the theory.

If a theory has multiple minimal models, then the set of atoms entailed by the theory (if any), don't form a model by themselves. Moreover, at least one atom in each minimal model isn't an entailment. However, those models do still represent the atoms common to different clusters of models. So there's a sense in which they loosely characterize the set of models.

Minimality, transitive closure, and FOL

Minimality comes up repeatedly in logic and computation. The difference between the primitive recursive functions and the general recursive functions is the addition of a minimization operator. The Y combinator of the λ-calculus doesn't compute an arbitrary fixed-point of the function, it computes the *least* fixed-point: the one that assigns output values to as few inputs values as possible. Inductively defined sets are the least fixed-points of whatever process is being iterated to generate them.

One place where minimality frequently comes up is with transitive closure. If R is a relation (a set of pairs), then its transitive closure R^* is the smallest relation that contains R and is transitive. If we adopt the standard recursive formalization of transitive closure:

[4] Specifically, a kind of object called a *structure*, which is a set equipped with some operations and relations that can be performed on it. For example, the natural numbers under arithmetic is a structure and one can ask whether a specific theory does or does not include the natural numbers as a model, and what other kinds of systems might also be models of it.

$$\forall x, y. R(x, y) \rightarrow R^*(x, y)$$
$$\forall x, y, z. R^*(x, y) \wedge R^*(y, z) \rightarrow R^*(x, z)$$

Then this says only that R^* must be transitive and contain R, not that R^* must be minimal. The model in which R is empty but R^* is all possible pairs, is a valid model of these axioms. It's only in the minimal Herbrand model that R^* is the true transitive closure of R.

This brings up the deeply inconvenient fact that first-order logic is not strong enough to formalize transitive closure, despite the latter's apparent simplicity. While the formalization includes the desired model, it also includes a number of unintended models, even though its entailments are only the true statements about the transitive closure. FOL is not expressive enough to rule out the unintended models.

As a result, there has been a great deal of work in finite-model theory on the expressiveness of so-called intermediate logics that add to FOL some operation, such as transitive closure or a fixed-point operator (Libkin, 2004). The resulting logic is more expressive than FOL without making it full second-order logic, which doesn't even have a proof system.

Monotonicity

First-order logic is monotonic: since the models of a theory are the intersection of the models of its sentences, adding a sentence to the theory can never add models, only remove them. And since the entailments of a theory are the sentences true of all its models, adding sentences to the theory can never remove entailments.

Monotonicity is a useful property. If you are trying to debug your formalization of a domain and something that should be a model isn't, there will be guaranteed to be at least one sentence in the theory of which your desired model isn't a model. That gives you a place to start debugging.

Unfortunately, monotonicity doesn't match certain kinds of real-world human reasoning, such as reasoning about defaults. If I tell you Bill is sitting down to dinner, you will draw one set of conclusions and imagine one set of situations. But if I add that Bill is a vampire, you will suddenly imagine a very different set of circumstances.

While the set of models of a theory is monotonic as one adds sentences to the theory, the set of minimal models is not. The theory $A \rightarrow B$ has the models {}, $\{B\}$, and $\{A, B\}$, which have the single minimal model {}. However, if we add the sentence A to the theory, then the models narrow to just $\{A, B\}$ and that is now minimal.

[5] The exposition is simpler in the propositional case. For the first-order case, we think of each rule with variables as being universally quantified and as standing in for the set of all its possible ground instantiations.

Semantics of Prolog-like languages

Although logic programming languages aren't logics *per se*, their semantics are typically defined by pretending that the statements of a logic program P are really a theory $\mathcal{T}(P)$ in some underlying logic, most commonly FOL or classical propositional logic, and then defining the meaning of the program in terms of the minimal Herbrand models of $\mathcal{T}(P)$.

Positive logic programs

A propositional positive logic program is a set of **rules** of the form:

$$C \leftarrow P_1, \ldots, P_n$$

Where the conclusion, C, and the premises, P_i, are propositions.[5] Note that we are not allowing negation on either side of the arrow.

Consider the following two inference algorithms:

Backward chaining:
C is true if there is some rule $C \leftarrow P_1, \ldots P_n$ for which all P_i are true. (Note: n may be zero in which case C is trivially true).

Forward chaining:
$S = \{\}$
repeat to convergence:
$S = S \cup \{C \mid C \leftarrow P_1, \ldots P_n$ is a rule and all $P_i \in S\}$

The former takes one proposition and recursively applies rules to attempt to prove it. The latter finds all provable propositions by starting with the empty set and iteratively adding all propositions provable from the rules and the previously proven propositions, until there is no change.

Both these algorithms find propositions that can be concluded from premises. That said, a rule in this style of logic program has a directionality to it and so is not a clause in the logical sense. In classical logic, $A \rightarrow B$ not only allows you to infer B from A, but also to infer $\neg A$ from $\neg B$. In Prolog (Warren et al., 1977) and Planner (Hewitt, 1969), $B \leftarrow A$ only allows you to infer B from A.

Minimal model semantics

Nonetheless, if we pretend the rules of a program P are Horn clauses, then we can compare P's behavior to the models of $\mathcal{T}(P)$. Here, $\mathcal{T}(P)$ is simply the theory we get when we rewrite all the rules $C \leftarrow P_1, \ldots, P_n$ into Horn clauses $P_1 \wedge \ldots \wedge P_n \rightarrow C$.

One could imagine the directionality of logic programming rules leading the system to miss propositions that are

entailed by $\mathcal{T}(P)$. However, Van Emden and Kowalski (1976) showed that for positive logic programs, $\mathcal{T}(P)$ has a unique, minimal model. This model:

- consists of exactly the set of atoms entailed by the theory
- is identical to the set S computed by the forward-chaining algorithm, and
- is also identical to the set of propositions provable by the backward-chaining algorithm

This minimal model was then taken by logic programming researchers to define the semantics of positive logic programs. From that point on, semantics for logic programming languages have been based on some notion of canonical or **preferred models** (Lifschitz, 2008a).

General logic programs

A general logic program is simply a logic program that allows negations in the premises of rules, e.g.:

$$C \leftarrow P_1, \ldots P_n, \neg Q_1, \ldots, \neg Q_m$$

where n and/or m might be zero, i.e. the Ps or Qs might be absent. Unfortunately, the clausal form of such a rule is no longer a Horn clause when $m > 0$. When interpreted as a theory in classical logic, it no longer has a single minimal Herbrand model. For example, the general logic program:

$$A \leftarrow \neg B$$
$$B \leftarrow \neg A$$

when interpreted as a theory in classical logic, has the models $\{A\}$ and $\{B\}$. However, their intersection, {}, is not a model. Both models are minimal, and there is no single minimal model to take as the meaning of the program. Indeed, while the theory entails the sentence $A \vee B$, it entails no individual literals.

If one were interested in satisfaction, that is, just asking what all the different models of the program might be, then this wouldn't be a problem; we would simply take the meaning of the program to be the same as the meaning of the theory: all the models. Indeed, this is the view I will suggest below is more appropriate for PCG. However, logic programming has generally focused on trying to model inference/entailment, and so throwing one's hands up and accepting all the models is less attractive.

Negation as failure

Negation is very useful. It's difficult to express many domains without it. Classical logic programming systems, which were implemented using backward chaining, implemented negations of the form $\neg X$ by exhaustively trying to prove X, taking $\neg X$ to be proven if the attempt fails. This is certainly an improvement over disallowing negation entirely. But it departs at unexpected times from classical logic, leading to erroneous results.

Some of these issues have to do with cases where the algorithm recurses infinitely. In these cases, it fails to give the right answer, but it at least also fails to give a wrong answer.

However, negation as failure also leads to a class of bugs in which one forgets that not provably true is different from provably false. A more subtle set of issues come up when the implementation of negation interacts unpredictably with the incremental variable binding performed in classical logic programming. For example, if we consider the single-line Prolog program that asserts the truth of $p(a)$ without asserting anything else:

```
p(a).
```

Then the results we get the following results for positive queries match their naïve glosses in first-order logic:

Query	Naïve FOL	FOL	Prolog
`p(a)`	$P(a)$	T	T
`p(b)`	$P(b)$	F	F
`p(X)`	$\exists x. P(x)$	T	T

However, negation can diverge from FOL:

Query	Naïve FOL	FOL	Prlg
`not p(a)`	$\neg P(a)$	F	F
`not p(b)`	$\neg P(b)$	T	T
`X=b, not p(X)`	$\exists x. x = b \wedge \neg P(x)$	T	T
`not p(X), X=b`	$\exists \boldsymbol{x}. \neg \boldsymbol{P(x)} \wedge \boldsymbol{x = b}$	**T**	**F**
`p(X), X=b`	$\exists x. P(x) \wedge x = b$	F	F
`not not p(X), X=b`	$\exists \boldsymbol{x}. \neg\neg \boldsymbol{P(x)} \wedge \boldsymbol{x = b}$	**F**	**T**

Hence, in Prolog, conjunction is not commutative, nor is negation is its own inverse. Worse, the only way to predict when Prolog code will diverge from the naïve logical interpretation is to mentally simulate it. Writing Prolog code thus requires not only comfort with logic, but an understanding of the detailed behavior of Prolog interpreters.

These problems led to the search for a version of logic programming in which negation by failure was better behaved, eventually resulting in stable-model semantics, answer-set programming and answer-set Prolog. These systems involve transforming a logic program into a SAT problem under classical propositional logic, solving for its models, and filtering them to find the stable models. They have the advantage that there is a natural definition of stable models, and hence their semantics, independent of the inference algorithm used to solve for them. However, stable model semantics is subtle and difficult. Like SLDNF, it's close to

classical FOL, but different enough for bugs to come up when the programmer fails to anticipate ASP's divergence from their logical intuitions. So even though it does not require an understanding of the solver algorithm, it's still difficult to master.

Stable model semantics

The semantics of ASP are defined in terms of a particular kind of minimal Herbrand model called a stable model. Lifschitz (2008a) discusses 12 different definitions of stable models that all turn out to be equivalent. However, most of them require enough background in other non-monotonic logics that they can't be fully defined within the paper.

The definition that is presented in its entirety in that paper, and the one that appears most often in the literature, uses the notion of the "reduct" of a general logic program. The reduct $\mathcal{R}(P, M)$ of a program P with respect to a Herbrand model M, is the positive logic program one obtains by partially evaluating all negations in P with their valuations in M. That is, we replace $\neg P$ with false if $P \in M$, and true if $P \notin M$. For those cases where $\neg P$ is true, this effectively removes $\neg P$ from the rule. In those cases where it's false, it effectively removes the rule entirely since it contains a false premise. The reduct has no negations, and so is a positive logic program, and so has a unique minimal Herbrand model.

A model M is a stable model of a program P is if it's a model of P and also the minimal model of its reduct of P:

$$\text{stable}_P(P) = \{M \vDash \mathcal{T}(M) \mid M \in \text{minimal}(\mathcal{T}(\mathcal{R}(P, M)))\}$$

Note that since a positive logic program is its own reduct, the unique minimal model of a positive program is also its unique stable model. Thus stable model semantics is "backward compatible" with the Van Emden and Kowalski semantics, but extends it to general logic programs with negation as failure.

The good

Stable model semantics gives ASP most of the attractive properties of Prolog, while avoiding the worst excesses of negation as failure. Although limited to finite-domain problems where the system can enumerate the possible values of a variable in advance, it gives a very convenient language for expressing SAT-like problems. It can be thought of as a general mechanism for eager reduction of NBSAT problems to SAT problems.

AnsProlog includes a number of syntactic extensions to general logic programs (Gebser et al., 2012) that are especially well suited to PCG applications. In particular, choice rules and pseudo-Boolean constraints (e.g. cardinality constraints) come up frequently in PCG applications.

Since ASP reduces programs to SAT problems plus some post-processing,[6] it can leverage the considerable algorithmic improvements and performance engineering that has gone into the development of SAT solvers in the last 30 years. This allows ASP solvers to be surprisingly fast and effective in many cases.

Because it looks for minimal models (stable ones, in particular), it's expressive enough to represent transitive closure. More generally, it can reason about reachability, which traditional SMT solvers cannot easily do. You can think of stable models as representing the results of a kind of reasoning process in which atoms only appear in the model if there is some chain of reasoning that would derive that model from the rules, starting with the assumption of everything being false, and then incrementally adding new atoms as rules allow them to be inferred.

Moreover, the reasoning rules in ASP programs are not as "one-way" as they are in Prolog. If you say $A \leftarrow B$ and $\neg A$, the system will know that B must also be false. ASP programs do constraint satisfaction, albeit a particular kind.

Finally, the non-monotonicity of ASP programs makes them natural for certain kinds of default reasoning problems. Since the system always defaults the value of an atom to false unless it has a rule to justify it, other kinds of default rules can be naturally encoded into ASP rules.

The availability of a high-level modeling language that is automatically expanded into a grounded form is also a very valuable aspect of ASP, although not unique to ASP, see for example, (Torlak & Bodik, 2013). In what follows, I assume the use of a high-level modeling language and critique merely the use of stable-model semantics.

The bad

Like Prolog, negation in ASP has strange, unanticipated properties. In classical logic, the statements:

$$p$$
$$\neg\neg p$$
$$\text{false} \leftarrow \neg p$$

Are all equivalent. In ASP, the first states that p must be true, and moreover, that that fact can be used to prove other atoms. The second is not valid ASP code. And the last of these effectively means only that p must not be provably

[6] A SAT solver by itself, will find models, but not necessarily stable ones. ASP solvers work reducing the program to a particular SAT problem, but then also test whether the generated model would require circular reasoning to prove (Lin & Zhao, 2002). If so, it adds a so-called "loop formula" to the problem and backtracks. It can't add all possible loop-formulae in advance because they can be large and there can be an exponential number of them. In practice, this process works surprisingly well.

false. It's effectively true, but you can't use its truth to make further inferences; you simply filter out any otherwise stable models in which it's false. Both the first and last of these forms get used in practice.

Finally, the nonmonotonicity of ASP is a two-edged sword. If you are trying to understand why it is that something isn't a model of your formalization in a monotonic logic,[7] then at least one statement in your formalization must contradict that model. You can find that statement and use that to understand what's wrong with your formalization. You can't do that when reasoning about stable models, because the statements interact with one another in non-local ways. Divide and conquer does not automatically work for debugging.

The ugly

This definition given above for stable models is concise, precise, and almost entirely useless for the practicing programmer. For one thing, it's difficult for a programmer to read that definition and envision what models will be stable for a given set of clauses.

Perhaps more importantly, it defines stable models only for programs written in the form of implications of conjunctions. Virtually no ASP programs written in the game AI world look like that. Rather, they frequently use constructs such as choice rules and pseudo-Boolean constraints:[8]

$$1\,\{\,a;\ b;\ c\,\}\,1 \leftarrow 0\,\{c\,;d\,;e\}\,1$$

that are macro-expanded by the system into sets of rules in the canonical form. The expansion of the rule above is too complex to include here, but for a simpler example, the choice rule:[9]

$$\{\,a;\ b;\ c\,\}$$

expands into the rules:

$$\begin{aligned} a &\leftarrow \neg\bar{a} \\ \bar{a} &\leftarrow \neg a \\ b &\leftarrow \neg\bar{b} \\ \bar{b} &\leftarrow \neg b \\ c &\leftarrow \neg\bar{c} \\ \bar{c} &\leftarrow \neg c \end{aligned}$$

It is the stable models of *this* program that form the semantics of the original. Note that these rules expand not only into multiple new rules, but also into new atoms of which the programmer is unaware. Those atoms are absent from the source code, present in the stable models, but may also be absent from the output, making it difficult for novice programmers to understand the behavior of the system.

Why do people find ASP so confusing?

Stable-model semantics is a brilliant way of selectively incorporating minimization into model finding in SAT-like systems. It makes it possible to capture entailment-like inference processes within a satisfaction-based framework, and selectively loosen some of those requirements using choice rules.

That said, ASP terminology and tutorials mix satisfaction terminology (models, satisfiability, unsatisfiability) with entailment/inference terminology (defaults, rules, proof). Since it looks kind of like FOL, it behaves kind of like FOL satisfiability, and also behaves kind of like FOL inference, it's easy to slip into thinking of it as actually being one or the other. But it's actually neither; it's something in between. That leads to confusion and bugs.

ASP for Procedural Content Generation

SAT-based logic programming provides a convenient and highly expressive language for expressing finite-domain constraint satisfaction problems. It allows the kind of first-order declarative programming familiar from Prolog, without some of its misfeatures. ASP is one particular approach that uses stable model semantics, which permits the expression of concepts such as reachability and transitive closure at the cost of a steeper learning curve and a more complex debugging task.

There are a number of PCG problems for which ASP is not only appropriate, but for which it's hard to imagine and alternative. Problems that require sophisticated reasoning about reachability and provability, such as game generation (Summerville et al., 2018) or generation of levels that force particular solution methods (Polozov et al., 2015) simply cannot be solved by standard SAT techniques.

Nevertheless, there are many PCG applications, such as the one discussed at the beginning of this paper, where some kind of constraint formalism is valuable, but ASP's minimization features are unnecessary. In those cases, I would argue that ASP's nonmonotonic semantics lead to unnecessary confusion, and a more conventional SAT or SMT solver[10] would be more appropriate. CatSAT (Horswill,

[7] For example, when one uses a higher-level modeling language such as Rosette (Torlak & Bodik, 2013) to generate a SAT or SMT problem.

[8] For those unfamiliar with ASP, this says that if no more than one of the propositions c, d, and e are true, then exactly one of a, b, and c should be true.

[9] This says that the truth values of a, b, and c may be chosen freely, modulo any constraints put on them by other rules.

[10] Satisfaction Modulo Theories. An SMT solver is essentially a SAT solver that coroutines with a domain-specific constraint solver for some specialized data type, such as integers, arrays, or bit vectors.

2018a) is an example of such a system that has been deployed in a commercial game (SomaSim, 2021).

Alternatives and future work

For many problems, what I wish I had was neither traditional SAT nor ASP, but rather something that provided the kinds of minimization supported by ASP in a more selective and controlled manner. There are at least two possible strategies for doing this.

One would be to implement a satisfiability solver for one of the intermediate logics studied in finite model theory (Libkin, 2004). FO(TC), a version of first-order logic in which specific predicates can be declared to be the transitive closure of other predicates, would be an obvious choice. This could potentially be implemented in the same style as traditional ASP solvers: the system could find a model, then test just the transitive closures for minimality, and add additional clauses at runtime to defeat the particular violations of minimality that came up in that particular model. These would be the equivalents of the loop formulae generated by traditional ASP solvers. I could easily imagine that failing miserably, that the system would need to generate too many such clauses to be practical. But I would have predicted that to be true of loop formulae in ASP, and that works well.

Another possibility would be to use an SMT solver that incorporated a theory solver for connectivity reasoning in directed graphs (Bayless, 2017; Rossi et al., 2006, chapter 17). While connectivity of a graph isn't expressible in first-order logic,[11] it can be tested in linear time and random connected graphs can be constructed in near-linear time. Bayless (2017) used SAT module monotonic theories[12] with graph intervals to efficiently acyclicity constraints as well as pairwise reachability, shortest path, and maximum flow constraints. These could certainly be used in principle to implement transitive closure.

Conclusion

Answer-set programming is a powerful and versatile tool for constraint-based procedural content generation. However, its semantics are subtle and difficult to learn. Moreover, it was developed for problems very different from PCG. While there are PCG problems for which ASP seems to be the only viable method, PCG tasks that don't specifically need its minimization or non-monotonic reasoning capabilities might be better served by more conventional satisfiability solvers with more predictable semantics.

Nevertheless, ASP clearly demonstrates the power of incorporating some form of minimization into a satisfiability solver. This suggests future work on incorporating such minimization in a more targeted and predictable manner into conventional satisfiability solvers.

Acknowledgements

I would like to thank Rob Zubek and the reviewers for their comments on the original draft of this paper. I would particularly like to thank Reviewer 3 for their reference to Bayless' work, which is very much in line with what I was hoping someone would do. I would also like to thank Adam Smith and Adam Summerville for their discussions with me over the years about the pleasures and perils of ASP.

References

Abelson, H., Sussman, G. J., & Sussman, J. (1996). *Structure and interpretation of computer programs* (2nd ed.). MIT Press.

Adams, T., & Adams, Z. (2006). *Slaves to Armok: God of Blood Chapter II: Dwarf Fortress*. Bay 12 Games.

Alspach, T. (2009). *One Night Ultimate Werewolf*. Bézier Games.

Bayless, S. (2017). *SAT Modulo Monotonic Theories*. University of British Columbia.

Chang, C. C., & Keisler, J. (2012). *Model Theory* (Third edit). Dover.

Clocksin, W. F., & Mellish, C. S. (2003). *Programming in Prolog: Using the ISO Standard* (5th ed.). Springer.

Cormen, T. H., Leiserson, C. E., & L., R. R. (1990). *Introduction to Algorithms*. MIT Press.

Evans, R. (2009). AI Challenges in Sims 3. *Artificial Intelligence and Interactive Digital Entertainment*.

Felleisen, M., Findler, R. B., Flatt, M., & Krishnamurthi, S. (2018). *How to Design Programs* (2nd ed.). MIT Press.

Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., & Thiele, S. (2010). *A User ' s Guide to gringo , clasp , clingo , and iclingo ∗.*

Gebser, M., Kaminski, R., Kaufmann, B., & Schaub, T. (2012). Answer Set Solving in Practice. In *Synthesis Lectures on Artificial Intelligence and Machine Learning*. https://doi.org/10.2200/S00457ED1V01Y201211AIM019

Gelfond, M., & Lifschitz, V. (1992). The stable model semantics for logic programming. *The Journal of Symbolic Logic*, *57*(1), 274–277.

[11] This is ultimately because first-order logic has a "compactness" property that the logic of graphs does not. However, it should be said that it is possible to formalize connectivity of a graph with any fixed, finite number of nodes using what amounts to an FOL encoding of the Floyd-Warshall algorithm (Cormen et al., 1990). However, this involves adding a cubic number of clauses to the SAT problem, which is not appealing.

[12] Note that this is a different sense of monotonic than used above. Here monotonic means that the predicate obeys some partial order such that if it is true (false) for a particular set of arguments, it is also true (false) when those arguments are less under the partial order.

Hewitt, C. (1969). PLANNER: A Language for Proving Theorems in Robots. *International Joint Conference on Artificial Intelligence (IJCAI)*.

Horswill, I. (2018a). CatSAT: A Practical, Embedded, SAT Language for Runtime PCG. *AIIDE-18*.

Horswill, I. (2018b). Imaginarium: A Tool for Casual Constraint-Based PCG. *Workshop on Experimental AI in Games, AIIDE 2018*.

Horswill, I. (2020). Generative Text using Classical Nondeterminism. *Workshop on Experimental AI in Games (EXAG-20)*.

Libkin, L. (2004). *Elements of Finite Model Theory*. Springer.

Lifschitz, V. (2019). *Answer Set Programming* (1st ed.). Springer.

Lifschitz, V. (2008a). Twelve Definitions of a Stable Model. *Proceedings of the International Conference on Logic Programming (ICLP)*, 37–51.

Lifschitz, V. (2008b). What Is Answer Set Programming? *Proceedings of the 23rd National Conference on Artificial Intelligence (AAAI)*, 1594–1597.

Lin, F., & Zhao, Y. (2002). ASSAT: computing answer sets of a logic program by SAT solvers. *Artificial Intelligence*, 112–117.

Maxis. (2009). *The Sims 3*.

Polozov, O., O'rourke, E., Smith, A. M., Zettlemoyer, L., GulWani, S., & Popović, Z. (2015). Personalized Mathematical Word Problem Generation. *International Joint Conference on Artificial Intelligence (IJCAI)*.

Rossi, F., Van Beek, P., & Walsh, T. (2006). Handbook of Constraint Programming. In F. Rossi, P. Van Beek, & T. Walsh (Eds.), *Change* (Vol. 35, Issue 9). Elsevier. http://books.google.com/books?hl=en&lr=&id=Kjap9ZWcKOoC&oi=fnd&pg=PR5&dq=Handbook+of+Constraint+Programming&ots=QADjFO6ROd&sig=4KRZ7V0HgUWTO4wS4_zEw0yaxhY

Smith, A. (2011). *A Map Generation Speedrun with Answer Set Programming*. Expressive Intelligence Studio Blog. http://eis-blog.ucsc.edu/2011/10/map-generation-speedrun/

Smith, A. M. (2017). Answer Set Programming in Proofdoku. *Proceedings of the Fourth Workshop on Experimental AI in Games (EXAG 4)*.

Smith, A. M., & 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. https://doi.org/10.1109/TCIAIG.2011.2158545

Smith, A. M., & Mateas, M. (2010). Variations Forever : Flexibly Generating Rulesets from a Sculptable Design Space of Mini-Games. *IEEE Conference on Computational Intelligence and Games*, 273–280. https://doi.org/10.1109/ITW.2010.5593343

SomaSim. (2021). *City of Gangsters*.

Summerville, A., Martens, C., Samuel, B., Osborn, J., & Mateas, N. W. M. (2018). Gemini : Bidirectional Generation and Analysis of Games via ASP. *Proceedings of the Fourteenth Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE 2018)*, *1*, 123–129.

Torlak, E., & Bodik, R. (2013). A lightweight symbolic virtual machine for solver-aided host languages. *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI '14*. https://doi.org/10.1145/2594291.2594340

Van Emden, M. H., & Kowalski, R. a. (1976). The Semantics of Predicate Logic as a Programming Language. *Journal of the ACM*, *23*(4), 733–742. https://doi.org/10.1145/321978.321991

Warren, D. H. D., Pereira, L. M., & Pereira, F. (1977). PROLOG - The Language and its implementation compared with LISP. *Symposium on AI and Programming Languages*, *12*(8), 109–115. https://doi.org/10.1145/800228.806939

Zubek, R., Horswill, I., Robison, E., & Viglione, M. (2021). Social Modeling via Logic Programming in City of Gangsters. *Artificial Intelligence and Interactive Digital Entertainment (AIIDE-21)*.

Zubek, R., & Viglione, M. (2021). *City of Gangsters*. Kasedo Games.