Proceedings of the Fourteenth Artificial Intelligence and
Interactive Digital Entertainment Conference (AIIDE 2018)

CatSAT: A Practical, Embedded, SAT Language for Runtime PCG

Ian Douglas Horswill
Northwestern University, Evanston IL, USA
ian@northwestern.edu

Abstract

Answer-set programming (ASP), a family of SAT-based logic programming systems, is attractive for procedural content generation. Unfortunately, current solvers present significant barriers to runtime use in games. In this paper, I discuss some of the issues involved, and present *CatSAT*, a solver designed to better fit the run-time resource constraints of modern games. Although intended only for small problems, it allows designers to compactly specify simple PCG problems such as NPC generation, solve them in a few tens of microseconds, and to adapt solutions dynamically based on the changing needs of gameplay. We hope that by making adoption as convenient as possible, we can increase the uptake of declarative techniques among developers.

Introduction

Many procedural content generation (PCG) programs amount to making a set of random choices subject to domain constraints. Constraint programming (Rossi, Van Beek, & Walsh, 2006) is an attractive approach for such systems because it allows designers to specify the choices and constraints without having to develop a bespoke search algorithm for solving them (G. Smith, Whitehead, & Mateas, 2011).

Boolean Satisfiability (SAT) has been extensively studied as a constraint programming framework, since it is highly expressive and supports surprisingly fast solvers (Biere, Heule, Maaren, & Walsh, 2009). Answer-set programming (ASP) is a particularly convenient way to formulate SAT problems for PCG (A. M. Smith, 2017; A. M. Smith, Andersen, & Mateas, 2012; A. M. Smith, Nelson, & Mateas, 2010; A. M. Smith & Mateas, 2011). It allows programmers to specify finite-domain constraint satisfaction problems as a set of Prolog-like first-order rules. The ASP system expands them into an equisatisfiable SAT problem, a process known as *grounding*, and solves the resulting problem, generally using some variant of Contradiction-Driven Clause Learning (Marques-Silva & Sakallah, 1999), a backtracking-based systematic search algorithm.

As a simple example, suppose we want to generate a party of 3 non-player characters. Characters have:

- Three possible races (human, electroid, insectoid)
- Four possible classes (fighter, magic user, cleric, thief).
- Humans additionally have one of 3 possible nationalities
- Clerics have one of 4 possible religions.

In addition, there are constraints on the possible solutions:

- Party members should have different classes.
- Electroids can't be clerics.
- One of the religions is outlawed in one of the nations
- Another religion is mandatory in one of the other nations.

This can be written as a 17-line ASP program. Clingo (Eiter, Faber, Fink, & Woltran, 2008), the most commonly used ASP solver, can generate a party in 6ms on a modern laptop.

This is very appealing. It makes it easy to phrase PCG problems and solve them efficiently. Designers are free to incrementally add options and constraints as they see fit, without having to redesign the generator algorithm each time they make a change.

Moreover, it's easy to tailor generation on the fly by adding and removing constraints based on immediate gameplay needs. Provided the constraints aren't inconsistent, the system will simply "solve around" whatever those needs are.

Barriers to in-game execution

Unfortunately, using ASP for in-game PCG faces several challenges.

Designer transparency

One of the key factors in the success of behavior trees was the existence of designer-facing tools that allowed non-programmers to understand and manipulate them (Isla, 2005). Equivalent tools for constraint-based PCG, such as (G.

Smith, Whitehead, & Mateas, 2010), will be necessary for constraint-based PCG to have significant impact on the practice of game design.

Performance

Although 6ms is fast from the perspective of AI research, it is unacceptable for many game applications. Moreover, Clingo's working set size for the problem above was 700KB, and for more complex problems can easily be in the 100s of MB. Although it will fail entirely for large problems, our system solves the party generation problem in $11\mu s$ using on the order of 4KB of RAM.

Run-time issues

ASP solvers are primarily designed to run standalone. The ASP system is either run in advance with the results stored to a file, or run in a separate process, communicating over TCP or file I/O. This is cumbersome for desktop games and untenable for console and mobile games. Although it is possible to link the Clingo DLL directly into a game, this requires a more intimate knowledge of Clingo than most developers or researchers are willing to invest in. It also requires allocating a separate, dedicated heap for the DLL, which is problematic for mobile and console platforms. The only game I'm aware of to use Clingo at runtime is Smith's (2017) *ProofDoku*. Smith experimented with a JavaScript port of the clasp solver that ran in-browser, although it wasn't used in the final game.

Determinism

Mainstream solvers default to deterministic behavior, which is inappropriate for PCG. While there are ways of forcing Clingo to behave in a random manner, the necessary command-line options are no longer included in the documentation. The Z3 SMT solver (De Moura & Bjørner, 2008) allows a random seed to be specified, but many of its components are still fully deterministic. One must also disable certain heuristics and/or optimizations to get proper random behavior, and these are often poorly documented. Many of the instructions posted in on-line forums for getting random behavior for Clingo don't actually disable these optimizations. In the case of Z3, it's unclear what optimizations are being done, so it's hard to know what to turn off or how.

PCG as a sampling problem

In-game PCG is more like a sampling problem than a decision problem. The search space may be large, but it is generally dense with solutions. The goal is to quickly generate a solution with sufficient randomness that the player won't perceive bias in the selection process.

There is a strain of work on using hashing functions with complete solvers, such as CDCL, to sample the space of solutions to SAT problems approximately uniformly (Gomes, Sabharwal, & Selman, 2006). This work is primarily motivated by the desire to compute approximate solutions to $\#P$-complete problems. As a result, these methods go to considerable effort to try to ensure a uniform distribution, for example by repeatedly solving successively constrained instances of the problem.

However, actual uniform sampling isn't necessarily desirable. In our example above, humans and clerics have additional attributes that other classes and races do not. As a result, most characters, if uniformly sampled, would specifically be human clerics. For parties, 98% of models have a human and/or a cleric, so designer tunability is more important than uniformity. Chakraborty et al. (2014) use similar hashing techniques to allow sampling with a distribution specified by a weighting function, but their algorithms involve enumerating or counting all solutions to the hashed problems, which is too slow for use in-game.

Stochastic local search

Stochastic local search algorithms (Hoos & Stützle, 2004) are an attractive approach for constraint-based PCG. These algorithms typically begin with a random truth assignment and use a combination of random walk and greedy search to find a solution, thus avoiding the determinacy issues typical of DPLL and CDCL-based SAT solvers without having to resort to hashing. While they do not guarantee uniform sampling, or any other particular distribution, designers are likely to want to intervene to tune the distribution in any case (see Probability Patching, below).

Stochastic methods such as genetic algorithms have been used extensively for PCG applications, see (Shaker, Togelius, & Nelson, 2014) for a recent survey. But there has been surprisingly little work on them for constraint-based PCG or ASP. Despite some initial experiments with stochastic ASP solvers in the early 2000s (Bertoni et al., 2000; Nicolas, Saubion, & Stéphan, 2002), work has focused almost entirely on deterministic search, save for unpublished work by Gebser, Schaub, and Schneider on hashing in an experimental version of Clingo (xorro).

Research on SLS SAT algorithms has largely focused solving random 3-SAT problems (Selman, Kautz, & Cohen, 1995). Relatively little work has been done on SLS algorithms for highly structured SAT problems with large clauses, such as one finds in ASP programs. However, promising initial results were found with `UnitWalk` (Hirsch & Kojevnikov, 2005). This paper shows SLS is a viable, general, framework for solving simple run-time PCG problems.

CatSAT

CatSAT is a stochastic solver for an ASP-like language. It is open source and can be used as a drop-in DLL in any Unity game. It is an embedded, domain-specific language (DSL) within C#, similar to Rosette (Torlak & Bodik, 2013). This allows it to function without a separate grounder; a

CatSAT program is simply a C# program that grounds itself when executed. This strategy has several advantages: code can be more tightly integrated with other components of the game; domain properties can be leveraged to reduce the size of the grounded problem; and the system can leverage host-language tooling, such as type checking, IDE support, etc.

CatSAT is not appropriate for difficult SAT problems. However, for the kinds of simple problems it's designed for, it makes it very easy for designers to add constraint solving to their games and incrementally adapt it as needed.

Logic programming as an embedded DSL

We start by introducing objects to represent problems, propositions, rules, and solutions. A `Problem` is a collection of propositions and rules. A `Proposition` is identified by a name, an arbitrary host-language object. Rules specify sufficient conditions for a `Proposition` to be true, and `Solutions` map propositions to their truth values. The fragment:

```
var problem = new Problem();
var p = (Proposition)"p";
var q = (Proposition)"q";
var r = (Proposition)"r";
problem.Assert( p <= q, p <= r);
```

creates a new problem, stipulating that p is true iff q or r is true. We can then solve the problem for a random solution (model) and test it for the truth of a proposition:

```
var s = problem.Solve();
if (s[p])
   Console.WriteLine("p is true!")
```

Under *CatSAT*'s semantics, this program has four solutions: $\{p,q\},\{p,r\},\{p,q,r\}$, and {}. We can also force the truth value of a proposition:

```
problem[q] = false;
```

to constrain it to generate only the solutions $\{p,r\}$ and {}.

Grounding first-order rules

We model predicates as host-language functions from arbitrary arguments to `Proposition`s. Suppose the domain D is a finite collection of strings. Then the first-order rule $\forall x \in D.\, p(x) \Rightarrow q(x)$ can be expressed in *CatSAT* as the C# code:

```
foreach(var x in D)
  problem.Assert(q(x) <= p(x));
```

where $D, p,$ and q are declared as:

```
var D = new string[] {"a","b","c"};
var p = Predicate<string>("p");
var q = Predicate<string>("q");
```

As before, we can solve the problem and query the solution for the truth value of predicates:

```
var s = problem.Solve();
if (s[p("a")])
   Console.WriteLine("p(a) is true");
```

Again, predicates are just normal C# functions and they can perform arbitrary computation. If the value of the predicate is fixed in advance, the predicate can return a Boolean constant rather than a `Proposition`. The system simplifies rules, removing those whose bodies are constants. Rules can thereby test game engine data directly.

We can also encode problem structure in the predicates themselves. For example, the `SymmetricPredicate` function is identical to the `Predicate` function above, except that it guarantees that its output will map $p(i,j)$ and $p(j,i)$ to the same internal `Proposition` object, reducing the search space and memory footprint of the solver.

Quantification

Quantified rules can be expressed using loops, as above. However, one can also add generalized cardinality constraints on solutions. For example, the statement[1]:

```
program.Exists(D, d => p(d));
```

adds the requirement that $\exists d \in D.\, p(d)$. Changing `Exists` to `Unique` imposes the requirement that there be exactly one such d. Other quantifiers supported include `Quantify`, which allows the programmer to give specific upper- and lower-bounds, and its special cases, `All`, `Exactly`, `AtMost`, and `AtLeast`. These are the equivalents of ASP's choice rules.

Semantics

CatSAT's semantics are somewhat different from ASP's, and both are different from classical logic. In classical logic the meaning of a set of statements is the set of models that are consistent with all the statements.

The history of semantic theories of logic programs is complicated, and a general survey is outside the scope of this paper. All theories seek to limit the models of a logic program to some small set that can be reached using some

[1] The => construction in C# is a lambda expression. x => y means $\lambda x.\, y$.

Program clauses	Classical	Minimal	Completion	Completion models	Tight?	Stable models
$p \leftarrow q$	$\{\}, \{p\}, \{p, q\}$	$\{\}$	$p \leftrightarrow q, \neg q$	$\{\}$	Yes	$\{\}$
$p \leftarrow q, q$	$\{p, q\}$	$\{p, q\}$	$p \leftrightarrow q, q$	$\{p, q\}$	Yes	$\{p, q\}$
$p \leftarrow q, p \leftarrow r, q$	$\{p, q\}\{p, q, r\}$	$\{p, q\}$	$p \leftrightarrow q \vee r, q, \neg r$	$\{p, q\}$	Yes	$\{p, q\}$
$p \leftarrow q, q \leftarrow p$	$\{\}, \{p, q\}$	$\{\}$	$p \leftrightarrow q$	$\{\}, \{p, q\}$	No	$\{\}$
$p \leftarrow \neg q, q \leftarrow \neg p$	$\{p\}, \{q\}$	None	$p \leftrightarrow \neg q, q \leftrightarrow \neg p$	$\{p\}, \{q\}$	Yes	$\{p\}, \{q\}$

Table 1: Semantics of logic programming rules

specific deductive system. The earliest semantics for logic programs is van Emden and Kowalski's demonstration that programs consisting only of Horn clauses have a unique minimal model, of which all other models are supersets (Van Emden & Kowalski, 1976). This model has classically been taken as the "meaning" of logic programs without negation. Introducing negation complicates matters. ASP takes the acceptable models to be so-called *stable* models (Gelfond & Lifschitz, 1988). Intuitively, a stable model is a model in which every proposition is justified by a rule that concludes it and there are no circular justifications. A program that doesn't allow circular justifications is said to be *tight* (Van Gelder, Ross, & Schlipf, 1991). A program is tight iff the propositions form an acyclic dependency graph.

ASP works by converting the program into a SAT problem whose models are exactly the stable models of the ASP program. The stable models for tight programs are computed using the program's *completion*. If a proposition is defined by a set of rules $p \leftarrow b_1, p \leftarrow b_2, \ldots, p \leftarrow b_n$, then the completion of the that proposition, $p \leftrightarrow b_1 \vee \ldots \vee b_n$, states that p is true iff some b_i is true. A proposition that has no rules that conclude it is always false. The stable models of p are exactly the (classical logic) models of p's completion.

Finding stable models for non-tight programs requires adding an additional set of constraints called *loop formulae* to rule out circular justifications. Since there can be an exponential number of loop formulae, ASP solvers add them only on demand. The SAT solver co-routines with a checker that inspects generated models for justification loops. When a loop is found, the checker adds a loop formula to rule it out, and the solver backtracks.

Stable model semantics limits programs to a single model if it contains no negations, or a small number if it uses negation. This is not a feature for PCG. However, ASP includes *choice rules* that introduce propositions that don't require justifications. The standard structure of an ASP program is a set of choice rules to generate candidate solutions, rules to generate inferences from the candidates, and constraints to rule out unwanted candidates.

CatSAT's semantics

ASP essentially defaults to propositions having stable model semantics unless they appear in a choice rule. *CatSAT* adopts the opposite convention: only propositions that appear as conclusions of rules are constrained to stable models of those rules; the generator is free to assign other propositions as it likes, modulo any explicit constraints.

CatSAT requires programs to be tight. Although this hasn't been a major issue in practice, it causes issues with certain recursive definitions, such as the standard definition of connectedness in a graph:

$$\text{connected}(x, y) \leftarrow \text{edge}(x, y)$$
$$\text{connected}(x, y) \leftarrow \text{edge}(x, z) \wedge \text{connected}(z, y)$$

This definition is not tight. Worse, there can be no tight program because the class of connected graphs is not first-order definable. That said, the connected graphs of size V can be axiomatized using the Floyd-Warshall algorithm:

$$\text{connected}(x, y) \leftarrow c(x, y, V)$$
$$c(x, y, 0) \leftarrow \text{edge}(x, y)$$
$$c(x, y, k) \leftarrow c(x, y, k-1)$$
$$c(x, y, k) \leftarrow c(x, k, k-1) \wedge c(k, y, k-1)$$

This program is tight and generates a problem of size $O(V^3)$ rather than $O(2^V)$. That said, given that a connected graph can be generated in $O(V)$ time for sparse graphs and $O(V^2)$ for dense graphs, using SAT to make a random graph may not be the best use of resources. Instead, a hybrid approach in which a fast algorithm chooses the topology of the graph and *CatSAT* fills it with other information (labels for nodes or edges) would be faster for many applications.

Solving

The system generates a SAT problem in conjunctive normal form (conjunction of disjunctions). The individual disjunctions, called *clauses*, require at least one of their literals to be true, however we allow arbitrary minimum and maximum numbers to be specified, as is common. This allows quantifications, such as `Unique` and `AtMost`, discussed above, to compile to a single "clause." The solver maintains counts of how many literals from each clause are satisfied, allowing it to quickly determine the effects of a given change to its truth assignment.

Optimization

Programmers can optionally run unit resolution over the program to find variables whose values are fixed across all

Task		Problem object μs		SAT problem		Solution time μs	
Description	**Code (lines)**	**Create**	**Generate SAT**	**Clauses**	**Vars**	**Average**	**Max**
NPC generator	10	-	-	26	16	3.2	6.9
NPC generator w/stats	24	-	-	15	21	1.7	6.5
Party generator	33	-	-	82	46	11	84
Sudoku generator/solver	15	-	-	243	730	45	815
Storyteller demo	42	1127	136	202	74	68	189
Inverse Floyd-Warshall (V=5)	22	632	217	500	226	41	121
Inverse Floyd-Warshall (V=20)	22	27,087	64,720	38,000	15,601	38,923	70,440

Table 2: Timing results for simple PCG problems

models. This reduces the search space the solver needs to search, and for some problems, it can reduce it dramatically.

Solving

CatSAT uses a variant of `WalkSAT` (Selman et al., 1995), a stochastic local search algorithm, modified to support generalized cardinality constraints:

procedure Solve
  $A =$ random truth assignment
  repeat until timeout
    $C =$ unsatisfied clause
    $p =$ proposition within C
    $A[p] = \neg A[p]$
    if all clauses satisfied, **return** A

The choices of initial truth assignment and unsatisfied clause are uniform random. Many variants of the policies to choose p from within C have been studied (Hoos & Stützle, 2004). The current version of *CatSAT* uses a variant of the `Novelty+` algorithm to choose p from within C. It chooses randomly with probability ϕ, otherwise chooses the p that will lead to the most satisfied clauses. Here ϕ is a parameter that controls the greediness of the search, smaller ϕ being more greedy. We use the adaptive strategy of (Hoos, 2002) to dynamically adjust ϕ. This allows it to be greedy when it's doing well, but to detect when it reaches a local minimum and gradually increase the noise in the search until it is kicked out of the local minimum.

This algorithm has several advantages from the standpoint of a game designer. It's simple to implement, fast for modest sized problems, has good cache locality, doesn't allocate memory while solving, and only has one parameter (timeout).

Evaluation

Table 2 shows the performance of the system on several PCG problems. Tests were run single-threaded on a 2015 laptop with a 2.6GHz Intel i7-6600U processor and 16GB RAM. Line counts for code omit comments and blank lines.

The NPC and party generators are for the problem used in the introduction. Sudoku is a minimal board generator/solver for standard Sudoku. It forces uniqueness of numbers in rows and columns, but does not choose clues to give.

The "Storyteller demo" is a reimplementation of original demo of Daniel Benmergui's forthcoming *Storyteller* game (Benmergui, 2013, 2018), in which players arrange characters in comic-book-style panels and the system determines the underlying story events that would explain the configurations, and their effects on the characters. The problem encoding uses 10 predicates: $\text{rich}(x)$, $\text{caged}(x)$, $\text{evil}(x)$, $\text{has\_sword}(x)$, $\text{kill}(x,y)$, $\text{loves}(x,y)$, $\text{dead}(x)$, $\text{has\_tombstone}(x)$, one proposition (someone_free), and 13 axioms.

The Inverse Floyd-Warshall tests are provided as an example of what the system is *not* good at. These use the FW axiomatization given above to solve for a random set of edges that will give a graph with a specified transitive closure. While it works, there are much faster ways of solving this problem, as discussed above.

Hybrid solving

The embedded nature of the system makes it easy to construct pipelines of special-purpose solvers in which each stage fixes particular aspects of the solution and passes it on to the next stage. If a given pipeline stage cannot find a solution, the previous stage is restarted to produce a new solution. However, if the problem is *separable*, meaning that a solution is guaranteed to exist for any set of choices made by the previous stages, then no backtracking is required. The use of a specialized random graph generator, discussed above, is one example of a hybrid generator.

As an example, we wrote a simple solver for numeric inequality constraints. It is essentially a simplified implementation of the `WalkSMT` algorithm for stochastic SMT solving (Griggio, Phan, Sebastiani, & Tomasi, 2011), and can be written as:

procedure CatSMT
  repeat until success or timeout

m = CatSAT.Solve()
c = all inequalities marked as true in m
Use rejection sampling to solve the inequalities c

The rejection sampler here is very simple (<230 lines of C#), but is sufficient to add the generation of stats to our NPC generator, including different class-dependent constraints, such as requiring fighters to have higher strength than intelligence, or magic users to have higher intelligence than strength. Results are shown in Table 2 under "NPC generator with stats"

Probability patching

One issue with any random generator is that some kinds of configurations have more solutions than others, and so are chosen more frequently. As discussed above, 98% of possible parties in our example involve a human and/or a cleric. If this doesn't bother the designer or player, then it's not a problem. If it is problematic, various techniques can be used to adjust the sample distribution.

The simplest is a to decide a race and class in advance using a random number generator, then force their values in the `Problem` object. This gives the designer direct control over the distribution of those specific variables, and allows the SAT solver to run faster. This is another example of hybrid solving.

The probability of particular combinations can be increased by giving those combinations extra, hidden attributes to choose values for. This increases the number of notional solutions for those combinations, and thereby their frequency of occurrence. The additional attributes can then be ignored.

Conversely, combinations that are judged to occur too frequently can be controlled using rejection sampling. To reduce the frequency of insectoid fighters by 50%, check the generated character to see if they're an insectoid fighter. If so, regenerate it with a probability of 50%.

Future work

There are many obvious additions that would make the system more useful. The most obvious of these would be to integrate SMT support in the solver. It would also be useful for the system to ship with a standardized implementation of a floating-point solver, perhaps based on *Craft* (Horswill, 2015). Another useful and straightforward extension would be to modify the solver to support MAXSAT (optimization). Another possible improvement would be to add optional incremental generation of loop formulae, allowing the use of non-tight programs. However, it's unclear how well this would work with stochastic local search.

There are many performance improvements that are possible, as the current system is not especially well optimized. The use of watched literals (Moskewicz, Madigan, Zhao, Zhang, & Malik, 2001) instead of counters, for example, may improve performance.

Finally, although it may be easier to write a Sudoku generator in *CatSAT* than in raw C#, it still requires considerable comfort with both C# programming and logical axiomatization. A designer-facing tool, a la *Tracery* (Compton, Filstrup, & Mateas, 2014), that would help designers build generators will be important in the future.

Conclusion

SAT-based systems provide a flexible and highly expressive framework for finite-domain PCG problems, but have traditionally been difficult to implement in a running game. *CatSAT* shows that modest-sized problems can be efficiently expressed, solved, and integrated with a performance profile that's acceptable for a wide range of games. Generators can be easily created with a few lines of code, and solved in tens of microseconds. No search algorithms need be written or debugged. Generators can be incrementally tuned, both in terms of their constraints, and their generation frequencies. Moreover, their behavior can be dynamically steered at run-time to suit gameplay needs.

In addition to making it easier to add PCG to a game, the steerability of the system allows it to be put to unusual uses. It could, for example, be used for dynamic difficulty adjustment (DDA) to adjust the level of challenge of enemies or puzzles. It can be used for bespoke boss and item generation to fit the narrative needs of the game, based on the player's current stats and inventory. And it can be used as an aid in configuration interfaces (e.g. character or ship designers) to allow the player to specify some attributes, while allowing the system to generate reasonable default values for the remaining attributes. If the player doesn't like a chosen value, they can change it or just ask the system to choose a different random configuration.

Most excitingly, easy constraint satisfaction could allow the creation of fundamentally new game mechanics, such as the narrative puzzles of Benmergui's *Storyteller*. The best way to discover such mechanics is to get deployable versions of these new technologies into the hands of practicing game designers.

Acknowledgements

I would like to thank Ethan Robison, Adam Smith, Rob Zubek, Robby Findler, Spencer Florence, and the reviewers for their helpful references, advice, and comments.

References

Benmergui, D. (2013). Storyteller. In *Experimental Gameplay Sessions, Game Developer's Conference*. San Francisco, CA: UBM Techweb.

Benmergui, D. (2018). Storyteller. Retrieved from http://www.storytellergame.com

Bertoni, a, Bertoni, A., Grossi, G., Grossi, G., Provetti, A., Provetti, A., … Tari, L. (2000). The Prospect for Answer Sets Computation by a Genetic Model. In *AAAI Spring Symposium on Answer-Set Programming* (pp. 1–5).

Biere, A., Heule, M., Maaren, H. van, & Walsh, T. (2009). Handbook of Satisfiability. *New York*.

Chakraborty, S., Fremont, D. J., Meel, K. S., Seshia, S. A., & Vardi, M. Y. (2014). Distribution-Aware Sampling and Weighted Model Counting for SAT. In *AAAI-14*.

Compton, K., Filstrup, B., & Mateas, M. (2014). Tracery : Approachable Story Grammar Authoring for Casual Users. *Papers from the 2014 AIIDE Workshop, Intelligent Narrative Technologies (7th INT, 2014)*, 64–67.

De Moura, L., & Bjørner, N. (2008). Z3: An efficient SMT Solver. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*.

Eiter, T., Faber, W., Fink, M., & Woltran, S. (2008). A user's guide to gringo, clasp, clingo, and iclingo. *Annals of Mathematics and Artificial Intelligence*.

Gelfond, M., & Lifschitz, V. (1988). The stable model semantics for logic programming. *5th International Conf. of Symp. on Logic Programming*.

Gomes, C. P., Sabharwal, A., & Selman, B. (2006). Near-uniform sampling of combinatorial spaces using XOR constraints. *Advances In Neural Information Processing Systems*.

Griggio, A., Phan, Q.-S., Sebastiani, R., & Tomasi, S. (2011). Stochastic Local Search for SMT: Combining Theory Solvers with WalkSAT. In T. C. & S.-S. V. (Eds.), *Frontiers of Combining Systems. FroCoS 2011. Lecture Notes in Computer Science, vol 6989* (pp. 163–178). Berline, Heidelberg: Springer.

Hirsch, E. A., & Kojevnikov, A. (2005). UnitWalk: A new SAT solver that uses local search guided by unit clause elimination. *Annals of Mathematics and Artificial Intelligence*.

Hoos, H. H. (2002). An Adaptive Noise Mechanism for WalkSAT. *Proceedings of the 18th National Conference on Artificial Intelligence - AAAI'02*, 655–660.

Hoos, H. H., & Stützle, T. (2004). Stochastic Local Search: Foundations and Applications. *Stochastic Local Search Foundations and Applications*.

Horswill, I. D. (2015). Craft: A Constraint-Based Random Number Generator. In *Foundations of Digital Games (FDG-15)*.

Isla, D. (2005). Handling Complexity in the Halo 2 AI. *Game Developer's Conference 2005*. San Francisco, CA, USA: CMP, Inc.

Marques-Silva, J. P., & Sakallah, K. A. (1999). GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*.

Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., & Malik, S. (2001). Chaff: engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC 2001)*.

Nicolas, P., Saubion, F., & Stéphan, I. (2002). Answer Set Programming by Ant Colony Optimization. In *Proc. 8th European Conf. Artificial Intelligence (JELIA)*.

Rossi, F., Van Beek, P., & Walsh, T. (2006). *Handbook of Constraint Programming*. (F. Rossi, P. Van Beek, & T. Walsh, Eds.), *Change* (Vol. 35). Elsevier. Retrieved from

Selman, B., Kautz, H., & Cohen, B. (1995). Local Search Strategies for Satisfiability Testing. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, *00*.

Shaker, N., Togelius, J., & Nelson, M. J. (2014). *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*.

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

Smith, A. M., Andersen, E., & Mateas, M. (2012). A Case Study of Expressively Constrainable Level Design Automation Tools for a Puzzle Game. In *International Conference on the Foundations of Digital Games*. Raleigh: ACM Press.

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.

Smith, A. M., Nelson, M. J., & Mateas, M. (2010). LUDOCORE : A Logical Game Engine for Modeling Videogames. *Elements*, 91–98.

Smith, G., Whitehead, J., & Mateas, M. (2010). Tanagra: A Mixed-Initiative Level Design Tool. In *5th International Conference on the Foundations of Digital Games FDG 2010* (pp. 209–216). ACM.

Smith, G., Whitehead, J., & Mateas, M. (2011). Tanagra : Reactive Planning and Constraint Solving for Mixed-Initiative Level Design. *IEEE Transactions on Computational Intelligence, AI and Computer Games*, *3*(3), 201–215.

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

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.

Van Gelder, A., Ross, K. A., & Schlipf, J. S. (1991). The well-founded semantics for general logic programs. *Journal of the ACM*, *38*(3), 619–649.