Fast Procedural Level Population with Playability Constraints

Ian Horswill and Leif Foged
Northwestern University EECS Department
2133 Sheridan Road, Evanston IL 60208
ian@northwestern.edu, leif@u.northwestern.edu

Abstract

We examine the use of constraint propagation for populating indoor game levels with enemies and other objects. We introduce a notion of *path constraints*, which bound some function over the possible paths a player might take, and show how to efficiently place objects while guaranteeing path constraints. This allows the system to guarantee that power-ups are balanced to the number of enemies occurring in the level, that they're placed early enough to be useful, that keys are not hidden behind the doors they are intended to unlock, and so on. We describe a constraint solver based on interval methods that allows natural processing of numeric constraints and show that it is efficient enough to be used even on very low-end platforms.

Introduction

Object placement is an important step of level design. Objects must be placed not only in the right quantities, but also in the right spatial relationships if the level is to be solvable. In this paper, we describe a class of declarative constraints that allow designers to easily specify certain notions of playability. We then describe a system for efficient object placement in indoor levels given these constraints.

Previous work (A. M. Smith and Mateas 2010; G. Smith et al. 2011) has shown that finite-domain constraint solvers can be very effective for procedural content generation (PCG). However, finite-domain solvers have difficulty handling numeric data. We show that by extending finite-domain techniques with interval methods (Benhamou and Granvilliers 2006), we can significantly reduce both the execution time and the memory footprint of the system, allowing it to be used even on low-end platforms. Our system is very fast, making it appropriate not only for full PCG in Rogue-like games, but also as a partial automation tool for designers (G. Smith et al. 2011), a design rule checker, or even as a component of a dynamic difficulty adjustment system (Hunicke and Chapman 2003).

Example

Suppose we need to populate a dungeon with monsters and health packs. As a simple example, we will assign to each room either a monster, a health pack, a treasure, or nothing, subject to the constraints:

- There must be at least 2 monsters
- There can't be more than 2 health packs
- There must be between 2 and 5 treasure rooms
- The user has a reasonable chance of surviving, regardless of what path they take

The first three of these are straightforward combinatorial search, which we will solve using constraint propagation.

The last constraint (survivability) can also be solved using constraint propagation; however formulating the constraint is trickier, since it requires assumptions about the kinds of paths the player will take through the level. To the extent that many levels are effectively linear (Stout 2012), this can be trivial; however, it can also be done for more complicated topologies, as we will discuss shortly.

Given some set of likely paths, together with an estimate of the number of hit points a player will lose while fighting a monster, we can determine whether a player is likely to survive a level. Consider some path, $p = (v_0, v_1, \ldots, v_n)$. We can model the player's health $h(v_i)$ at any node (room) v_i along the path from their initial health at $h(v_0)$ and the estimated health increase or decrease, $\Delta(v_i)$, incurred while passing through v_i and the previous nodes:

$$h(v_i) = h(v_0) + \sum_{j=1}^{i} \Delta(\mathrm{v_j})$$

We can model $\Delta(v)$ based the item in the room v; if the room contains a health pack, $\Delta(v)$ is simply the number of HP that the health pack restores; if it contains a monster,

it's the number of hit points a typical player would lose when fighting the monster. Otherwise, it is zero.

The path is then survivable if $h(v) > 0$ for all v along the path; the *level* is survivable if every likely path is survivable, or if at least some path is survivable, depending on the designer's taste. The designer can also control difficulty by applying upper bounds to $h(v)$.

We will show that this survivability test can be efficiently computed using dynamic programming, and efficiently integrated into a constraint propagation system to guide item placement. The survivability criterion is **tunable**, allowing designers to rebalance levels or games to perform dynamic difficulty adjustment based on player performance; by increasing or decreasing a monster's estimated damage, or by raising the lowering bounds on h, we can make the level easier or harder.

Health survivability is one example of a **path constraint**. The same mechanisms can be used to implement other path constraints. For example, we can define ammo survivability in direct analogy: we score rooms in terms of how they add to or remove from the player's ammo supply, and then test whether the ammo expected ammo supply goes negative.

Lock and key problems (locked doors, special items required to defeat bosses, etc.) can also be phrased as path constraints (one per lock); the key has a score of 2, the lock of -1, and the puzzle is solvable if the path function is non-negative and has a value of 1 at the exit. Similarly, object **placement behind locked doors** can be forced by defining a path function summing the number of locked doors that have been passed through, and requiring that particular types of objects only be placed in rooms of a given lock depth.

Again, these path constraints can be used to drive object placement to guarantee that the desired properties are satisfied, e.g. that the lock is solvable along some paths, or along all paths, if that is preferable.

Formal definition

Let $G = (V, E)$ be a connected, undirected graph representing the level's topology. Here V is the set of rooms, corridors, or other spaces in the level, and E is the set of doors or other portals connecting them. An **attribute** $a : V \to S$ is then a labeling of the nodes of the graph, i.e. a function from nodes to some set, S. Populating a level is then a problem of solving for some set of user-specified attributes, given another set of user-specified constraints.

We will focus on two types of attributes. Attributes whose values are chosen from some small, finite sets, called *finite domains* in the constraint-programming literature, will be used to represent properties such as item placement (e.g. $S = \{\text{empty, monster, healthpack, treasure}\}$ in the example above; multiple attributes could be used to allow rooms to contain multiple items) or room type (e.g. $S = \{\text{room, corridor, staircase, elevator}\}$). Real-valued attributes can be used to represent general numeric properties of rooms, such as the amount of gold in the room. However, we will focus on using them to represent summary information about the other attributes of rooms and paths, such as expected health or health deltas.

A **constraint** is any relation that is required to hold on the values of attributes. For finite domain attributes, we will focus on **cardinality constraints**, which limit the number of nodes that can be labeled with a specific value (e.g. there must be exactly one boss, there should be 5-10 monsters), and **point constraints** (e.g. the last room must be a boss). For real-valued attributes, we will focus on **interval constraints**, which limit a single node or all nodes to lie in a certain range (or a single value, if the lower and upper bound are the same), and **functional constraints**, where the value of one attribute is constrained to be some function of the value of another attribute.

Finally, a **scoring** $s: D \to R$ is a mapping from some finite domain D to a number. We will use scorings to derive real-valued attributes from finite domain attributes. Let a be some node attribute, then by abuse of notation we will use $s(a)$ to denote some real-valued attribute, f, subject to the functional constraint $f(v) = s(a(v))$ for all v. In the example above, the attribute Δ was a scoring of the item placed in the room where:

$$\Delta(x) = \begin{cases} -k_1, & \text{if } x = \text{monster} \\ k_2 & \text{if } x = \text{healthpack} \\ 0, & \text{otherwise} \end{cases}$$

for some appropriate $k_1, k_2 > 0$ chosen by the designer.

Path constraints

A **path function** is some summarization of an attribute over a path. More formally, let $\oplus$ be some (associative, monotone) operator. We will generally use addition for the operator, but others can be useful too, such as multiplication or the max function, or potentially an operator over some other simply-ordered set. A path function f is then any function of the form:

$$f(p) = \oplus_{v \in p}\, a(v)$$

where p is some path in the graph and a is some numeric attribute. In the health example above, h is a path function for which the summation operator is simply $+$ and the attribute summed over is Δ.

Standard paths

It's difficult to make guarantees about truly arbitrary paths in arbitrary games. For example, if user insists on repeatedly revisiting a trap that removes 10 hit points each

time, there is little the designer can do to make that survivable, short of removing the trap entirely. So we will only consider paths in which the player is making some sort of forward progress, as specified by the designer (e.g. the notion of "flow distance" in *Left 4 Dead* (Booth 2009)). In particular, we will assume specific nodes are designated as the entrance and exit, and that expected directions of travel are provided for each edge in the graph, in the sense that players will "generally" go from room A to room B, rather than from B to A. A path is then a **standard path**, if it starts at the entrance, ends at the exit, and only follows edges in their expected directions.

More formally, we will define a connected, directed subgraph $\vec{G}$ of G (the undirected graph) to be a **standard path DAG** if every vertex of G lies along a path in $\vec{G}$ from the entrance to the exit, or (equivalently), that $\vec{G}$ is rooted at the entrance, and the exit is reachable from every node.

It should be noted that standard path DAGs are not unique, so the designer has some leeway in choosing the directions of edges. It should also be noted that many graphs have no standard path DAGs. We will discuss how to extend the method to handle these topologies shortly.

In PCG scenarios, standard path DAGs can be generated as part of the topology generation process. When using the population algorithm as part of a designer-support tool, the designer may prefer to specify edge direction manually. However, SPDs can also be generated fully automatically, using the algorithm given in the appendix.

Bounding path functions over standard paths

Given a path function f and a standard path DAG $\vec{G}$, we can compute the minimum of f over any standard path using dynamic programming. Let $f(p) = \bigoplus_{v \in p} a(v)$ for some a and $\bigoplus$, and let:

$$f^{\min}(v) = \begin{cases} a(v), & \text{if } v = \text{entrance node} \\ a(v) \oplus \min_{(v',v) \in D} f^{\min}(v'), & \text{otherwise} \end{cases}$$

That is, to find the value of $f^{\min}(v)$, we first find its value at all its predecessors in the standard path DAG $\vec{G}$. We find its minimum over all the predecessors and add the value of a at the node v. The path function $f^{\max}$ is defined analogously, with mins changed to maxes.

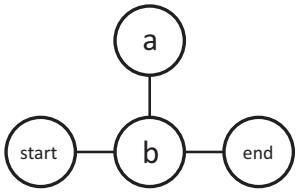
Proposition: For any vertex v, the smallest and largest values of f over any path in $\vec{G}$ from the entrance to v are given by $f^{\min}(v)$ and $f^{\max}(v)$, respectively.
Proof: by induction on the depth of v (i.e. its distance from the entrance).

Corollary: The minimum and maximal values of f over all standard paths are given by $f^{\min}(v_{\text{exit}})$ and $f^{\max}(v_{\text{exit}})$, respectively.

This then gives us an efficient way of computing the minimal/maximal values of any path function over standard paths. More importantly, by reducing an optimization over paths to finding the value of a derived attribute at a single vertex, it gives a constraint solver an efficient way of determining whether a given partial assignment of values to nodes potentially violates the constraint, allowing early filtering of candidates in the search (see below).

Handling general topologies

As stated before, standard path DAGs do not exist for many (in fact most) undirected graphs. The problem comes from *cul-de-sac* topologies, such as the following:



No standard path can exist through a because a has only one edge. Since the edge can only be assigned one direction, a path through a must violate the edge's direction either entering or exiting the node. More generally, a *cul-de-sac* is any group of nodes not including the entrance and exit, which is linked to the rest of the graph by only a single edge.

There are two approaches to handling *culs-de-sac* when computing path functions. The first is to assume the player *never* enters the *cul-de-sac* and simply remove them from the DAG entirely. This amounts to assuming the player is on a speed run, bypassing any unnecessary nodes. Enforcing path constraints then amounts to enforcing solvability on speed runs.

Alternatively, one may assume the player *always* enters *culs-de-sac* and fully explores them. We can then treat all the *cul-de-sac* nodes as if they were part of the entrance to the *cul-de-sac*. The entrance then looks like a normal node with a lot of items in it. We define the *support* of a node to be the node, together with the nodes in the *culs-de-sac* to which it is an entrance:

$$S(v) = \{ v \} \cup \{ v' \mid v' \text{ in a } \textit{cul-de-sac} \text{ of } v \}$$

We then modify the definition of $f^{\min}$ to sum over the support of a node:

$$f^{\min}(v) = \begin{cases} a(v), & \text{if } v = \text{entrance node} \\ \left(\bigoplus_{v' \in S(v)} a(v')\right) \oplus \min_{(v',v) \in D} f^{\min}(v'), & \text{otherwise} \end{cases}$$

For this to be well-defined, $\bigoplus$ must be a commutative operator, so that order doesn't matter in the sum.

This approach has the weakness of ignoring the order in which the player encounters *cul-de-sac* nodes. That weakness can be handled either by recursively forming a DAG for the cul-de-sac and applying constraints along it, or by imposing design constraints, e.g. that a *cul-de-sac* may contain either enemies or health packs, but not both.

Constraint solving

Constraint satisfaction problems (CSPs) involve finding values for some set of variables given some set of constraints. In our case, we have one variable for each attribute/node pair. That is, a variable represents the value of a single variable on a single node of the graph.

We solve the CSP using *constraint propagation*, which works by considering entire sets of candidate values for each variable. Each variable begins with the largest possible set of candidates. The algorithm then *narrows* those sets in search of specific solutions. Each time a variable is narrowed, the values ruled out are *propagated* through the constraints on the variable to rule out values of the other variables involved in the constraint. Those narrowings may, in turn, lead to further narrowings, until the process reaches fixed point. Then another variable is chosen for narrowing, until all variables have been narrowed to unique values.

The algorithm used here is a standard backtracking search, combined with a variant of Mackworth's AC3 algorithm (Mackworth 1977). Space precludes a detailed presentation, but see (Rossi et al. 2006) for an extensive survey of constraint algorithms. The heart of the algorithm is the three procedures, Solve, Narrow, and Propagate:

```
Solve( )
   choose some variable x that is not already a singleton
   for each x ∈ x            // Choose some possible value
      create a new frame on the undo stack
      try { Narrow(x, {x}); Solve( ) }
      catch (failure) { unwind undo stack frame }

Narrow(x, s)                 // Narrow variable x to set s
   n = x ∩ s                 // New value of x
   if n empty throw failure
   if (n ≠ x)
      save x on undo stack
      old = x
      x := n
      for each constraint c applied to x
         Propagate(c, x)
```

Propagate(c, $\boldsymbol{x}$), called whenever some variable $\boldsymbol{x}$, constrained by some constraint c, is narrowed, narrows the other variables involved in c, to rule out combinations that violate the constraint. It dispatches to different methods, depending on the type of constraint. For example, cardinality constraints count the number of variables that could have some specified value k, and the number that definitely have that value, then compare them to the minimum and maximum allowed occurrences:

```
PropagateCardinality(c, x, k) {
   X = set of variables constrained by c
   possible = { x ∈ X | k ∈ x}
   definite = { x ∈ X | x = {k}}
   if (possible < minimum allowed || definite>maximum)
      throw failure
   if (possible = minimum)  // make all possible definite
      for each x′ ∈ possible, Narrow(x′, {k})
   if (definite = maximum)
      for each x′ ∈ possible          // rule out remaining
         if x′ ∉ definite, Narrow(x′, x′ − {k})
}
```

This requires linear time as written, but can be reduced to constant amortized time (at the cost of complicating the undo logic) by caching the possible and minimum counts.

Constraint propagation requires that sets of possible variable values be represented efficiently. Finite domain variables have been highly studied in large part because they can be represented efficiently as bitmasks. However, this forces finite-domain solvers to quantize numeric variables into a small set of possible values and operate on those values as a discrete set.

A better choice for numeric values is to approximate sets of reals as closed intervals (i.e. $[x, y] \stackrel{\text{def}}{=} \{z \mid x \leq z \leq y\}$). Real intervals are closed under addition, subtraction, multiplication, and all monotone functions, and so are precise when constraints are limited to these operations. The propagate method for the constraint $\boldsymbol{x}_1 = \boldsymbol{x}_2 + \boldsymbol{x}_3$ is then simply

```
PropagateAddition(c, x) {
   if x ≠ x1 Narrow(x1, x2 + x3)
   if x ≠ x2 Narrow(x2, x1 − x3)
   if x ≠ x3 Narrow(x3, x1 − x2)
}
```

Where arithmetic on interval-valued variables is defined by $[a, b] + [c, d] \stackrel{\text{def}}{=} [a + c, b + d]$ and $[a, b] - [c, d] \stackrel{\text{def}}{=} [a - d, b - c]$. Propagation for min and max are defined similarly. See (Benhamou and Granvilliers 2006) for a more detailed discussion of interval methods in constraint processing.

Experimentation

We implemented the system in C# and tested it on two different graphs and four different constraint configurations. The "mansion" topology is the west wing of the first floor of the mansion from the original *Resident Evil*. Like all *RE* levels, it is designed as a single, linear path, with the remaining nodes forming 4 *culs-de-sac* along the way. The "quad" topology consists of four copies of the mansion, arranged in a diamond pattern, allowing 2 different possible routes. *Culs des sacs* were handled using the support method (summation).

Rooms were assigned contents, either empty, a large or small health pack or large or small ammo supply, a

zombie, two zombies, a zombie dog, a boss, a trap, or a locked door. Tests varied depending on the bounds of different item types:

Config	H.Pack (S/L)	Ammo (S/L)	Zomb	2 Zomb	Dog	Trap
Easy	0/1-7	n/a	1-5	1-3	n/a	n/a
Med	0/1-40	0/1-40	1-40	1-40	1-2	1
Hard	0/1-40	3	2	2	1-2	1
Big	3-10/3-10	3-10/3-10	5	5	5	2

The "easy" configuration used a reduced version of the problem involving only health survivability and 6 room contents types. All other configurations involved solving simultaneously for health and ammo survivability, as well as placing the locked door and boss (who drops the key for the door) such that they cannot be bypassed and they are encountered in the right order (boss/key before door). Problems become harder as they require more enemies to be included, since they requires more ammo and health packs, and the solver runs out of rooms to put them in.

Execution times for our system, and for Clingo (Gebser et al. 2010), a popular answer-set solver, are shown below.[1] Clingo works by first "grounding" the problem into SAT problem, then optimizing the problem, and finally solving it. Clingcon (Gebser et al. 2009), which combines ASP with a FD constraint solver may have performed better, but has not been released as of this writing. The "total" columns include the time for grounding and optimization, while the "solve" columns include only the time to solve the SAT problem. The last line shows the time for Clingo on pre-grounded input. The total column is then the solving time plus the time needed to read and unpack the ground form.

Config	Map	Time (sec)	Clingo (sec)		Speedup	
			Total	Solve	Total	Solve
Easy	Mansion	94µs	1.6	0.011	17K	117
Medium	Mansion	900µs	7.0	0.016	7818	18
Medium	Quad	0.019	34.5	0.062	1816	3.26
Hard	Mansion	0.0079	6.58	0.016	834	2.03
Hard	Quad	0.0161	33.8	0.047	2105	2.92
Big	Quad	0.0549	34.7	0.109	632	1.99
			1.03	0.078	19	1.42

The interval method is quite fast on these problems, even compared to Clingo on pre-grounded input. It also requires very little storage; we estimate less than 5KW of data for the big/quad case, small enough to easily fit in cache.[2] Clingo requires 37MB is data working set for pre-grounded input (based on Windows performance counters), and 1.5GB for doing the grounding itself.

That said, Clingo is a much more powerful solver; given sufficient time, it can find solutions to very hard problem instances (instances that are almost but not quite unsolvable) where the interval solver will just thrash. So the extra space is worthwhile for hard problems, although they would have to be solved off-line.

Related work

Procedural level generation is an active area of research. Early efforts date back at least to *Rogue* (Toy et al. 1980). A number of researchers have examined the problem of populating levels with objects. Ashmore (2007) describes a system for placement of key and lock puzzles implemented as a set of rules, although it does not provide explicit solvability guarantees. Dormans (2011) presents a system for combined topology generation and item population using graph rewriting. However, rewrite rules must be written to ensure invariants, such as puzzle solvability. *Left 4 Dead*'s "Director" (Booth 2009) places enemies and items dynamically based on candidate locations specified by designers. It uses a notion of "directionality" similar to standard path DAGs to reason about which areas are behind or in front of the player.

A number of authors have used finite-domain constraint satisfaction for PCG. *Tanagra* (G. Smith et al. 2011) uses the CHOCO solver (Jussien et al. 2008) to help generate level designs for 2D platformer games. There is also considerable work by Smith, Mateas, and colleagues on using answer-set solvers. *DIORAMA* (2011) populates levels with bases and oil wells subject to distance constraints. *Refraction* (2012) automatically generates puzzles while guaranteeing solvability and other desired properties. *Variations Forever* (2010) generates entire game rule sets using ASP. Nelson (2011) has used logical representations and constraint reasoning for determining properties of game rule sets such as solvability.

Conclusion

Procedural content generation is an interesting domain for constraint satisfaction because it is motivated by variety rather than intrinsic difficulty. CSP systems can perform very well on these problems, even fast enough for real-time operation, for such problems that have dense solution spaces. This paper can be seen both as additional support for the viability of constraint satisfaction, and as the introduction of a new, useful class of constraints.

[1] Execution times are averages per solution for the 10,000 randomly generated solutions. Tests were run on a 2009 laptop, a Sony VGN-Z691Y/B (2.66GHz Core 2 Duo P9600, 4GB of RAM), running under the .NET runtime on Windows 7 (64 bit) and using AC power.

[2] 521 variables (4 words each), 340 constraints (average of 12 words each), 1538 constraint arcs (2 words), and a maximum undo stack depth of 2038 (3 words per entry), for a total of 1534KW, plus stack, and GC overhead.

The choice of representation and solver do have a profound impact on performance, however. While not appropriate for all applications, interval methods offer an easy to implement technology for Rogue-like games that require fast, in-game solution of simple path constraints.

Appendix: Computing a standard path DAG

We can compute a standard path DAG by treating the graph as a resistor network, connect the entrance and exit to a power source, and solving for the directions of the resulting current flows using Kirchhoff's and Ohm's laws. Due to space constraints, we will only sketch the algorithm, but the approach is conceptually straightforward.

We construct a linear system with one variable V_j for each node j, representing its potential (voltage), and one variable $I_{j,k}$ for each edge $\{j,k\}$ representing the current flow through the edge. From Ohm's law, we have that

$$R_{j,k}I_{j,k} = (V_k - V_j)$$

where $R_{j,k} > 0$ is an arbitrary constant. The simplest choice is to set all $R_{j,k}$ to 1, but they can be chosen randomly or through any policy desired. Now, from Kirchhoff's current law, we have that for each node j other than the entrance and exit:

$$\sum_{\{j,k\}\in E} I_{j,k} = 0$$

Since all current entering a node must also leave it. Finally, we set:

$$V_{\text{entrance}} = 1$$
$$V_{\text{exit}} = 0$$

This gives us a system of $|V| + |E|$ equations and an equal number of unknowns. Solving linear systems generally takes cubic time; however very sparse systems such as this can typically be solved by sparse solvers in linear time.

Having solved for the currents, $I_{j,k}$, we now simply set the directions of the edges to the directions of the currents: the edge $\{j,k\}$ goes from j to k if $I_{j,k} > 0$, and k to j, if $I_{j,k} < 0$.

The interesting case comes when $I_{j,k} = 0$, i.e. where no current flows between nodes j and k. There are two cases. If current flows through both nodes, just not between them, then we have a situation where the two nodes just happen to have the same potential, and we can arbitrarily direct the edge either way, or omit it entirely from the DAG. If one of the nodes has no current flowing through it at all, then it is part of a *cul-de-sac*. The *cul-de-sac* to which it belongs will be precisely the connected set of nodes with equal potential and zero current flow, and they will be connected to the main graph by exactly one node with the same potential, but non-zero current flow. The *cul-de-sac* can then be handled using either of the methods discussed above.

References

Ashmore, Calvin. 2007. "The quest in a generated world." *Proc. 2007 Digital Games Research Assoc.* 503-509.

Benhamou, Frédéric, and Laurent Granvilliers. 2006. Continuous and interval constraints. In *Handbook of Constraint Programming*, ed. Francesca Rossi et al. Elsevier.

Booth, Michael. 2009. The AI Systems of Left 4 Dead. In *Artificial Intelligence and Interactive Digital Entertainment 2009*. Stanford, CA: AAAI Press.

Dormans, Joris. 2011. "Proceedings of the 2nd International Workshop on Procedural Content Generation in Games." *Workshop on Procedural Content Generation in*.

Gebser, Martin et al. 2009. Constraint Answer Set Solving. In *ICLP '09 Proceedings of the 25th International Conference on Logic Programming*, 235 - 249. Pasadena, CA: Springer.

Gebser, Martin et al. 2010. *A User ' s Guide to gringo , clasp , clingo , and iclingo ∗*. Potsdam.

Hunicke, Robin, and Vernell Chapman. 2003. "AI for Dynamic Difficulty Adjustment in Games." *Proceedings of the Challenges in Game AI Workshop, Nineteenth National Conference on Artificial Intelligence*: 91–96.

Jussien, Narendra et al. 2008. The CHOCO constraint programming solver. In *CPAIOR08 workshop on OpenSource Software for Integer and Contraint Programming OSSICP08*.

Mackworth, Alan K. 1977. "Consistency in networks of relations." *Artificial Intelligence* 8: 99-118.

Nelson, Mark J. 2011. Game Metrics Without Players : Strategies for Understanding Game Artifacts. In *Workshop for AI in the Game Design Process at AIIDE 2011*, 14-18. Stanford, CA: AAAI Press.

Rossi, F et al. 2006. *Handbook of Constraint Programming*. Ed. F Rossi et al. *Change*. Vol. 35. Elsevier.

Smith, Adam M, and Michael Mateas. 2010. Variations Forever : Flexibly Generating Rulesets from a Sculptable Design Space of Mini-Games. In *IEEE Conference on Computational Intelligence and Games*, 273-280. Copenhagen: IEEE Press.

Smith, Adam M, and Michael Mateas. 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, Adam M. et al. 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, Gillian et al. 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.

Stout, Michael. 2012. "Learning From The Masters: Level Design In The Legend Of Zelda." *Gamasutra*.

Toy, Michael et al. 1980. Rogue. Computer Science Research Group, UC Berkeley.