Chapter 8
ASP with applications to mazes and levels (DRAFT)

Mark J. Nelson and Adam M. Smith

A common theme underlying procedural content generation is that we need to be able to specify both *what* we want our generated content to be like, and *how* to generate it. Sometimes these two parts are tightly intertwined. In the constructive methods of Chapter 3 and the fractal and noise methods of Chapter 4, we can produce different kinds of output by tweaking the algorithms until we're satisfied with their output. But if we know what properties we'd like generated content to have, it's more convenient to be able to directly specify what we want, and then have a more general algorithm find content meeting our criteria.

The search-based framework introduced in Chapter 2 is one common way of making a content-generation algorithm general, so we can tell it what kind of content we want, and have it search for content meeting our request. An *evaluation function* specifies the properties we'd like the content to have, by numerically rating the quality of generated content according to whatever criteria we choose. A *search algorithm* then searches a space of *content encodings* to find highly rated content.

Evaluation functions summarize a large range of possible content qualities into a single numerical rating. Then the search process, such as an evolutionary algorithm, finds content that rates highly on that scale. Elements of an evaluation function may include both *hard constraints*—things that the content absolutely must have, such as a level being passable—and softer preferences. Evaluation of content quality may also depend on the game's mechanics. For example, whether a level is passable can depend on how a player can move, what items are available for the player's use, how enemies move, and so on; in search-based PCG this is often addressed by simulating gameplay when compute the rating.

8.1 Game logic and content constraints

Instead of using a content encoding and a numerical evaluation function, an alternative way of specifying what content we want generated is to define the *logic* of a

content domain, along with *constraints* on the properties that we want the generated content to exhibit [9].

The logic of a game content domain is its structure and game mechanics. A grid-based map has a structure in which tiles are arrayed horizontally and vertically, with walls, items, structures, or other entities placed on tiles. Mechanics specify how gameplay takes place on this grid. Common mechanics include: a player starts somewhere, can move to any unoccupied adjacent square, can pick up certain kinds of items, can break certain kinds of barriers (this might require an item), etc. In short, how a game *works* makes up its logic. This logic can be encoded in computational logic [7], which means we will be able to use it to guide PCG. We don't encode how the *entire* game works, to be clear: just how the game works to the extent that it's relevant to generating the content we want.

Once we have the logic of a domain, we can write down properties that we want all generated content to have, by writing constraints that refer to the game's logic. For example: a level must have a valid path through it. What is a valid path? A sequence of moves that a player can legally make; this in turn depends on the logic of the particular game's world and rules, which we have specified. Some other possible constraints: all valid paths should be at least a certain minimum length, the exit and entrance must be at opposite edges of the map, and so on. We can add and remove from these properties, as we think of them: perhaps the player shouldn't be able to get through a level without using at least one item (if our game has items). Maybe at least one jump should be required, or there should be a boss placed somewhere that can't be avoided. Specifying these constraints will often be done iteratively. Once we generate a few example levels, we may see things we didn't expect, and modify the set of desired properties accordingly.

The logic and constraints together serve the role that the encoding and evaluation function in search-based PCG, but in a more explicit, symbolic form, where we've written out the logic of a game world and the properties we'd like in the generated content. The logic and constraints are then passed to a tool, called a *solver*, which solves the logic problem: it finds content that conforms to the logic of the game world and satisfies all the constraints we've specified. This approach is particularly useful when many of our desired properties are hard constraints, and may depend (perhaps in complex ways) on the game's mechanics.

8.2 Answer set programming

To apply the approach we just described in practice, we need a specific language in which to encode the game logic and constraints, and a solver for that language. In this chapter, we use *answer-set programming* (ASP), a logic programming language. While there are other possible ways to do PCG with constraint solving [6], answer-set programming is a well-developed programming language with reliable existing tools, and which can be used to specify both game logic and constraints

within the same language. Therefore it serves as good general-purpose choice for programming logic- and constraint-based PCG systems.[1]

Before we jump into using ASP for a content generation task, we will first introduce some basic syntax. Answer set programs are expressed in a language called AnsProlog [1, 4], a language that visually resembles Prolog while having semantics that are more directly relatable to SAT and MAX-SAT problems.

The simplest ASP construct is a *fact*. A fact is something we declare to be true. It can be an atomic fact, which is simply a symbol that is declared true:

```
game_over.
gravity_enabled.
```

Or, a fact can be specified using *predicates*, which take parameters. A predicate can be declared true for specific choices of parameters:

```
max_jump(3).
item_at((2,2),wall).
```

So far, this is just a bare list of facts. We could nonetheless encode a whole level this way, specifying where walls, items, etc. are located. But the interesting part comes when we add rules. Rules specify that we can infer certain facts from others, which encodes dependencies between game elements, and also lets us start specifying game mechanics. For example, let's say a tile is impassable for the player if it contains a wall:

```
impassable(Tile) :- item_at(Tile,wall).
```

Here, `Tile` is a logic variable. In AnsProlog, variables start with a capital letter, while predicates and atoms start with a lowercase letter. These rules can be thought of as a reversed version of the implication formulas in first-order logic. Written in conventional mathematical notation, it would be:

$$\forall Tile, item\_at(Tile, wall) \implies impassable(Tile)$$

Read left to right, this says: for all tiles, if the tile contains a wall, then the tile is impassable. One can think of `:-` as a leftward-pointing implication arrow, following the programming-language convention that assignments go from right to left. Variables in AnsProlog are implicitly universally quantified, so the "for-all" ($\forall$) isn't necessary.

Once we have facts and rules, that would in principle be enough to constructively generate content. However, it is typically difficult to write a set of facts and rules so that *only* content we want is derived by the implications, placing everything in exactly the right combination of places and never generating broken or undesirable output. Instead, we usually generate content in two steps. First, we constructively define a *design space*. Then we specify constraints that exclude unwanted parts of the design space.

The initial, larger design space is created by using the AnsProlog construct of *choice rules*. A choice rule specifies that the solver has an arbitrary choice in how to assign certain facts—as long as they meet some numerical constraints, and any

[1] ASP has also been used for content generation outside of games, notably to generate music [2].

other constraints that we might add later. The following choice rule specifies that there are between 5 and 10 walls in the level, but it doesn't specify exactly how many, or on which tiles they're located:

```
5 { item_at(T,wall) : tile(T) } 10.
```

More precisely, this syntax says that, if we construct a big collection of candidate `item_at(T,wall)` facts, for every possible `T` that is a `tile`, then the size of this set is at least 5, but no greater than 10. If we have no desire to constrain the set size, we can leave off one or both of these numbers. The following choice rule simply says that a level has any number of walls:

```
{ item_at(T,wall) : tile(T) }.
```

A program consisting of only the above rule produces a generative space of levels that contains any possible arrangement of walls on a grid. Of course, interesting levels require more than this. Besides adding numerical constraints on how the ASP solver makes it choices, we can exclude unwanted choices by adding different constraints that the solver must take into account. A standalone constraint is written like a rule, but has nothing on the left hand side of the `:-` syntax. A solution that matches the right-hand side of the rule will be *rejected* as an invalid choice. The following example rule excludes any generated map that has a wall at $(1,1)$:

```
:- item_at((1,1),wall).
```

If we read this again as a logical implication written backwards, this translates to:

$$item_at((1,1),wall) \implies [\text{reject}]$$

By intermixing rules that create generative spaces, and others that prune them back down to interesting subsets, we can achieve strong control over the kinds of content that is generated.

AnsProlog code is put into files with the conventional extension `.lp` (for "logic program"), and then passed to the solver. In this chapter we use the solver `clingo` from the University of Potsdam, a free and actively maintained AnsProlog solver that's part of the Potassco project of answer-set programming tools [5].

Now that we have the basic machinery of AnsProlog, we can define facts and implications, specify design spaces as free choices, and specify constraints rejecting some of those choices. We'll walk through some complete examples to show how to build and modify procedural level generators using this method.

8.3 Perfect mazes

Using our new found ability to reason over all possible logical worlds, we will start with a simple maze generation problem. In particular, we will look at generating *perfect* mazes. A perfect maze (which may or may not actually be a desirable maze) is one in which every location is reachable while not having any closed loops. In

effect, perfect mazes are trees that have been embedded into a fixed space, usually a grid.

One way to represent a tree embedded in a grid is to assign each tile in the grid a parent pointer that points to one of its adjacent cells. If the choice of parent pointers actually forms a tree, then it will be possible to traverse these pointers back to the root of the tree no matter where we start.

Let's begin by establishing a representational vocabulary for our mazes. Figure 8.1 is a self contained AnsProlog program that uses a choice rule to assign each X/Y location a unique parent direction. This choice rule can produce facts like `parent(5,7,0,-1)` which might read that the tile at location $(5,7)$ has a parent of $(5,6)$. The location $(1,1)$ will later function as the root of our tree, so we don't assign it a parent direction.

```
#const width = 5.
dim(1..width).

1 { parent(X,Y, 0,-1),
   parent(X,Y, 1, 0),
   parent(X,Y,-1, 0),
   parent(X,Y, 0, 1) } 1 :-
   dim(X), dim(Y), (X,Y) != (1,1).
```

Fig. 8.1: maze-core.lp

With just a single interesting rule, we can already begin visualizing the output of the design space we are representing so far. Using a command like the following, which uses the answer set solving system from the Potassco project (discussed in the previous section), we can generate ASCII-art previews of possible mazes. Examples from our program so far can be seen in Figure 8.2.

```
clingo maze-core.lp --rand-freq=1
```

```
()  ()  ()==()==()
        ||      ||
()==()  ()  ()==()
                ||
()  ()==()  ()==()
    ||      ||
()  ()==()  ()==()
||      ||  ||
()  ()  ()  ()  ()
```

```
()  ()==()==()  ()
    ||  ||  ||
()  ()  ()  ()  ()
||  ||          ||
()  ()  ()==()  ()
||
()  ()==()==()==()
        ||      ||
()  ()  ()==()  ()
```

```
()  ()  ()==()  ()
||      ||
()==()  ()==()  ()
    ||  ||      ||
()==()  ()==()==()
||  ||
()  ()  ()  ()  ()
    ||  ||  ||  ||
()  ()  ()  ()==()
```

Fig. 8.2: When each tile in the maze is assigned a random parent, typical outputs show several disconnected components. Some tiles on the edges of the maze even point to a parent cell outside of the maze.

To make sure we only see valid trees, we should enforce the property that the root is reachable from every tile on the grid. Figure 8.3 uses a fact, a recursive rule, and an integrity constraint to accomplish this. The `linked(X,Y)` property holds for the root of the tree trivially. Any tile that has a parent that is linked is linked as well. Finally, if there is some tile which does not have the linked property, something is wrong with the current assignent of parent directions and this possible world should be rejected.

```
linked(1,1).
linked(X,Y) :- parent(X,Y,DX,DY), linked(X+DX,Y+DY).

:- dim(X;Y), not linked(X,Y).
```

Fig. 8.3: maze-reach.lp

After adding these rules, we can sample examples of all and only those perfect mazes by running a command like the following. Example outputs are showin in Figure 8.4

```
clingo maze-core.lp maze-reach.lp
```

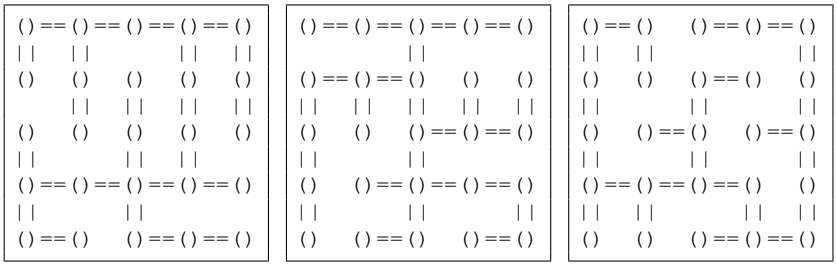

Fig. 8.4: After adding reachability constraint for each tile, the desired tree network appears. This program captures exactly the set of all perfect mazes of a given width.

So far, we have used only hard constraints: tiles have exactly one parent, and every tile must be linked to the root. We can express soft constraints in AnsProlog as well by defining optimization criteria. As an example of this for the primitive domain of mazes, let us suppose that vertical links in the maze are undesirable and that their use should be minimized. To accomplish this, the rules in Figure **??** define two ways of detecting a vertical link (an upward or downward parent), and the `minimize` statement tells the solver that solutions which use the least-possible count of vertical links are those that interest us. Although such statements are typically read as implying an *optimality constraint* (that only globally optimal solutions should be emitted, most answer set solvers will emit a series of answer sets they find along the way to finding one such optimal solution. By stopping the solver once it gets

close enough or runs for enough time, we can implement approximate optimization within this framework as well.

```
% soft style preferences: minimize vertical links
vertical(X,Y) :- parent(X,Y,0, 1).
vertical(X,Y) :- parent(X,Y,0,-1).
#minimize { vertical(X,Y) }.
```

Fig. 8.5: maze-bias.lp

Including the rules defining our bias against vertical links, a command like the following will allow us to sample maze designs that optimize our working evaluation criterion. Example outputs are show in Figure 8.6.

```
clingo maze-core.lp maze-reach.lp maze-bias.lp
```

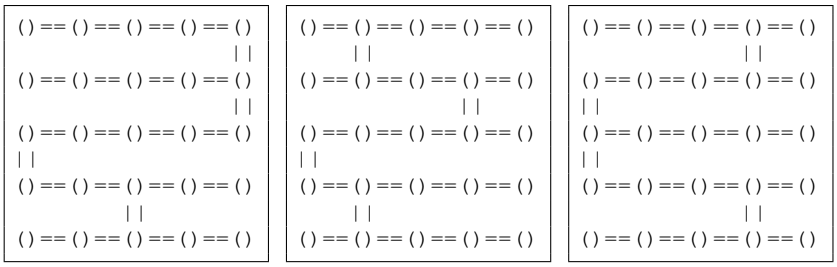

Fig. 8.6: Using the count of horizontal connections as an evaluation function, we can sample several alternative designs with a globally optimal score.

8.4 Playable dungeons

Mazes are an overly simplistic example of how to carry out content generation using ASP because they can be represented with only a single kind of choice. As a slightly richer example, this section looks at generating simple dungeon maps in which a few different types of sprites are stamped down onto the familiar two-dimensional grid.

Our task will be to design a level in which the player character starts in the top-left of the grid, finds a gem in the wall of the dungeon, carries it to a central altar when it is used to magically unlock the exit, and then walks out of that exit in the bottom right. We would like every generated level to be guaranteed to be solvable as well as have some basic control over the pacing of the level.

To begin, examine Figure 8.7. This program establishes a vocabulary of dimension values, tiles as value pairs, and adjacency between pairs of tiles. In the character

movement model we intend to capture, tiles that are one step up/down/left/right of each other are considered adjacent. A mathematical statement of this is that tile pairs with a coordinate distance of one are considered adjacent. The key part of this program is the choice rule that states that every tile has between zero and one sprites from the set of walls, the gem, and the altar. Because we know we only want to see maps with one gem and one altar, we immediate add integrity constraints that reject those maps for which there isn't exactly one of each.

```
#const width=10.

param("width",width).

dim(1..width).

tile((X,Y)) :- dim(X), dim(Y).

adj((X1,Y1),(X2,Y2)) :-
  tile((X1,Y1)),
  tile((X2,Y2)),
  #abs(X1-X2)+#abs(Y1-Y2) == 1.

start((1,1)).
finish((width,width)).

% tiles have at most one named sprite
0 { sprite(T,wall;gem;altar) } 1 :- tile(T).

% there is exactly one altar and one gem in the whole level
:- not 1 { sprite(T,altar) } 1.
:- not 1 { sprite(T,gem) } 1.
```

Fig. 8.7: level-core.lp

Starting with these core rules, commands like the following will generate outputs like those seen in Figure 8.8.

```
clingo level-core.lp --rand-freq=1
```

Our preliminary outputs hardly resemble interesting dungeon maps. There are many interesting maps lurking in the space we have defined, but they are hard to spot amongst the multitude of other combinations in the space. To zoom in on those maps of stylistic interest, we'll use a mixture of rules and integrity constraints to carve undesirable alteratives. A dungeon with only a sparse set of walls doesn't feel like a dungeon. A single wall sprite takes on the character of a wall when it is placed contiguously with other wall sprites. An altar should be surrounded by a few tiles of blank space, and gems should be well-attached to surrounding walls. Examine Figure 8.9 for a one-line encoding of each of these concerns.

With this addition, commands like the following can be used to sample stylistically-valid maps such as those in Figure 8.10. Note that while the levels look reasonable locally, they are still completely undesirable on the basis of them not supporting the kind of play we want—there's often not even a path from the gem to the altar, let alone from the entrance to the exit.

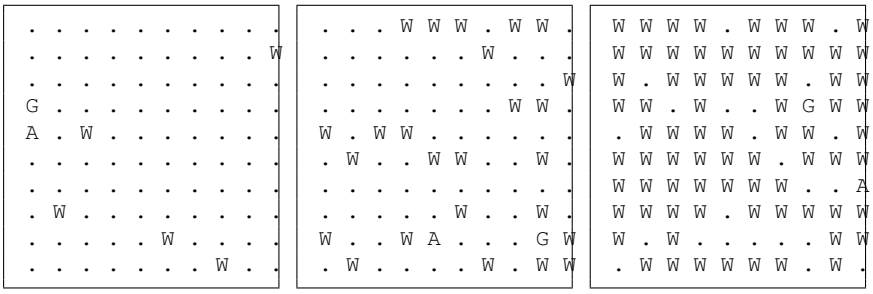

Fig. 8.8: A random result given rules that capture the basic representational vocabulary for the dungeon generation problem. A few walls (W) are present, along with exactly one gem (G) and one altar (A).

```
% style: at least half of the map has wall sprites
:- not (width*width)/2 { sprite(T,wall) }.

% style: altars have no surrounding walls for two steps
0 { sprite(T3,wall):adj(T1,T2):adj(T2,T3) } 0 :- sprite(T1,altar).

% style: altars have four adjacent tiles (not up against edge of map)
:- sprite(T1,altar), not 4 { adj(T1,T2) }.

% style: every wall has at least two neighboring walls (no isolated rocks and spurs)
2 { sprite(T2,wall):adj(T1,T2) } :- sprite(T1,wall).

% style: gems have at least three surrounding walls (they are stuck in a larger wall)
3 { sprite(T2,wall):adj(T1,T2) } :- sprite(T1,gem).
```

Fig. 8.9: level-style.lp

```
clingo level-core.lp level-style.lp
```

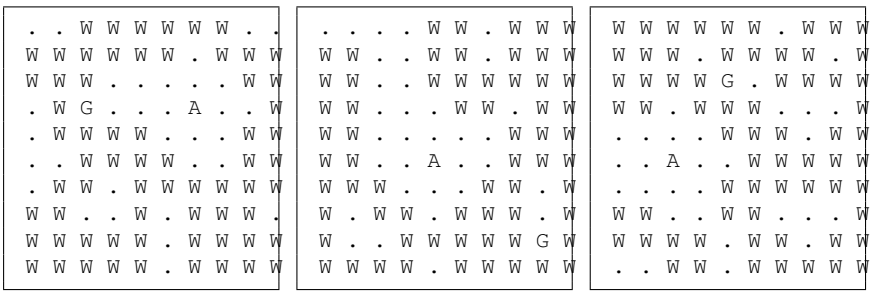

Fig. 8.10: After adding style constraints, there are many walls, the altar is surrounded by open space, and the gem is surrounded by walls on three sides. The fact that the gem is walled-off is a clue that we have not yet modeled a key contraint: the level must be playable.

The general strategy for ensuring we only generate playable maps is conceptually simple: generate a reference solution along with the level design. If a map contains a valid reference solution, we have a proof (by existence) that it is solvable. Even though we won't be representing the reference solution in our final output involving sprites on tiles, we can use the same language constructs as before to describe and constrain the space of possible solutions for a working map design.

Examine the rules in Figure 8.11. The key predicate is `touch(Tile,State)` which describes which tiles we expect the player character to touch in which gameplay state on the path to solving the level. To capture the sequence of picking up the gem, bringing it to the altar, and then exiting the level, we define three numbered states. The first rule tells us that the player will touch the start tile in state 1. From here, a series of choice rules say that touching one tile allows the player character to potentially touch any adjacent tile while retaining the same gameplay state. If the character is touching a tile containing the gem or the altar, they can transition to the next state in the sequence. The `completed` predicate holds (is true) if the player character touches the finish tile in final state (after placing the gem in the altar). By rejecting every logical world where `completed` is not true, we zoom in on the space of different ways of solving the level. No algorithm is needed to solve a level, only a definition of what it means for a set of touched tiles to constitute a valid solution.

```
% states :
% 1 --> initial
% 2 --> after picking up gem
% 3 --> after putting gem in altar

% you start in state 1
touch(T,1) :- start(T).

% possible navigation paths
{ touch(T2,2):adj(T1,T2) } :- touch(T1,1), sprite(T1,gem).
{ touch(T2,3):adj(T1,T2) } :- touch(T1,2), sprite(T1,altar).
{ touch(T2,S):adj(T1,T2) } :- touch(T1,S).

% you can't touch a wall in any state
:- sprite(T,wall), touch(T,S).

% the finish tile must be touched in state 3
completed :- finish(T), touch(T,3).
:- not completed.
```

Fig. 8.11: level-sim.lp

Although we could use the contents of Figure 8.11 as a stand-alone playability checker for human-designed dungeon maps, it is easy enough to simply use it as the same time as our previous map generator to construct a representation of the space of maps-with-valid-solutions. A command like the following yields guaranteed-playable, styled dungeon maps like those in Figure 8.12.

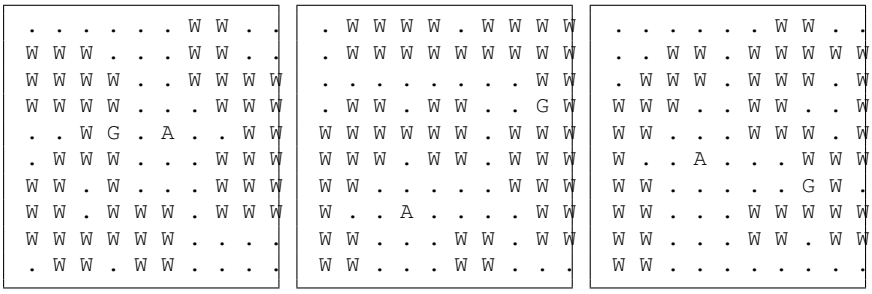

Fig. 8.12: After adding a simulation of player activity and placing constraints on the outcome, we now only see dungeon maps that have valid solution.

8.5 Constraining the entire space of play

The dungeon maps emerging from the previous section look about as good as sprite-on-grid maps containing two special objects and some walls can get. However, if we imagine playing through these maps, perhaps with simple arrow-key controls, there are still problems to resolve. In many of these maps, the task of placing the gem in the altar represents only a minor deviation from the more basic task of walking from the entrance to the exit of the dungeon. If the gem and the altar are to have any meaning for the gameplay of these maps, their placement and the arrangment of walls should conspire to make us explore the map, take detours from a start-to-finish speed-run, and backtrack through familiar areas. Although each of these concerns *could* be boiled down to a set of overlapping evaluation criteria in the form of statements about the relative distances between sprites, there is a better strategy.

If our goal is to get the player to work to progress through the sequence of gameplay states, we can state a much higher-level goal. The low-level design details of the map should somehow work to make sure the player character spends at least some amount of time walking around the map in each state. How this is accomplished (with with a network of rooms connected by indirect passages perhaps) is not immediately important to us. Our high-level design goal is most directly cast as a statement about the player's experience, not the form of any particular level. We'd like to demand that, across all possible solutions to a given level design, spending a minimum amount of time in each state is unavoidable. Interpreted logically, this is a statement that is quantified over the entire space of play.

Recent advances in the use of ASP for representing design spaces now allow the direct expression of this kind of design goal. Smith et al. [8] offer a small metaprogramming library that extends normal ASP with two special predicates. Their `__level_design(Atom)` and `__concept` predicates allow the expression of a query like this: starting with a given level design and reference solution, does the design space model allow another possibility in which identical choices are made for every predicated tagged with `__level_design(Atom)` and in which `__concept` is *not* true? If so, the tagged `__concept` condition must not be true for the entire

space of play for the given level design, and it should be rejected. The end result is a design space of level design with reference solutions in which `__concept` is an *unavoidable* condition across all alternative solutions to the level. As `__concept` could be any quantifier-free logical formula, this language extension allows the class of extended answer set programs to express any problem in the complexity class Σ_2^P (conventionally assumed to be much larger than the class NP).

Returning to the dungon map generation scenario, the rules in Figure 8.13 tag the `sprite(Tile,Name)` predicate as uniquely defining a level and the condition of touching at least `width`-many tiles in each of the three states as the desired unavoidable condition. A command like the following, which makes use of a special *disjunctive* answer set solver capable of solving the broader class of high-complexity problems, yields outputs like those shown in Figure 8.14.

```
clingo level-core.lp \
     level-style.lp \
     level-sim.lp \
     level-shortcuts.lp \
     --reify \
   | clingo - meta{,D,O,C}.lp -l \
   | clasp
```

```
% holding sprites constant, ensure every solution touches at least "width" many tiles in each state
__level_design(sprite(T,Name)) :- sprite(T,Name).
__concept :-
  width { touch(T,1) },
  width { touch(T,2) },
  width { touch(T,3) }.
```

Fig. 8.13: level-shortcuts.lp

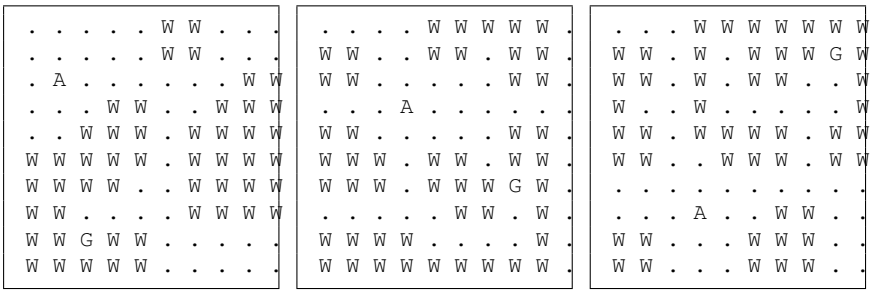

Fig. 8.14: Ensuring that the player cannot avoid spending a certain amount of time in each state has interesting emergent effects. Certain patterns that we might expect in human-crafted designs, such as the presence of hidden rooms of the main path through the level, occur naturally as the solver searches for the form of a level that gives rise to our requested function at a higher level.

Before we close this section, it is instructive to ask why the following simple rule doesn't achieve the same outcome. It would seem to prune away all those solutions in which the player doesn't spend enough time in each state.

```
:- width { touch(T,1) }, width { touch(T,2) }, width { touch(T,3) }.
```

This integrity constraint works like the "`:- not completed.`" rule from before. It works to make sure we only observe solutions (choices for `touch(Tile,State)`) that demonstrate an interesting property. Zooming in on solutions which complete the level doesn't preclude the player from choosing *not* to complete the level by simply wasting time before quitting. Likewise, zooming in on solutions in which the player wanders for a while doesn't imply that the wandering was inescapable. If we were to use this rule instead of the `__level_design`/`__concept` construction, we would most likely see many more examples like those from the previous section (Figure 8.12). In every example, it would be *possible* to wander and backtrack, but it would be unlikely to be actually required.

The idea of casting the most important properties of a level design as statements *quantified over the entire space of play* was first developed in the context of the educational puzzle game *Refraction*. What makes a given *Refraction* level desirable and relevant to its location in a larger level progression is strongly tied to which spatial and mathematical problem solving skills *must* be exercised to solve the level, even if the level admits many possible solutions. The idea of defining a level progression primarily on the basis of which concepts were required in which levels was the basis for one of the direct-manuipulation controls in the mixed-initiative progression design tool for *Refraction* [3].

Answer-set programming is not the only way to write down constraints over which kinds of gameplay must be possible (e.g. a level should be solvable) and which properties of gameplay are required (e.g. that a certain skill is exercised). The key strategy to follow is to generate not just a minimal description of the content of interest, but also a description of how the content can be used towards its desired function (such as a reference solution). Many interesting properties of a piece of game content are most naturally expressed as criteria that refer to how the content is used, as opposed to any direct properties of the content itself: a good level is one that produces desired gameplay when used together with a particular game's mechanics. Despite the fact that generating content under universally-quantified constraints maps to extremely high-complexity search and optimization problems, many of these problems can be solved, in practice, in short enough times to power interactive design tools and responsive online content generators embedded into games. The use of ASP as a generation technique provides a declarative modeling language that separates the designer of a content generator from the design of the search algorithms that will be applied to these complex problems.

8.6 Exercises: Elaborations on dungeon generation

1. Run each of the examples from the text on your own machine.
2. Add a new style constraint. Make sure you understand how it changes the maps that are generated.
3. Add a new type of tile sprite, call it `lava`, that can only be traversed after the player character has touched the special `boots` tiles.
4. Change the generator so that it can be initialized with a partial map, and the generator only fills in unconstrained tiles, in a way that fits style constraints.
5. Separate the playability checker from the rest of the dungeon generation program. Now apply it as a "machine playtester" [10] to point out playability flaws in levels you create yourself.
6. Design question: In the previous exercise, you took a playability checker whose initial job was to say "I wouldn't let a PCG system generate this level", and adapted it to say, "you, human designer, might have some flaws in this level you showed me". Are these really answering the same question? If you were writing a playability checker specifically to comment on human designers' levels, would you have written it differently? (See also Chapter 11.)

References

1. Baral, C.: Knowledge Representation, Reasoning, and Declarative Problem Solving. Cambridge University Press (2003)
2. Boenn, G., Brain, M., De Vos, M., Ffitch, J.: Automatic music composition using answer set programming. Theory and Practice of Logic Programming **11**(2–3), 397–427 (2011)
3. Butler, E., Smith, A.M., Liu, Y.E., Popovic, Z.: A mixed-initiative tool for designing level progressions in games. In: Proceedings of the 26th ACM Symposium on User Interface Software and Technology, pp. 377–386 (2013)
4. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Answer Set Solving in Practice. Morgan and Claypool (2012)
5. Gebser, M., Kaufmann, B., Kaminski, R., Ostrowski, M., Schaub, T., Schneider, M.: Potassco: The Potsdam answer set solving collection. AI Communications **24**(2), 107–124 (2011)
6. Horswill, I.D., Foged, L.: Fast procedural level population with playability constraints. In: Proceedings of the Eighth Artificial Intelligence and Interactive Digital Entertainment Conference, pp. 20–25 (2012)
7. Nelson, M.J., Mateas, M.: Recombinable game mechanics for automated design support. In: Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference, pp. 84–89 (2008)
8. Smith, A.M., Butler, E., Popović, Z.: Quantifying over play: Constraining undesirable solutions in puzzle design. In: Proceedings of the Eighth International Conference on the Foundations of Digital Games, pp. 221–228 (2013)
9. Smith, A.M., Mateas, M.: Answer set programming for procedural content generation: A design space approach. IEEE Transactions on Computational Intelligence and AI in Games **3**(3), 187–200 (2011)
10. Smith, A.M., Nelson, M.J., Mateas, M.: Computational support for play testing game sketches. In: Proceedings of the Fifth Artificial Intelligence and Interactive Digital Entertainment Conference, pp. 167–172 (2009)