Generating cellular puzzles with logic programs

Raphael Finkel raphael@cs.uky.edu
Wiktor Marek marek@cs.uky.edu
Mirek Truszczyński mirek@cs.uky.edu
Department of Computer Science, University of Kentucky

Abstract

We show how to characterize puzzles by logic programs and how to use those characterizations to build puzzles automatically. We can control the difficulty level of the puzzles by choosing how and when to invoke the logic program.

1. Introduction

Cellular puzzles take the form of **cells** whose **values** are constrained by **rules** involving groups of cells. The puzzle starts with **clues**, which are cells that already possess their value. It is the pleasant task of the puzzle **enthusiast** to solve the puzzle by entering values in all the cells in a way that satisfies the constraints. A **well-formed** puzzle has exactly one solution. To assist the enthusiast, the puzzle may contain a **hint sequence**, which is a sequence of cells that suggest the order in which the enthusiast should be able to complete the puzzle.

We consider Sudoku puzzles [TFE05] as a typical cellular puzzle. Figure 1 shows a sample Sudoku puzzle. The cells are arranged in a 9×9 grid subdivided into nine 3×3 **sections**. We say a set of 9 cells is **complete** if the numbers $1 \ldots 9$ appear exactly once in that set. The constraints that link the cells are that each row, column, and section is complete. The first hint, `bE`, directs our attention to a cell that must contain 2, because in its section, all other locations for 2 are excluded by being filled in with other numbers or because their row or column already has a 2.

This paper shows how we use logic programs to generate cellular puzzles of varying degrees of difficulty.

2. Generating a random complete solution

We begin by expressing the constraints in a logic formalism. For this paper, we present constraints using the Aspps [ET01] answer-set formalism, although many alternatives, such as smodels [NS97] and dlv [EFLP00] exist.

Given n as the width of a section (for us, 3), and m the width of the puzzle (for us, 9), the logic rules of Figure 2 represent the constraints.

Lines 1 and 2 introduce `numsmall` and `numlarge` as predicates true over the range $1 \ldots 3$ and $1 \ldots 9$, respectively. Lines 3, 4, and 5 give the signature of the ternary predicate `place` and the variables `I`, `J`, `N`, `K`, and `M`. We use `place(I,J,N)` to represent the fact that the value `N` is placed in cell `(I,J)`. Line 6 can be read "given a cell `(I,J)`, there is at least one and at most one (that is, exactly one) value `N` that is placed in that cell." Line 7 represents the constraint that there is exactly one row `I` for which any given value `N` appears in column `J`. Line 8 constrains columns in a similar way. Lines 9 and 10 introduce the section constraints. Given a section numbered `(K,M)` and a value `N`, there is exactly one cell `(I,J)` in that section that has that value.

When we run these rules as a logic program, we get a very large number of results. We pick the first result as a desired solution, relying on the randomization performed by Aspps. We also randomize the solution by randomly preloading values in a few cells. If our preloading is inconsistent with the constraints, we choose a different random preloading and repeat.

Puzzle

	A	B	C	D	E	F	G	H	I
a				8				2	3
b									4
c	6				7	5	9		
d						2			
e				9			8	3	
f		4				8		1	6
g		3	8		5				
h				7		1			
i	1	5		2				6	

Solution

	A	B	C	D	E	F	G	H	I
a	5	7	4	8	1	9	6	2	3
b	8	9	1	3	2	6	5	7	4
c	6	2	3	4	7	5	9	8	1
d	3	8	5	1	6	2	7	4	9
e	2	1	6	9	4	7	8	3	5
f	9	4	7	5	3	8	2	1	6
g	7	3	8	6	5	4	1	9	2
h	4	6	2	7	9	1	3	5	8
i	1	5	9	2	8	3	4	6	7

Hints

```
 1:bE  2:eF  3:eE  4:fE  5:hG  6:iF
7:fD  8:cH  9:cI 10:gG 11:fG 12:cB
13:eI 14:hH 15:bH 16:eA 17:hC 18:gI
19:gA 20:fC 21:fA 22:hA 23:aA 24:dC
25:bG 26:aG 27:aB 28:dA 29:dB 30:bA
31:iG 32:iC 33:bB 34:bF 35:gD 36:gF
37:cD 38:bD 39:cC 40:bC 41:eB 42:eC
43:hB 44:aE 45:dD 46:dE 47:aF 48:hE
49:iE 50:hI 51:iI 52:dG 53:dI 54:gH
55:dH 56:aC
```

Fig. 1. A 9×9 Sudoku puzzle, its solution, and hints

```
 1  numsmall(1..n).
 2  numlarge(1..m).

 3  pred place(numlarge, numlarge,
     numlarge).
 4  var numlarge I,J,N.
 5  var numsmall K,M.

 6  1 { place(I,J,N)[N] } 1.
 7  1 { place(I,J,N)[I] } 1.
 8  1 { place(I,J,N)[J] } 1.
 9  1 { place(I,J,N)[I,J]: I<=n*K:
     J<=n*M: n*(K-1) < I:
10     n*(M-1) < J} 1.
```

Fig. 2. Aspps rules for the Sudoku puzzle

```
removableClues := solution
preservedClues := ∅
while removableClues ≠ ∅ do
  victimClue :=
choose(removableClues)
  removableClues -:= {victimClue}
  if puzzleBad(removableClues
     + preservedClues) then
    preservedClues +:= {victimClue}
  end if
end while
clues := preservedClues
```

Fig. 3. Algorithm to reduce the clue set

3. Reducing the solution to a minimal set of clues

We take the solution as the initial set of clues and reduce the clue set until it is minimal by the algorithm of Figure 3. Each iteration randomly picks a single clue and tries to solve the puzzle without it. If the puzzle is now bad (we'll explain that shortly), the clue is added to the set of fixed clues, which must be preserved. Eventually, all the original clues are either removed or preserved; the preserved clues become the final set of clues.

The only complex part of this algorithm is the `puzzleBad` function, which determines whether a given set of clues makes a good puzzle. We try solving the puzzle with Aspps. The Aspps software has two stages, the **grounder** and the **solver**.

The grounder expands the rules of Figure 2 for all possible values of the variables, and then it uses unit propagation to determine as many facts as it can. It is able to look ahead one step, that is, try setting a fact to true and to false hoping to find a solution in only one of those cases. The solver employs full backtrack, using a heuristic to pick facts to assert and following the consequences. The following results can arise from applying Aspps.

- The puzzle has no solutions. This situation should never happen, because each iteration tests a less-constrained puzzle than the previous one.

- The grounder finds a solution without lookahead. The puzzle is "good" and the clue may be safely removed.

- The grounder finds the solution, but it needs lookahead. Depending on how difficult we wish to make the puzzle, we might call the puzzle either "good" or "bad". In practice, unless we are trying to generate a hard puzzle, we tell the grounder not to use lookahead. So when this situation arises, we call the puzzle "good".

- Aspps requires the full solver and backtrack to find a sole solution. Although the puzzle is well-formed, it is too hard for general enthusiasts. We call the puzzle "bad".

- The puzzle has multiple solutions. The missing clue is required to keep the puzzle well-formed. The puzzle is "bad".

These last two situations require the solver to distinguish. But we consider both to be "bad". Therefore, we don't ever call the solver. Either the grounder can solve the puzzle ("good") or it can't ("bad").

If we are trying to generate a hard puzzle, we let the grounder use one-step lookahead. We also apply one more test after reaching the final set of clues: Can the grounder solve the final set of clues without using lookahead? If so, then the puzzle isn't hard enough, but removing any clues would make it ill-formed or too hard. In this case, we reject the puzzle completely and start afresh.

size	initial time	iteration time	mem
16×16	0.2	0.10	4
25×25	0.6 – 2.7	0.14	6
36×36	4.8 – 54	0.24	14
49×49	22 – 300	0.49	30

Fig. 4. Time (seconds) and memory (MB) requirements for generating Sudoku puzzles

4. Generating a hint sequence

In a cellular puzzle, most atoms turn out to be false. For example, in the puzzle of Figure 1 (page 2), it turns out that `place(1,1,5)` is true. Therefore the grounder at some point also derives that `place(1,1,1)` is false, as is `place(1,1,2)` and all the other related values. The positive (true) instances of `place` are the interesting ones. By examining the grounder's log, we can determine the order in which it discovers positive atoms by unit propagation. That list begins with the clues themselves. The rest of the positive instances of `place` in the log form the hint sequence.

5. Performance

The logic-program approach to generating cellular puzzles is remarkably efficient in programmer time. Each puzzle type requires only a few lines of Aspps code. The algorithm of Figure 3 is encoded in about 500 lines of Perl [WS90], much of which is devoted to generating formatted puzzle and hint output.

Given the Aspps rules, generating a puzzle has two phases: Finding the solution and reducing the clues. The time needed for the first phase depends, of course, on the complexity of the constraints and the size of the puzzle. On a 3GHz Pentium 4 running Linux, we accomplish the first phase for Sudoku puzzles of various sizes in time and memory shown in Figure 4. The time for the first phase has a large variance, especially for larger puzzles. Figure 4 also shows the time for each iteration in the second phase. This value has much smaller variance.

6. Puzzle types

We have investigated several cellular puzzle types.

- Sudoku. These are $n^2 \times n^2$ squares with n^2 sections containing $n \times n$ cells, where each row, column, and section are complete.
- Diagonal Sudoku. These puzzles are the same as Sudoku, but the two diagonals are also complete. Cells (i, i) constitute the first diagonal, and $(i, n^2 + 1 - i)$ constitute the other. We show such a puzzle in Figure 5.
- MultiFour. This puzzle is our own invention. It has $4n \times 4n$ cells. Again, each row and column is complete, but now a set of cells is complete if every value $1 \ldots 4$ occurs exactly n times. We show such a puzzle in Figure 6. For $n = 2$, we can add the extra constraint that every 2×2 section has each of $1 \ldots 4$ exactly once. This constraint is apparently not satisfiable for larger n.
- MultiSpot. These puzzles are our own invention. They have $n \times n$ cells, for $n = 5, 6, 7, 8, 9$. Each row and column is complete, as is each irregularly-shaped region. We show such a puzzle for $n = 7$ in Figure 7.

All these puzzles use only completeness constraints. We could also impose numerical constraints, such as requiring that particular cells have values adding to some value. One form of numeric cellular puzzle is arranged as a planar graph of a fixed shape. Each region and each edge is a cell that must have an integer value. The value on each edge must be the sum of the values of the regions it borders. It remains to be seen if such puzzles are attractive to enthusiasts. It is certainly easy to generate them.

Puzzle

	A	B	C	D	E	F	G	H	I
a							1		4
b						2			3
c				9	5		2		
d									1
e									
f		9						8	
g	1			2	9	3			
h						8	6	7	
i				7	4				

Solution

	A	B	C	D	E	F	G	H	I
a	2	8	5	6	3	7	1	9	4
b	9	1	6	4	8	2	7	5	3
c	3	7	4	9	5	1	2	6	8
d	6	4	8	3	7	9	5	2	1
e	5	2	1	8	6	4	9	3	7
f	7	9	3	1	2	5	4	8	6
g	1	6	7	2	9	3	8	4	5
h	4	3	9	5	1	8	6	7	2
i	8	5	2	7	4	6	3	1	9

Fig. 5. A 9×9 diagonal Sudoku puzzle

7. References

[EFLP00] Thomas Eiter, Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Declarative problem-solving in DLV. In Jack Minker, editor, *Logic-Based Artificial Intelligence*, pages 79–103. Kluwer Academic Publishers, Dordrecht, 2000.

[ET01] D. East and M. Truszczyński. *aspps* — an implementation of answer-set programming with propositional schemata.

Puzzle

	A	B	C	D	E	F	G	H	I	J	K	L
a	3					1	3		1	4	4	
b		1		2			4	3	4	2	3	
c	1	1	4	4					3	1		
d		1	1	2				2				2
e	2		1	1		1		2	4	4		4
f	2		2	1		4		1		4		
g		3			4		3					2
h	4		4		2			2		1	2	1
i		4			3	2		3				1
j	3	3		2	4		2			2		3
k							2	1	3	2		2
l	4	3		3	2	2					4	1

Solution

	A	B	C	D	E	F	G	H	I	J	K	L
a	3	2	2	3	2	1	3	1	1	4	4	4
b	2	1	3	2	1	1	4	3	4	2	3	4
c	1	1	4	4	4	2	2	3	3	1	2	3
d	3	1	1	2	1	3	4	2	4	3	4	2
e	2	2	1	1	3	1	3	2	4	4	3	4
f	2	2	2	1	3	4	4	1	1	4	3	3
g	1	3	2	1	4	4	3	4	2	3	1	2
h	4	4	4	3	2	3	1	2	3	1	2	1
i	1	4	4	4	3	2	1	3	2	3	2	1
j	3	3	1	2	4	4	2	4	1	2	1	3
k	4	4	3	4	1	3	2	1	3	2	1	2
l	4	3	3	3	2	2	1	4	2	1	4	1

Fig. 6. An $n = 3$ MultiFour puzzle

Puzzle

	A	B	C	D	E	F	G
a			4	3			
b		7					
c	3	6	1		4		
d			5				3
e							
f	7					1	
g							

Solution

	A	B	C	D	E	F	G
a	2	5	4	3	7	6	1
b	1	7	6	5	2	3	4
c	3	6	1	7	4	2	5
d	4	2	5	1	6	7	3
e	6	3	7	4	1	5	2
f	7	4	3	2	5	1	6
g	5	1	2	6	3	4	7

Fig. 7. An $n = 7$ MultiSpot puzzle

In *Proceedings of Logic Programming and Nonmonotonic Reasoning Conference, LPNMR 2001*, volume 2173, pages 402–405. Lecture Notes in Artificial Intelligence, Springer Verlag, 2001.

[NS97] I. Niemelä and P. Simons. Smodels — an implementation of the stable model and well-foundd semantics for normal logic programs. In *Logic Programming and Nonmonotonic Reasoning (the 4th International Conference, Dagstuhl, Germany, 1997)*, volume 1265 of *Lecture Notes in Computer Science*, pages 420–429. Springer-Verlag, 1997.

[TFE05] Wikipedia: The Free Encyclopedia. Sudoku, 2005. `http://en.wikipedia.org/wiki/Sudoku`.

[WS90] L. Wall and R. L. Schwartz. *Programming Perl*. O'Reilly and Associates, 1990.