CS 550 Programming Languages Project Writeup
Due: December 10, 2006 Aaron Clauset

Sudoku solvers

My project was to implement a sudoku solver, and within that larger goal, I implemented two versions. The first is a pure prolog implementation, while the second uses Ciao's finite domain (constraint programming) framework. The general structure of each implementation is the same; thus, only when necessary while I distinguish between the two, i.e., when it is relevant that one is a finite domain-based solver while the other is a prolog-based solver.

Introduction

The traditional sudoku game is played on a 9×9 board and has the following rules simple rules: each row, column and sub-matrix must be a permutation of the numbers $1 \ldots 9$. The sub-matrices are simply the 3×3 matrices that tile the larger board. In the generalization of sudoku, the board is $n^2 \times n^2$ in size, the numbers run from $1 \ldots n^2$ and the sub-matrices are of size $n \times n$. An instance of a sudoku puzzle is a partially completed board, and a solution is the completed board, in which each entry respects the constraints. Figure 1 illustrates a sudoku puzzle and a satisfactory solution.

My implementation solves the general sudoku case, and in the files themselves are examples of 4×4, 9×9 and 16×16 puzzles, with solutions. Specifically, I implemented a program in ciao that, when given a partially completed sudoku puzzle, will return a solved version of it. Variations of my program (also provided) allow the user to ask for only a single solution or all solutions, for a given input.

As a brief aside, sudoku has garnered some interest from the mathematics and theory of computer science communities. A Latin square (also called an Euler's square) has the same structure as a sudoku puzzle, except that the constraint on the sub-matrices is absent. The completion of Latin squares is known to be an NP-complete problem, through a connection with quasigroups [1]. There is also apparently a reduction of the general case of sudoku to the Latin Squares completion problem, thus putting sudoku completion in the NPC complexity class [4]. Other work on sudoku has focused on its constraint-satisfaction properties [3], in which different propagation schemes for sudoku were considered and a technique for deriving a solution without searching was proposed. Sudoku can also be considered as a SAT problem [2], which presents the potential to apply a wide variety of well-studied techniques such as unit propagation to this domain.

One interesting question about sudoku puzzles in general concerns the number valid solutions for a number k of given values in the partially completed puzzle. Empirically, the transition from multiple solutions to a single solution seems to begin around $k = 16$, but its precise value remains an open question. Presumably, as with most constraint satisfaction problems such as 3-SAT, there is a phase transition, for random puzzles, from almost-always-satisfiable to almost-always-unsatisfiable, but I'm unaware of any extension of these questions to sudoku.

	6			5			2	
			3				9	
7			6				1	
		6		3		4		
		4		7		1		
		5		9		8		
	4				1			6
	3				8			
	2			4			5	

(a)

8	6	1	4	5	9	7	2	3
4	5	2	3	1	7	6	9	8
7	9	3	6	8	2	5	1	4
2	1	6	8	3	5	4	7	9
9	8	4	2	7	6	1	3	5
3	7	5	1	9	4	8	6	2
5	4	7	9	2	1	3	8	6
1	3	9	5	6	8	2	4	7
6	2	8	7	4	3	9	5	1

(b)

Figure 1: (a) An example 9×9 sudoku puzzle, in which 27 entries are given. (b) A solution to the puzzle, computed using the `sudoku` solver described here.

Usage

The main function call is `sudoku(P)`, where `P` is the partially solved sudoku puzzle. The function's output is a pretty-printed version of `P`, in which any variable or unknown values have been replaced with integers that satisfy the sudoku constraints.

`sudoku/1`: PREDICATE
Usage: `sudoku(Puzzle)`
– *Description*: `Puzzle` is a list of numbers, of length n^4, in which variables in the list correspond to unknown values in the sudoku puzzle. For example,

```
?- sudoku([_,6,_,_,5,_,_,2,_,_,_,_,3,_,_,_,9,_,7,_,_,6,_,_,_,1,_,_,_,6,_,3,_,4,_,_,_,
_,4,_,7,_,1,_,_,_,_,_,5,_,9,_,8,_,_,_,4,_,_,_,1,_,_,6,_,3,_,_,_,8,_,_,_,_,2,_,_,4,_,_,5,_]).
```

corresponds to the function call for the puzzle shown in Figure 1, and produces the following as output.

```
8 6 1   4 5 9   7 2 3
4 5 2   3 1 7   6 9 8
7 9 3   6 8 2   5 1 4

2 1 6   8 3 5   4 7 9
9 8 4   2 7 6   1 3 5
3 7 5   1 9 4   8 6 2

5 4 7   9 2 1   3 8 6
1 3 9   5 6 8   2 4 7
6 2 8   7 4 3   9 5 1

yes
```

`sudoku_noprint/1`**:** PREDICATE
Usage: `sudoku_noprint(Puzzle)`
– *Description*: `Puzzle` is as above; in this version, no output is written to the top-level of the interpreter. This predicate is intended for use with the `findall` predicate, for counting the number of solutions to a particular instance. For example,

```
?- findall(X,sudoku_noprint([_X,6,_,_,5,_,_,2,_,_,_,_,3,_,_,_,9,_,7,_,_,6,_,_,_,1,
_,_,_,6,_,3,_,4,_,_,_,_,4,_,7,_,1,_,_,_,_,5,_,9,_,8,_,_,_,4,_,_,_,1,_,_,6,_,3,_,_,_,8,
_,_,_,_,2,_,_,4,_,_,5,_]),_L), length(_L,N).

N = 1 ?  ;

no
```

`sudoku1/1`**:** PREDICATE
Usage: `sudoku1(Puzzle)`
– *Description*: `Puzzle` is as above; in this version, at most a single solution is given. This predicate is only available in the prolog implementation, `aaron_sudoku_pl`.

Extensive examples can be found within the project files themselves, including various potentially interesting queries using these predicates.

Internals

As mentioned above, there are two modules in my project: `aaron_sudoku_pl` implements the prolog version of the sudoku solver, while `aaron_sudoku_fd` implements the finite domain version. Each is a stand-alone module that implements everything necessary for supporting the predicates described above (except where noted). Because my application is quite specific, I'll briefly discuss "the internals" of the solver by discussing its computation in the context of its internal predicates. The general program structure is as follows:

```
sudoku(P) :-
  (initialize variables or constraints) ,
  check_rows(X,P),
  check_cols(X,P),
  check_blks(X,P),
  (write to top-level, if necessary) .
```

In the finite-domain version, the initialization step extracts the value of n from the list and establishes the initial constraints on the variables, i.e., that each is an integer from $[1 \dots n^2]$. In the prolog version, the initialization step constructs the list $[1 \dots n^2]$. The respective `check_*` functions in both work roughly the same – they traverse the list `P` to extract the n^2 vectors, each of length n^2, that contain the elements of each row, column or sub-matrix, and then enforce the requirement that it be a permutation of the integers $[1 \dots n^2]$. In order to account for the variable length of the input list `P`, the `check_cols` and `check_blks` functions use a complicated indexing scheme in order to determine how many initial elements of `P` to discard between copying a value or variable into the output vector.

Discussion

The most surprising difference in my two implementations is their respective running times. In the prolog version, even nearly completed, e.g., perhaps only 25 unknowns, 9×9 puzzles require more than a reasonable amount of time (I let it run for almost a day). In contrast, the finite domain version can solve a 16×16 with 154 unknowns in a few minutes. The efficiency of the finite domain version is particular surprising given that Ciao does not have a very efficient fd-solver implementation.

In general, the differences between the two implementations are quite small. The prolog version uses a more obtuse method for enforcing the constraint that rows, columns and sub-matrices are proper permutations – specifically, it passes a sorted version of each to a predicate `permutation` – while the finite-domain version uses the constraint-programming function `all_different` to accomplish the same goal. The bulk of the internals of the solvers actually concern themselves with traversing the list data structure to extract lists that correspond to rows, columns and sub-matrices. Because my solver works with the general puzzle with $n^2 \times n^2$ entries, these internal predicates are somewhat complicated.

Finally, an interesting extension of this work might be to try to tackle the question of where the transition from multiple solutions to a single solution occurs. It should be relatively easy to write a simple program that generates sudoku instances with k known values and search over these for the minimum k such that `sudoku` finds a single solution.

References

[1] C. Gomes and D. Shmoys. Completing Quasigroups or Latin Squares: A Structured Graph Coloring Problem. In *Proc. of the Comp. Symp. on Graph Coloring and Extensions* (2002).

[2] I. Lynce and J. Ouaknine. Sudoku as a SAT problem. In *Proc. of the 9th International Symposium on Artificial Intelligence and Mathematics* (January 2006).

[3] H. Simonis. Sudokku as a constraint problem. In *Proc. of the CP Workshop on Modeling and Reformulating Constraint Satisfaction Problems*, p13–27 (October, 2005).

[4] T. Yato and T. Seta. Complexity and completeness of finding another solution and its application to puzzles. In *Proc. of the National Meeting of the Info. Processing Soc. of Japan (IPSJ)* (2002).