Vol. 7, No. 3, May 2007, pp. 228–237
ISSN 1532-0545 | 07 | 0703 | 0228
INFORMS Transactions on Education
informs
DOI 10.1287/ited.7.3.228

Advanced Lessons on the Craft of Optimization Modeling Based on Modeling Sudoku in Excel

Rasmus A. Rasmussen
Molde University College, 6402 Molde, Norway, Rasmus.Rasmussen@hiMolde.no
Howard J. Weiss
Department of Management Science/Operations Management, Fox School of Business, Temple University, Philadelphia, Pennsylvania 19122, USA, hweiss@temple.edu

In a previous paper we (Weiss and Rasmussen 2007) demonstrated lessons that can be learned by formulating Sudoku in Excel using Solver's standard tools. In this paper we use the advanced tools and solver engines available with the Premium Solver Platform in order to demonstrate more sophisticated lessons regarding optimization modeling. Optimization modeling is a skill developed by building and testing alternative formulations for new problems. This paper gives advanced undergraduate and graduate students an opportunity to develop the craft of optimization modeling, by presenting the construction of two new alternatives for modeling Sudoku in Excel. We do not present these models because they are more efficient than the previous models; in fact one of them does not work well at all. The main reason is to highlight strengths and weaknesses when using different modeling approaches, and to display some of the additional modeling capabilities available using the Premium Solver Platform rather than the standard Solver.

1. Introduction

Recently, in this journal, Chlond (2005) presented an integer programming formulation of the popular puzzle, Sudoku. More recently, we (Weiss and Rasmussen 2007) have demonstrated several lessons that students can learn by formulating Sudoku puzzles in Excel. Essentially, we reduced the number of variable subscripts from five to three so that we could easily formulate the problem in Excel. In this paper, we will take advantage of some of Solver's advanced features to reduce the problem size and to reduce the problem dimension to two. The work in this paper requires the use of the Premium Solver Platform and some plug-in Solver engines. Trial versions (15 days) are available through Frontline Systems, Inc http://www.solver.com/.

In the next section we present a model that takes advantage of Solver's more advanced features. Then we present a two dimensional model that is based on the literal requirements of Sudoku and uses only two subscripts which makes it relatively straightforward to formulate in Excel. While it is easy to create the 2-dimensional model in Excel, this model creates difficulties for Solver due to the combination of non-smooth functions and problem size. The difficulties afford the instructor the opportunity to discuss the different solution engines available in Solver.

2. Reducing the Number of Variables

The popular Sudoku puzzle is a 9 by 9 puzzle as exemplified by Chlond's example displayed in Table 1.

The formulation in both Chlond (2005) and Weiss and Rasmussen (2007) required n^3 or 729 variables. We begin our discussion by reducing the number of variables.

The previous model was 3-dimensional consisting of n^3 *binary* variables. A more direct formulation uses $n \times n$ *integer* variables. An Excel implementation of such a formulation is displayed in Figure 1.

The problem is to fill the green grid (B14:J22, the 9×9 decision variables) so that the 9 rows all contain the integers 1 thru 9, and the same for the 9 columns, and also for the 9 sub-grids. Finally the green solution grid must contain the initial numbers given by the specific puzzle in the white grid (B3:J11).

Lesson 1: Alldifferent Constraints Are Efficient in Combinatorial Problems

Table 2 lists the 5 types of constraints (relations) that can be used or programmed in the basic Solver. (See Anonymous 2006b.)

The Premium Solver contains other constraint types including the "alldifferent" constraint. The Premium Solver Platform User Guide (Anonymous 2006a, p. 17) defines this constraint as follows.

Table 1 A Sample 9 by 9

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

The alldifferent constraint specifies that at the solution, each integer variable in the group must have a value that is different from all the others. Hence, the variables in the group form an ordering, or permutation, of integers.

Alldifferent constraints can only be applied to decision variables. If cells A1:An are defined as Changing Variable Cells, then the constraint: "**A1:An = alldifferent**" will act as 4 constraint sets:

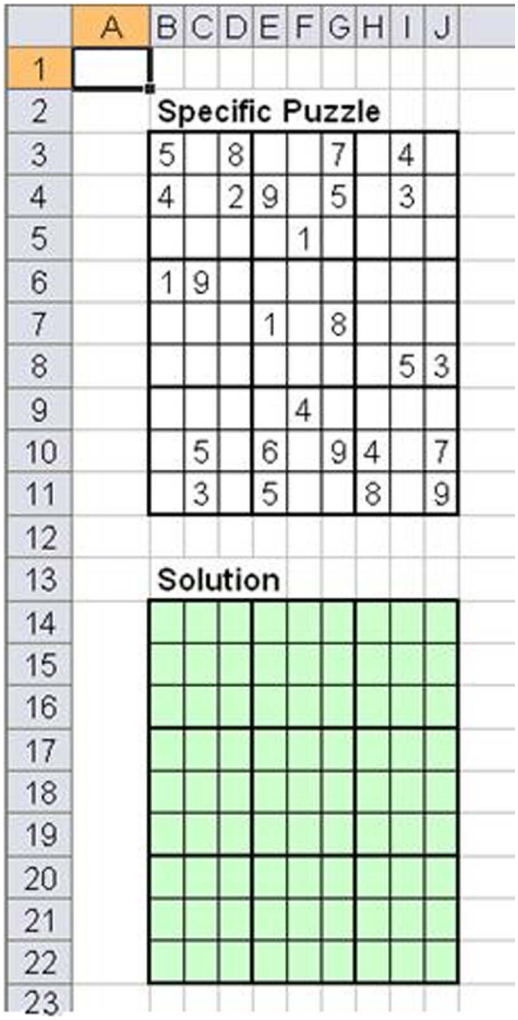
- A1:An = integer
- A1:An >= 1
- A1:An <= n
- All cells A1:An must be different (no pair of cells are allowed to be equal). This is the same as requiring that all integers 1 through n must be used (at least or exactly) once.

Sudoku seems to be a perfect case for the alldifferent constraint in Solver. If we apply such a constraint to the row B14:J14 Solver will automatically determine the size to be 9 cells, all restricted to integers between 1 and 9, and all cells have to be different (no pair of cells being equal). Thus we could apply the alldifferent constraint to the 9 rows, 9 columns, and 9 sub-grids.

However, Solver has a restriction that a cell can only be part of one alldifferent constraint, and in this layout every cell would be part of 3 alldifferent constraints. To overcome this limitation we simply copy the original grid two times so we end up with 3 separate grids as displayed in Figure 2. We use one grid for the row constraints, one grid for the columns constraints, and one grid for the sub-grid constraints. Even though in our previous paper we have promoted the advantages of keeping variables contiguous for easier entry into Solver, we have used columns K and U to separate the three tables for ease of discussion.

As the alldifferent constraints can only be applied to decision variables, we have to define all 3 grids as decision variables, for a total of $3n^2$ integer variables. Technically, there are three dimensions for the variables (row, column, and table) but since the three

Figure 1 An Excel Spreadsheet for the 9×9 Sudoku Puzzle Formulated as a 2-Dimensional Problem



tables must be equal to each other, for all intents and purposes, this is a two-dimensional problem. For the 9 by 9 puzzle this representation will have 243 (3×81) integer variables rather than the 729 (9×81) binary variables in the models by Chlond and in our previous paper.

Of course, all 3 grids must be equal since each represents the same solution to the puzzle. The last 2 grids are only added to implement some of the alldifferent constraints originally meant for the first grid. To ensure that the 3 grids are identical we add the constraints: B14:J22 = L14:T22 and B14:J22 = V14:AD22.

At this point we have the basic Sudoku requirements implemented and need only to implement the constraints that will require the solution in rows B14:J22 (and L14:T22 and V14:AD22) to include the specific puzzle in B3:J11. We need to set up an extra table on the spreadsheet to do so but we must be very careful for the following reason.

Table 2 Basic Solver Constraint Types

SolverAdd (CellRef, Relation, FormulaText)

- **CellRef** references one or more cells that form the left side of the constraint.
- **Relation** is the arithmetic relationship between the left and the right sides of a constraint.
- **Relation** can be a value between 1 and 5 as in the following example:
  - The value 1 is less than or equal to (<=).
  - The value 2 is equal to (=).
  - The value 3 is greater than or equal to (>=).
  - The value 4 is an integer.
  - The value 5 is the binary (a value of zero or one).
- **FormulaText** references one or more cells that form the right side of the constraint

Lesson 2: $\mathbf{x} = \mathbf{c}$ Is Not Always the Same as $\mathbf{c} = \mathbf{x}$

If **x** is a decision variable and **c** is a constant, then a constraint like $\mathbf{x} = \mathbf{c}$ (or $\mathbf{x} <= \mathbf{c}$ or $\mathbf{x} >= \mathbf{c}$) would be treated as a bound on **x** by Solver. When **x** is an alldifferent variable this will result in an error message, because Solver does not accept additional bounds on alldifferent variables.

However, a constraint like $\mathbf{c} = \mathbf{x}$ will be replaced by $\mathbf{c} - \mathbf{x} = \mathbf{0}$ by Solver. This applies to all RHS involving decision variables, and will also be evident in a Sensitivity Report for Linear Programming, where the value of the RHS of the constraint $\mathbf{c} = \mathbf{x}$ will be displayed as **0**. Such a constraint is not treated as a bound on the decision variable **x** and is therefore accepted, even when **x** is an alldifferent variable.

For example, adding the constraint B14 >= B3 will lead to the message "Variable bounds conflict in binary or alldifferent constraint" whereas adding the constraint B3 <= B14 will cause no problems.

The complete model is in the spreadsheet named "Combinatorial" which we display in Figure 3.

To make sure that the puzzle fits the initial numbers the formula: B25 =IF(ISNUMBER(B3);B3-B14;0) is copied to B25:J33. This formula checks if there is an initial number for the specific puzzle, and if so computes the difference between the initial number

Figure 2 A Formulation with $n \times n \times 3$ Integer Decision Variables

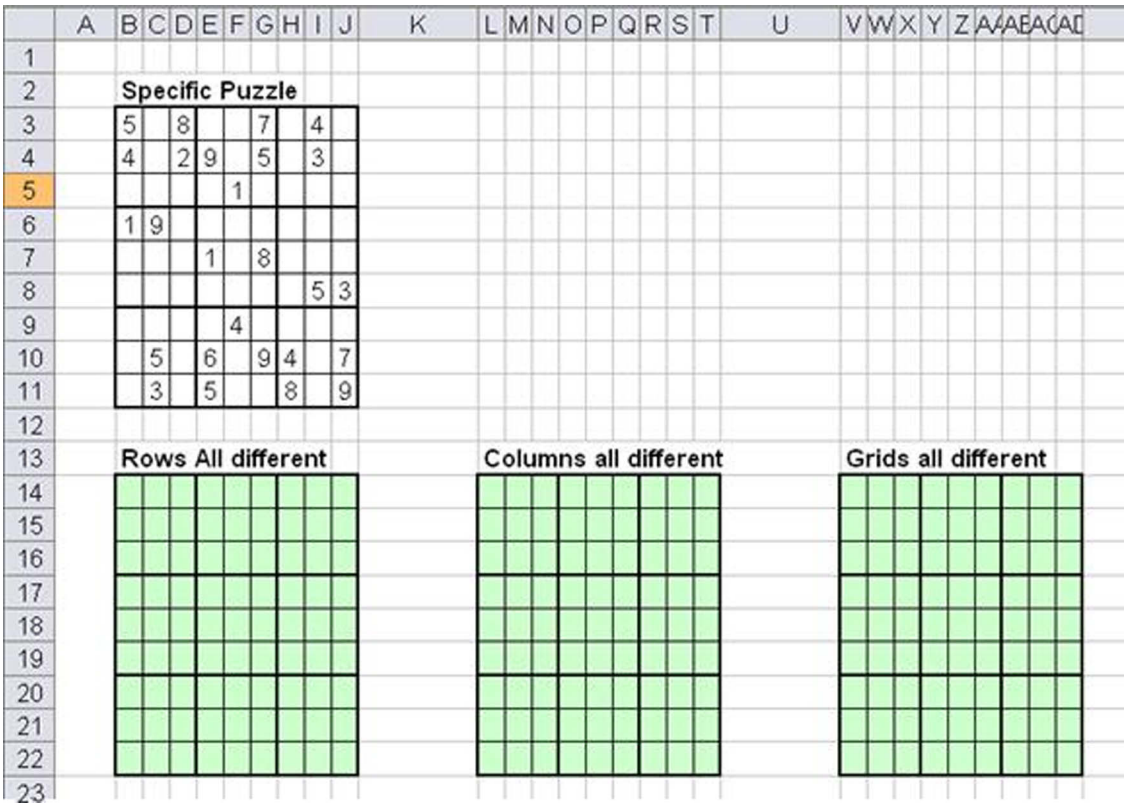
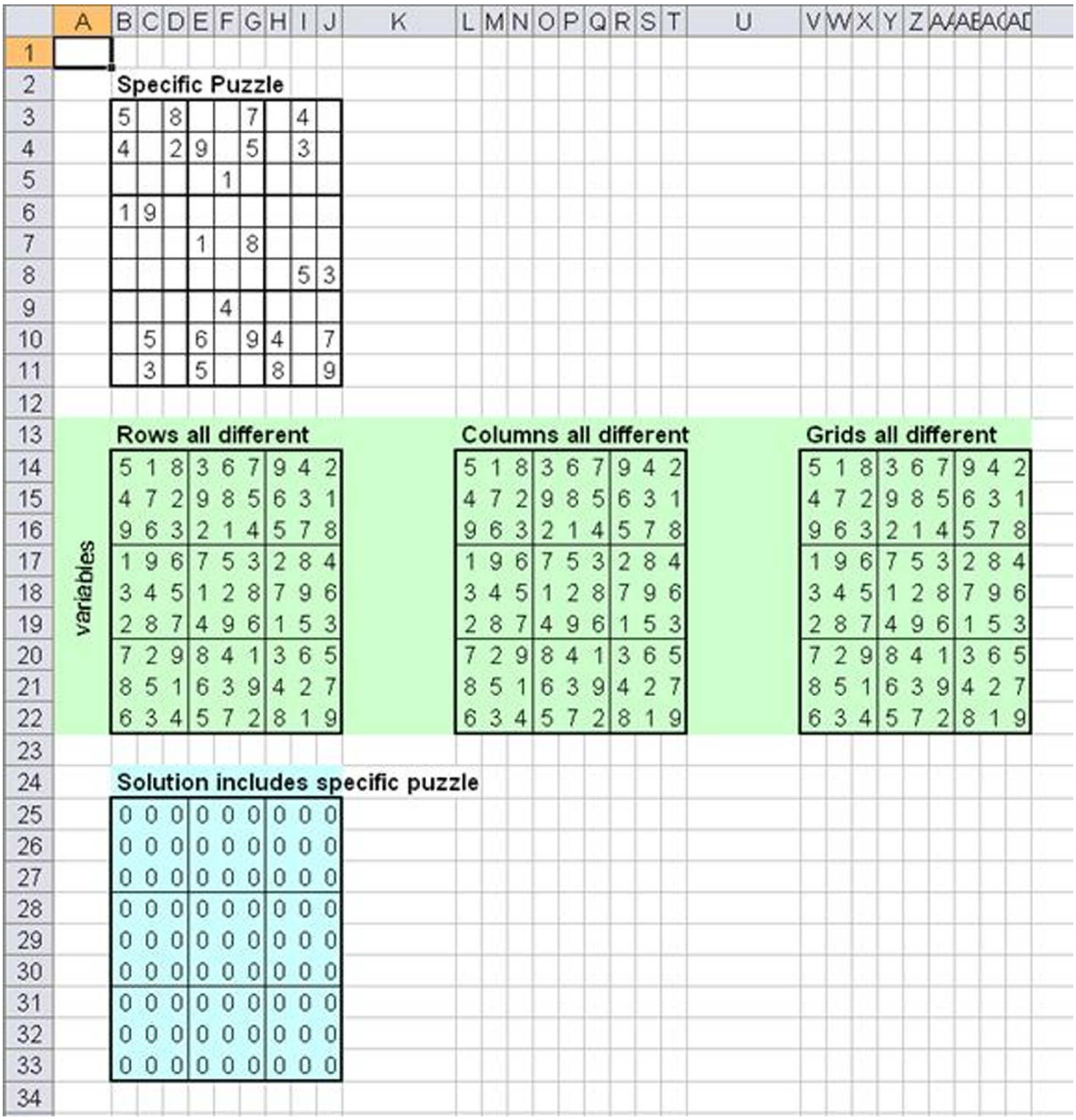


Figure 3 The Complete Formulation with $n \times n \times 3$ Integer Variables



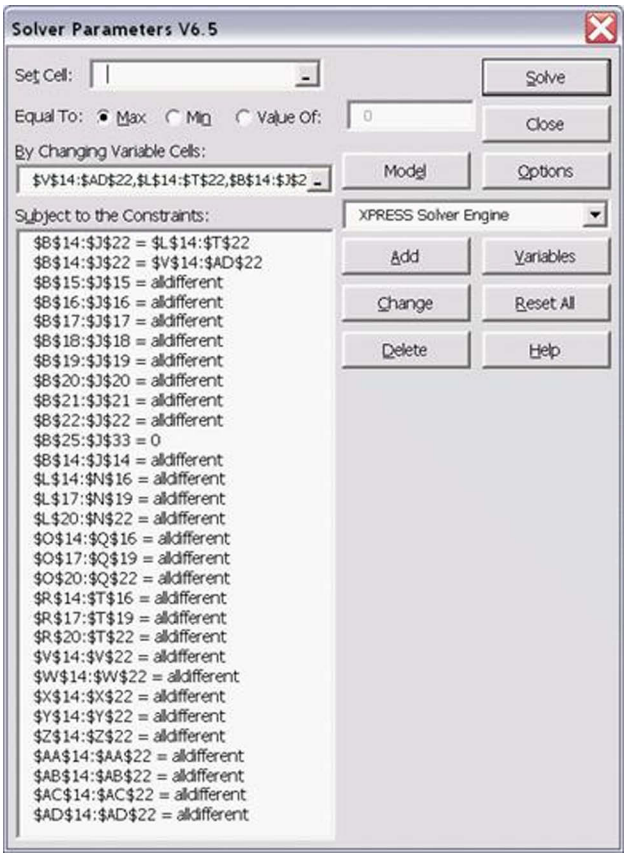
and the number selected by Solver. If no initial number is given the formula returns 0. Thus the constraint B25:J33 = 0 will only be satisfied if Solver has chosen the same numbers as the initial numbers given for the specific puzzle. While in some cases an IF function may lead to a non-linear model, in this case, the cells with IF simply act as constants in the linear programming model and do not cause any non-linearity problems. This is because the IF test only depends on the constant input data.

The constraints in Solver include 9 alldifferent constraints for the 9 rows, 9 alldifferent constraints for the 9 columns, and 9 alldifferent constraints for the 9 sub-grids. Upper and lower limits are automatically taken care of in the alldifferent constraints. In addition we add the constraint making the numbers chosen by Solver equal the initial numbers. We also add the two constraints forcing the two extra grids of decision variables equal to the initial grid. In general a total of $n + n + n + 1 + 2 = 3n + 3$ constraint lines are

Table 3 Excel Formulas for Checking the Solutions

Cell	Formula	Copied to
B25	=IF(ISNUMBER(B3);B3-B14;0)	B25:J33

Figure 4 Solver Settings for the Formulation with $n^2 \times 3$ Integer Variables



required for this linear formulation of Sudoku, and $3n^2$ integer variables are needed. The Solver settings are displayed in Figure 4.

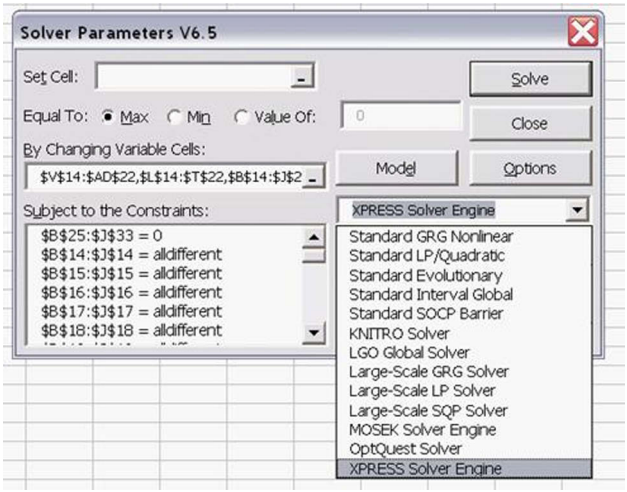
Please notice the dropdown box in Solver that reads "XPRESS Solver Engine." There are multiple solver engines available for Solver. We show these engines in Figure 5. Most Solver Engines accept multiple alldifferent constraints except OptQuest which accepts only one set of alldifferent constraints.

With this formulation we do not have to use any code for adding constraints as we had done in our other paper. The constraints are the same regardless of the specific puzzle! The student can simply enter the new puzzle in Cells B3:J11 and use the Excel menu commands of Tools, Solver, Solve to find the solution to the puzzle.

Lesson 3: Select the Proper Solver Engine with Care

For version 6.5 of the Premium Solver Platform, the XPRESS Solver Engine finds a feasible solution in a few seconds. Most of the other LP Solver engines have trouble finding a feasible solution to this formulation. However, in the recently released version 7.0 of the Premium Solver Platform, all LP Solver engines (Standard LP, Large-scale LP, and XPRESS) find a feasible solution quickly. Unfortunately some other engines normally capable of solving ILP problems fail (Large-scale SQP, KNITRO, and MOSEK) due to the fact that their branch and bound code is not as sophisticated as the branch & bound code of the LP Solver engines.

Figure 5 Solver Engines



Lesson 4: Formulation Matters

If we run these solver engines on the corresponding 9×9 model in our previous paper, all of them succeed. Obviously the formulation using binary variables is easier to solve than the formulation using general integer variables presented above.

3. Another 2-Dimensional Formulation

The instructions at www.sudoku.com http://www.sudoku.com/ are, "Fill in the grid so that every row, every column, and every 3×3 box contains the digits 1 through 9." Thus, the problem can be formulated as follows. Let $x_{i,j}$ be the integer that should be in cell (i, j) in the puzzle. The problem statement quoted above indicates that there are three types of constraints that must be considered.

All integers 1 through n must be used in each row.

All integers 1 through n must be used in each column.

All integers 1 through n must be used in each sub-grid.

Each condition generates n^2 constraints and therefore the total number of constraints is $3n$. Obviously, there are n^2 integer variables.

Our previous model started as a 2-dimensional formulation. Since a cell can only be part of one alldifferent constraint, we had to increase the size of the model to be able to include all alldifferent constraints. However, we can manually construct constraints that function in the same manner as alldifferent constraints to avoid this increase in model

size. The COUNTIF function can be used to check that each integer 1 through n is used at least once in each row, column, and sub-grid. This way we can create a 2-dimensional formulation with only n^2 integer variables, because we do not have to add variables to include the manually constructed alldifferent constraints.

To test that each row contains all n integers, the formula COUNTIF($B4:$J4;B$3) is copied to cells M4:U12. Cell M4 will then count the number of times the integer 1 (in cell B$3) appears in row 1 (cells $B4:$J4), cell N4 will count the number of times the integer 2 (in C$3) appears in row 1(cells $B4:$J4), and so on.

To test that each column contains all n integers, the formula COUNTIF(B$4:B$12;$A4) is copied to cells V4:AD:12. Cell V4 will then count the number of times the integer 1 (in Cell $A4) appears in column 1 (cells B$4:B$12), cell V5 will count the number of times the integer 2 (in cell $A5) appears in column 1 (cells B$4:B$12), and so on.

To test that the first sub-grid contains all n integers, the formula COUNTIF($B$4:$D$6;B$3) is copied to cells AE4:AM4. Cell AE4 will then count the number of times the integer 1 (in Cell B$3) appears in sub-grid 1 (cells $B$4:$D$6), cell AF4 will count the number of times the integer 2 (in cell C$3) appears in sub-grid 1 (cells $B$4:$D$6), and so on. Note that we have to make a unique formula for each sub-grid. For example, the formula COUNTIF($E$4:$G$6;B$3) refers to the second sub-grid, and is copied to AE5:AM5. We therefore have to enter n different formulas for the n different sub-grids (rows 4 through 12 in the columns AE thru AM).

Just one constraint in Solver, M4:AM12 >= 1, is sufficient to implement all $3n^2$ constraints described above that function in the same manner as the alldifferent constraints. And, of course, all decision variables have to be constrained to integers between 1 and n. Conditional formatting has been applied in the spreadsheet to highlight violations of the manually constructed alldifferent constraints.

Unfortunately the COUNTIF function used in all of these cells is non-smooth. We therefore must select the solver engine carefully.

Most Solver engines rely on derivatives. For linear models the derivatives are computed only once, as the derivatives are constant. For non-linear models, when using a gradient-based solver, the derivatives are computed for each iteration. For non-smooth functions the derivatives are not always defined, and traditional solver engines are likely to fail. The Premium Solver Platform has a PSI Interpreter (Polymorphic Spreadsheet Interpreter) that when used will diagnose a function and calculate derivatives more precisely (in most cases) than the numerical estimates made when using Excel's recalculator.

If we try to apply a Linear Programming solver engine to this model, we will get an error message stating that this problem is not linear. If we apply a non-linear solver engine it will most likely fail to find a feasible solution, due to the non-smooth constraints. The appropriate solver engines for non-smooth models are the heuristic solver engines Standard Evolutionary Solver Engine or OptQuest Solver Engine.

Heuristic solvers do not rely on derivatives. In fact they do not have optimality tests either, they simply try to improve the value of the objective function based on several heuristic strategies. Instead, they use simple stopping rules like maximum time or maximum number of iterations. In addition, both the Standard Evolutionary Solver and OptQuest have additional tests for improvements in the objective function. If we let the heuristic solvers continue to run after these tests (we get an option to Stop or Continue), then Solver will in principle run forever.

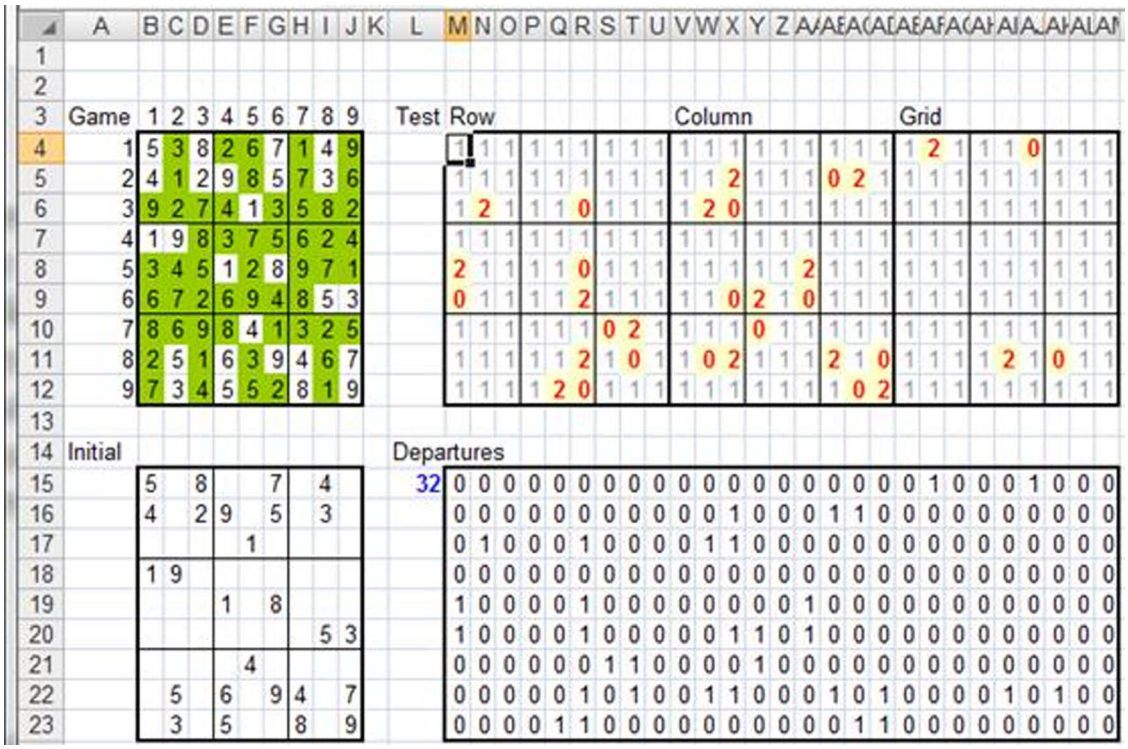
To this point we have not used an objective function in our models because the goal is simply to find a feasible solution. The lack of an objective is bad for heuristic solvers because typically they are not very good at handling constraints. Winston and Venkataramanan (2003) suggest that one way to reduce this difficulty is to include a penalty in the objective for violating a constraint. Fortunately we can easily construct an objective function, which will consist of the sum of the penalties for not satisfying each constraint. We can achieve this by computing the sum of the absolute deviations from the constraints, and then minimizing this sum. We accomplish this by placing the formula ABS(M4-1) into M15 and copying this formula to cells M15:AM23, and entering the formula SUM(M15:AM23) in L15, which we designate in Solver as the objective we want to minimize.

When Solver is running, the value of the objective function, cell L15, is displayed as the Set Cell in the status bar, in the bottom of the Excel window. When we see that the objective function has not improved for a very long time (or we see that the optimal value has been found), we can press the Esc button and stop Solver. We display a screen of the problem in mid execution in Figure 7. Notice in the status bar that the solution has improved from 32 violations in Figure 6, to 4 violations in Figure 7, but has not reached the 0 violations we require for a Sudoku solution.

Lesson 5: Ease of Data Entry May Make the Solution More Difficult

We display the Solver settings in Figure 8. Observe that we have omitted constraints for the initial values given in each specific puzzle. Since heuristic solvers are not good at handling constraints, adding more constraints is not helpful. In each Sudoku puzzle there

Figure 6 The 2-Dimensional Formulation with n^2 Integer Variables



are some initial numbers given for a few selected cells. These cells are therefore no longer among the decision variables. The simplification to include the total grid as decision variables and later add constraints for the initial fixed cells is a shortcut due to how most optimization software accepts input. Algebraically it looks similar, but it is not an identical formulation. Adding non-existing decision variables and correcting this shortcut by adding non-existing equality constraints does not help the heuristic solvers to find a feasible solution.

The actual number of decision variables in a 2-dimesional model of the Sudoku puzzle is therefore n^2 minus the number of initial fixed values given in a specific puzzle. The actual decision variables are highlighted using conditional formatting as can be seen in Figures 6 and 7.

Unfortunately, when the variables are not all contained in one rectangular range in Excel, Solver does not have an option to use all of the decision variables in one constraint line in order to define them to be integers, or to add upper or lower bounds on the decision variables. Therefore each group of decision variables has to be defined as: integers; with lower bounds of 1; and with upper bounds of 9. Finally in order to be consistent with our previous model we add one constraint for the alldifferent type of tests. Most of the constraints are displayed in Figure 9. The objective will then help Solver to find feasible solutions, negating the difficulty that the heuristic solvers typically have with constraints.

The OptQuest Solver Engine finds a feasible solution to a 2-dimensional 4×4 puzzle in a few seconds.

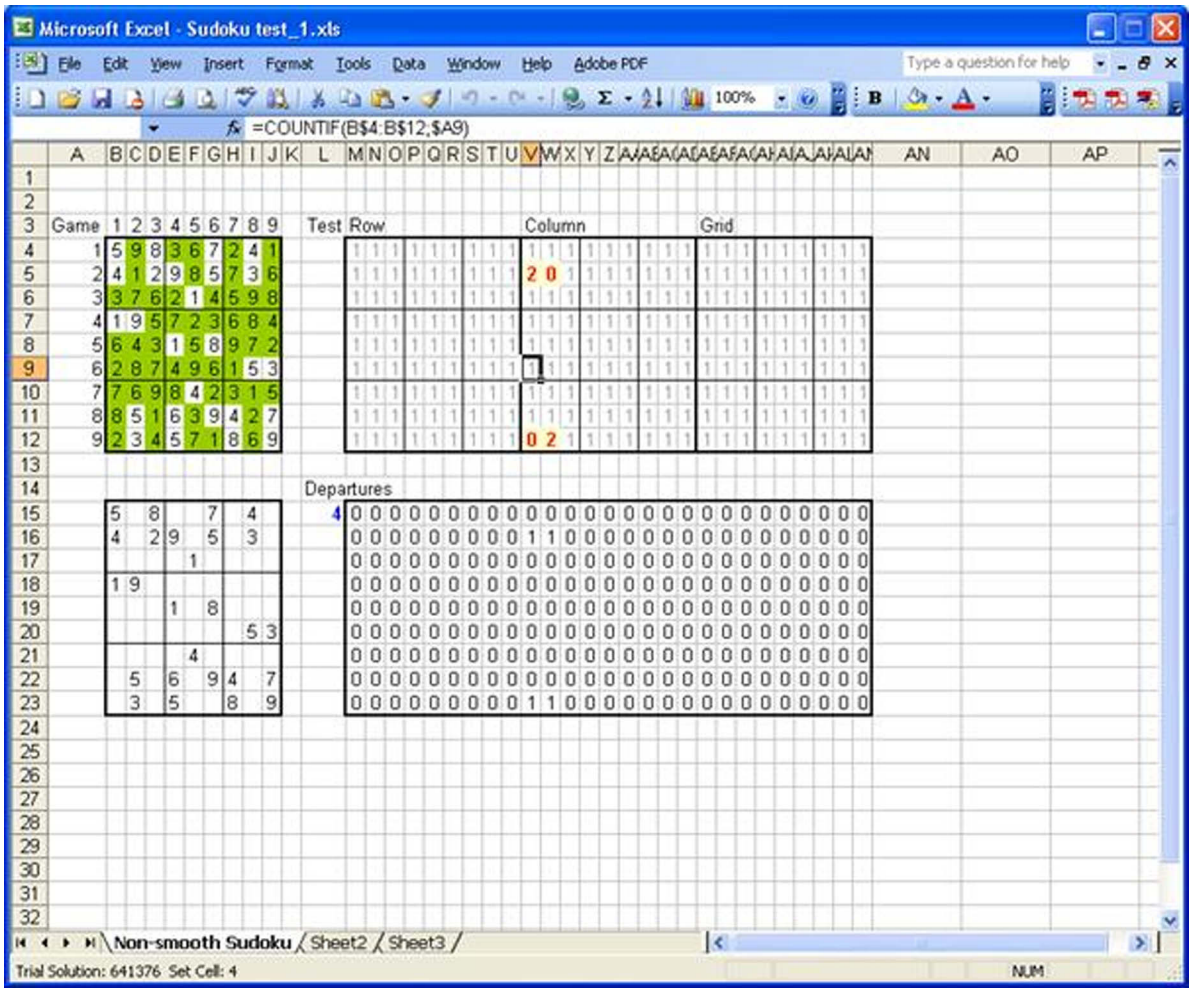
Table 4 Formulas for the 2-Dimensional Formulation with n^2 Integer Variables

Cell	Formula	Copied to:
M4	=COUNTIF($B4:$J4;B$3)	M4:U12
V4	=COUNTIF(B$4:B$12;$A4)	V4:AD12
AE4	=COUNTIF($B$4:$D$6;B$3)	AE4:AM4
AE5	=COUNTIF($E$4:$G$6;B$3)	AE5:AM5
AE6	=COUNTIF($H$4:$J$6;B$3)	AE6:AM6
AE7	=COUNTIF($B$7:$D$9;B$3)	AE7:AM7
AE8	=COUNTIF($E$7:$G$9;B$3)	AE8:AM8
AE9	=COUNTIF($H$7:$J$9;B$3)	AE9:AM9
AE10	=COUNTIF($B$10:$D$12;B$3)	AE10:AM10
AE11	=COUNTIF($E$10:$G$12;B$3)	AE11:AM11
AE12	=COUNTIF($H$10:$J$12;B$3)	AE12:AM12
M15	=ABS(M4-1)	M15:AM23
L15	=SUM(M15:AM23)	

Lesson 6: Problem Size Does Matter, Especially for Nonlinear Problems

While this model solved the 4 by 4 problem OptQuest was not able to solve the 9 by 9 problem within a reasonable amount of time. The Set Cell in The Status Bar never went to 0 (the value of the objective function), which would indicate that all constraints are satisfied.

Figure 7 The 2-Dimensional Formulation with n^2 Integer Variables During Solution Execution



Lesson 7: Speeding Up by Removing Constraints

We have added an objective to help the heuristic solvers with the constraints that are like the alldifferent constraints. Actually we do not need both the artificial objective and the constraints, and if we remove these constraints the objective should still direct us towards a feasible solution.

When we remove these constraints the solvers will of course run faster, as the problem size is reduced. In this situation, OptQuest finds a feasible solution more quickly when the constraints are dropped.

The Evolutionary Solver seem to have difficulties with this type of model and makes progress more slowly than OptQuest. After about 4 days (and more than 4 million iterations) the Evolutionary Solver still had 18 deviations, compared to Optquest that found a solution with only 4 deviations within 20 minutes. However, even though OptQuest got off to a much better start, it could not reduce the number of deviations even after 4 days and over 36 million iterations.

Lesson 3 Revisited: Select the Proper Solver Engine with Care

The Premium Solver Platform offers several standard solver engines, and even more solver engines can be added (at an extra cost). Each of these solver engines has its advantages and disadvantages, making them especially suitable for particular problems and unsuitable for others. Generally speaking we should use linear solvers if they accept the problem. If the problem is non-linear and smooth we should use one of the non-linear solvers. If the problem is non-smooth we should use a heuristic solver. The Premium Solver Platform can automatically diagnose the problem and list only suitable solvers for a particular model. This dialog box appears if we click the "Model" button in the main Solver window:

Observe that if the user checks the "Best" option in "Select Solver Engines Based on Model Type," this may involve none of the installed engines. The option "Good" or "Valid" should therefore be used if not all available solver engines have been installed.

Figure 8 Solver Settings for the Objective and Listing of the Decision Variables for the 2-Dimensional Model

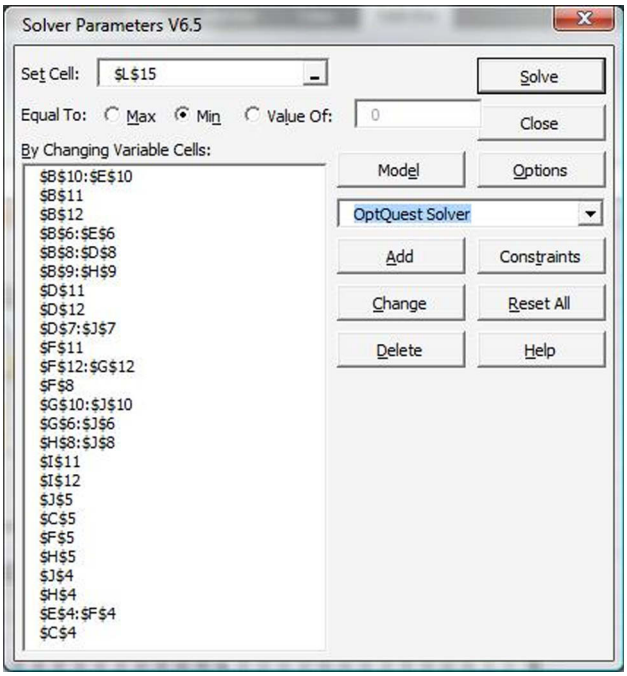


Figure 9 Solver Settings for the Constraints for the 2-Dimensional Model

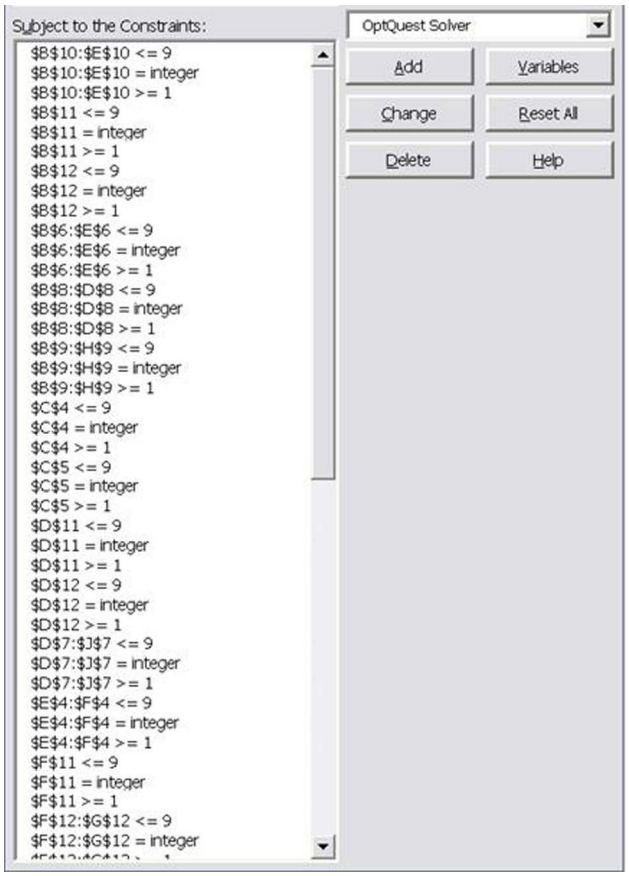
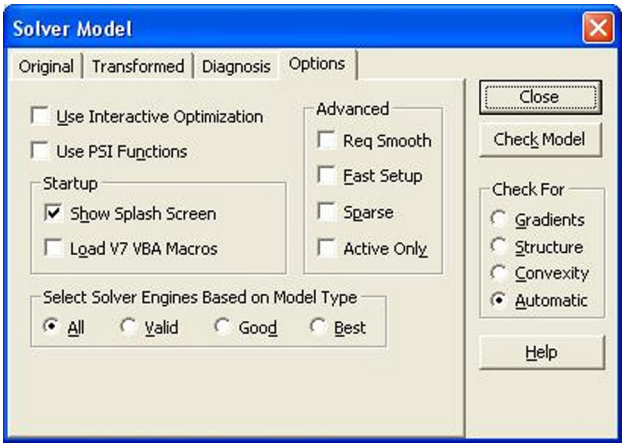


Figure 10 Solver Can Recommend a Suitable Solver Engine Based on a Model Diagnose



For heuristic solvers there are some limitations that must be considered. Adding an artificial objective for non-smooth problems may be helpful, but adding non-existing constraints to adjust for non-existing decision variables is generally not recommended.

4. Conclusions

In our previous paper we noted that in a different context, Koch (2005) wrote, "Choosing the right formulation is often more important than having the best solver algorithm." This is demonstrated here by the reduction in problem size achieved through the use of additional modeling techniques such as the alldifferent constraint, and comparing formulations using binary versus integer variables.

Students can learn several lessons from Sudoku. The new models have provided an excellent opportunity to introduce students to the alldifferent constraint and to assorted Solver engines. Hopefully the alternative models presented to solve the Sudoku puzzle will help students build their craft of optimization modeling.

Acknowledgments

We are extremely grateful to the editor and referees for their comments. In addition, we are especially grateful to Edwin Straver of Frontline Systems, the company that developed Solver, for his help regarding the alldifferent constraint and his comments on the branch and bound routines in the Solver engines.

References

Anonymous. 2006a. Premium Solver Platform User Guide, Frontline Systems, http://www.solver.com/supp_pspguide70.php (login required), (last accessed on Nov. 3, 2006).

Anonymous. 2006b. SolverAdd function. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/off2000/html/xlfctsolveradd.asp, (last accessed on Nov 3, 2006).

Anonymous. 2006c. Sudoku, http://www.sudoku.com/, (last accessed on May 11, 2006).

Chlond, M. J. 2005. Classroom exercises in IP modeling: Sudoku and the log pile. *INFORMS Trans. Ed.* **5**(2), http://ite.pubs.informs.org/Vol5No2/Chlond/, (last accessed on May 11, 2006).

Koch, T. 2005. Rapid mathematical programming or how to solve sudoku puzzles in a few seconds. Konrad-Zuse-Zentrum fur Informationstechnik Berlin, ZIB Report 05-51, http://www.zib.de/Publications/Reports/ZR-05-51.pdf, (last accessed on May 11, 2006).

Weiss, H. J., R. A. Rasmussen. 2007. Lessons from modeling sudoku in excel. *INFORMS Trans. Ed.* **7**(2), http://ite.pubs.informs.org/Vol7No2/Weiss/.

Wayne, W., M. A. Venkataramanan. 2003. *Introduction to Mathematical Programming: Applications and Algorithms*, 4th ed. Duxbury.