Paper 433-2012

# Yet Another Sudoku Solver: PROC FCMP

Matthew Kastin, I-Behavior, Inc., Louisville, Colorado
Arthur S. Tabachneck, Ph.D., myQNA, Inc., Thornhill, Ontario Canada

## ABSTRACT

There have been many different Sudoku solvers written using various SAS® tools (e.g., DATA step, PROC CLP, PROC OPTMODEL with MILP Solver, and PROC SQL). This paper introduces yet another tool, PROC FCMP, which can be used to solve Sudoku puzzles using recursive functions. A unique feature of this solver is the inclusion of the ability to determine whether a puzzle has a unique solution, which is a necessary step toward actually generating Sudoku puzzles directly with SAS. Additionally, the paper presents a comparative analysis of some of the other solutions that have been proposed.

## THE PROBLEM

Sudoku, according to Wikipedia, is "a log-based, combinatorial number-placement puzzle." The puzzle's name is the Japanese word for "single number." The puzzle was popularized in 1986 in Japan by a company called Nikoli and, by 2005, had gained wide-spread international popularity.

Potential puzzle solvers are provided a matrix that has nine rows and nine columns and at least seventeen (17) of the eight-one cells containing single digits that can range between one and nine, and all of the other cells will be blank. The solver's object is to fill the missing cells with digits that can range between one and nine, but every row and column must contain all nine digits. The matrix is also divided into nine (9) sub-matrices containing nine cells each, and each sub-matrix must contain all nine digits. Again, quoting Wikipedia, "completed puzzles are always a type of Latin square with an additional constraint on the contents of individual regions."

And, as evidenced by the many SAS-L posts that can be found on the topic, a number of SAS programmers have written various approaches to automatically solve Sudoku puzzles ever since they gained international popularity. Those efforts culminated in a 2007 SAS Global Forum paper and round table discussion that highlighted six different solutions that included using macros, arrays, SAS Windowing commands, various data step approaches, search algorithms, PROC SQL, using a cube, linear optimization with a combination of PROC ASSIGN and PROC LP, and constraint programming with PROC CLP,

The number of Sudoku-related SAS-L posts dropped significantly following the publication of that overview, but increased again this year when someone asked if it was possible to create a program that developed Sudoku puzzles (see: http://www.listserv.uga.edu/cgi-bin/wa?A2=ind1203C&L=sas-l&P=R9923 ). The present paper doesn't provide a solution to the requested task but, instead, proposes yet another method for solving such puzzles but one which, in our opinion, should be able to serve as the basis of an approach to use SAS to create valid Sudoku puzzles.

We will be presenting that approach in the next section of this paper, and will be attempting to address the original question in time to present a solution at next year's SAS Global Forum, we think that such efforts have a benefit that surpasses the task they are trying to accomplish. One of the problems the current authors have always faced in learning new methods and procs that are available in SAS, is being sufficiently familiar with both the problem being solved and the data involved. However, given a problem like that posed by a Sudoku puzzle, both the problem and the data are easily understood. And, by seeing how others have approached the problem with various methods, those methods and their application become obvious, thus making it easier to learn how to use and apply those methods. We hope this paper accomplishes that task with respect to showing how PROC FCMP can be used to solve many of the analytical tasks we all face.

## THE FCMP APPROACH

Any approach to solving a Sudoku puzzle has to consider the data in the form of the 9x9 two-dimensional array(s) that the data represent. Since base SAS is not inherently a matrix-based language, we elected to input puzzles as eighty-one (81) character strings, with each set of nine characters representing one of the array's nine rows. Thus, the last section of our code is where we define the problems we want to solve, and call the functions which were

created earlier in the code. Basically, this section serves as a driver to call the three routines which actually solve the puzzles, and then show the results in human readable form.

The puzzles, themselves, are defined as rows of a one dimensional array:

```
array args[11] $81 (
 '100007090030020008009600500005300900010080002600004000300000010040000007007000300'
 '000000070060010004003400200800003050002900700040080009020060007000100900700008060'
 '100500400009030000070008005001000030800600500090007008004020010200800600000001002'
 '080000001007004020600300700002009000100060008030400000001700600090008005000000040'
 '100400800040030009009006050050300000000001600000070002004010900700800004020004080'
 '005009700060000020100800006010700004007060030600003200000006040090050100800100002'
 '600000200090001005008030040000002001500600900007090000070003002000400500006070080'
 '100000060000100003005002900009001000700040080030500002500400006008060070070005000'
 '000010004030200000600008090007060005900005080000800400040900100700002040005030007'
 '400060070000000600030002001700008500010400000020950000000000705009100030003040080'
 '005300000800000020070010500400005300010070006003200080060500009004000030000009700'
);
```

While we didn't have time to expand the code to build Sudoku puzzles, as mentioned previously, the code was designed to later facilitate that effort. As such, it includes an array labeled test, that is used to specify whether a unique solution is required (i.e., a value of 1), or if only a solution is needed (i.e., a value of 0):

```
array test[11] (0 0 0 0 0 0 0 0 0 0 0);
```

The code then uses do loop logic to: (1) address each puzzle separately; (2) restructure each row of array *args* to be in the form of the 9x9 matrix it represents; (3) define an array to hold the puzzle; (4) use PROC FCMP's *read_array* function to actually read each restructured array; (5) output the original puzzle statement; (6) create and use a rather simple function, *writeMatrix,* to output arrays in a user-friendly form; (7) keep track of the time required to solve the problem (for testing purposes); (8) actually solve each puzzle; and, finally, (9) compute and output both the time elapsed and final solutions.

```
/*1*/ do i=1 to dim(args);
/*2*/   rc=parseProblem(args[i]);
/*3*/   array problem[9,9] /nosymbols;
/*4*/   rc=read_array('sudoku',problem);
/*5*/  put 'Problem= ' args[i];
/*6*/  x=writeMatrix(problem);
/*7*/  _time=time();
/*8*/  if solveProblem(test[i],problem)=0 then put 'No Unique Solution Found';
/*9*/  _diff=time()-_time;
       put 'Time Elapsed: ' _diff best. 'seconds';
       put _page_;
      end;
run;
```

**THE PARSEPROBLEM AND WRITEMATRIX FUNCTIONS.** These functions are rather self-explanatory.

```
function parseProblem(p $);
  length puzzle $82;
  puzzle=p || ' ';
  array problem[9,9] /nosymbols;
  k=1;
  do i=1 to 9;
   do j=1 to 9;
    problem[i,j]=input(substr(puzzle,k,1),1.);
    k+1;
   end;
  end;
  rc=write_array('sudoku',problem);
  return(rc);
 endsub;
```

```
function writeMatrix(solution[*,*]);
 put "   ----------------------";
 do i=1 to 9;
  put @2 '|' @;
  h=4;
  do j=1 to 9;
    x=ifc(solution[i,j]=0,' ',put(solution[i,j],1.));
    put @h x $2. @; h+2;
    if mod(j,3)=0 then do; put @h "| " @; h+2; end;
  end;
  put /;
  if mod(i,3)=0 then put @1 "   ----------------------";
 end;
 return(rc);
endsub;
```

**THE SOLVEPROBLEM, SOLVEFORWARD AND SOLVEREVERSE FUNCTIONS.** These are the main subroutines that form our solution and accomplish all of the required tasks in such a way that makes the final operational use as simple as possible. The principal action is to take two arguments: u (Boolean, indicating whether the solver should search for a unique solution (1) or just a solution (0); and, p (the puzzle string of 81 numbers with missing cells given as zeros). The first action performed is to convert the puzzle string to an array. The next action is to print the beginning puzzle. The goal from here is to fill in the missing numbers and for this there are two functions solveForward and solveReverse. These two functions are mostly identical with the difference being the direction and starting point from which each starts is solution. These two approaches will together be able to determine if puzzles have a unique solution. Used independently we can also optimize our strategy by approaching the puzzle for the side with the least initial guesses. The solveProblem subroutine implements the overall logic and, once solved, each solution is printed along with the time that was required to find the solution.

The solveForward and solveReverse functions are where the recursive piece of the algorithm is established. There are three main actions in this function. We need to control out position in the array as we step through the puzzle skipping boxes that contain given values and checking guesses then either stepping forward or backing up to move through until the puzzle is either solved or proved invalid. The recursion is accomplished rather simply by having these functions continuously call themselves until a final resolution is met. Function checkGuess performs the necessary actions in order to confirm or deny a value guess as being valid and enforces the three main constraints of the Sudoku game: a value can only appear once in a row, column and box.

The functions use a rather deterministic, back-tracking logic, similar to that which has been proposed for solving the eight queens puzzle (see, e.g., http://en.wikipedia.org/wiki/Eight_queens_puzzle ), where structured programming is used in a brute force approach and various heuristics are applied as shortcuts.

## COMPARING THE VARIOUS METHODS

While we originally intended to show efficiency comparisons between the various methods that have been proposed to solve Sudoku problems, we quickly discovered that each method has its own benefits and limitations. The method proposed in this paper appears to be able to solve any Sudoku puzzle, and can solve some puzzles faster than the other methods, but even the most limited approach will be faster for particular puzzles. As such, the authors decided that presenting the results of a comparison would be misleading and contrived.

## FUTURE CHANGES TO THE CODE

We did our best to only include carefully written and tested code, but the code may have to be updated to correct for errors that we or others might discover. As such, we created a page for the paper on sasCommunity.org. The page includes copies of this paper, the powerpoint that was used to present the paper, and the code described earlier. The page can be found at: http://www.sascommunity.org/wiki/Yet_Another_Sudoku_Solver:_PROC_FCMP.

## DISCLAIMER

The contents of this paper are the work of the authors and do not necessarily represent the opinions, practices or recommendations of the authors' organizations.

## CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the authors at:

Matthew Kastin
i-behavior, Inc.
2051 Dogwood Street, Suite 220
Louisville, CO 80027
E-mail: matthew.kastin@gmail.com

Arthur Tabachneck, Ph.D., President
myQNA, Inc.
80 Willowbrook Road
Thornhill, ON L3T 5K9 Canada
E-mail: atabachneck@gmail.com

## APPENDIX I

**/*CODE FOR SOLVING SUDOKU PROBLEMS USING PROC FCMP*/**

```
proc fcmp;
 function parseProblem(p $);
  length puzzle $82;
  puzzle=p || ' ';
  array problem[9,9] /nosymbols;
  k=1;
  do i=1 to 9;
   do j=1 to 9;
    problem[i,j]=input(substr(puzzle,k,1),1.);
    k+1;
   end;
  end;
  rc=write_array('sudoku',problem);
  return(rc);
 endsub;

 function solveProblem(u,p[*,*]);
   if u then do;
    if solveForward(1,1,p) then do;
     array f[9,9] /nosymbols;
     rc+read_array('sudoku_f',f);
    end;
    if solveReverse(9,9,p) then do;
     array r[9,9] /nosymbols;
     rc+read_array('sudoku_r',r);
    end;
       m=0;
       if rc=0 then
     do i=1 to 9;
      do j=1 to 9;
          z=(f[i,j]=r[i,j]);
       m+z;
      end;
     end;
   end;
   else do;
    r1=0;
       r9=0;
       do i=1 to 9;
     r1+ifn(p[i,1]=0,1,0);
     r9+ifn(p[i,9]=0,1,0);
       end;
       if r1<r9 then do;
     if solveForward(1,1,p) then do;
      array f[9,9] /nosymbols;
         m=81+read_array('sudoku_f',f);
     end;
    end;
    else do;
     if solveReverse(9,9,p) then do;
      array f[9,9] /nosymbols;
      m=81+read_array('sudoku_r',f);
     end;
    end;
   end;
   if m=81 then z=writeMatrix(f);
  return(ifn(m=81,1,0));
 endsub;

 function solveForward(_i,_j,c[*,*]);
  array cells[9,9] /nosymbols;
```

```
 do a=1 to 9;
  do b=1 to 9;
   cells[a,b]=c[a,b];
  end;
 end;
 i=_i; j=_j;

 if i>9 then do;
  i=1; j+1;
  if(j>9) then do;
   rc=write_array('sudoku_f',cells);
   return(1);
  end;
 end;

 if cells[i,j] ne 0 then
  return(solveForward(i+1,j,cells));

 do val=1 to 9;
  if legal(i,j,val,cells) then do;
   cells[i,j]=val;
   if (solveForward(i+1,j,cells)) then return(1);
  end;
 end;

 cells[i,j]=0;
 return(0);
endsub;

function solveReverse(_i,_j,c[*,*]);
 array cells[9,9] /nosymbols;
 do a=1 to 9;
  do b=1 to 9;
   cells[a,b]=c[a,b];
  end;
 end;
 i=_i; j=_j;

 if i<1 then do;
  i=9; j=j-1;
  if(j<1) then do;
   rc=write_array('sudoku_r',cells);
   return(1);
  end;
 end;

 if cells[i,j] ne 0 then
  return(solveReverse(i-1,j,cells));

 do val=1 to 9;
  if legal(i,j,val,cells) then do;
   cells[i,j]=val;
   if (solveReverse(i-1,j,cells)) then return(1);
  end;
 end;

 cells[i,j]=0;
 return(0);
endsub;

function legal(i,j,val,cells[*,*]);
do k=1 to 9; *scan row;
 if val=cells[k,j] then
  return(0);
end;
do k=1 to 9; *scan col;
 if val=cells[i,k] then
```

```
   return(0);
 end;
 roffset=i-mod(i-1,3);
 coffset=j-mod(j-1,3);
 do k=0 to 2; *scan box;
  do m=0 to 2;
   if val=cells[roffset+k,coffset+m] then
        return(0);
  end;
 end;
 return(1);
 endsub;

 function writeMatrix(solution[*,*]);
 put "   ------------------------";
 do i=1 to 9;
  put @2 '|' @;
  h=4;
  do j=1 to 9;
       x=ifc(solution[i,j]=0,' ',put(solution[i,j],1.));
       put @h x $2. @; h+2;
       if mod(j,3)=0 then do; put @h "| " @; h+2; end;
  end;
  put /;
  if mod(i,3)=0 then put @1 "   ------------------------";
 end;
 return(rc);
 endsub;

array args[11] $81 (
  '100007090030020008009600500005300900010080002600004000300000010040000007007000300'
  '000000070060010004003400200800003050002900700040080009020060007000100900700008060'
  '100500400009030000070008005001000030800600500090007008004020010200800600000001002'
  '080000001007004020600300700002009000100060008030400000001700600090008005000000040'
  '100400800040030009009006050050300000000001600000070002004010900700800004020004080'
  '005009700060000020100800006010700004007060030600003200000006040090050100800100002'
  '600000200090001005008030040000002001500600900007090000070003002000400500006070080'
  '100000060000100003005002900009001000700040080030500002500400006008060070070005000'
  '000010004030200000600008090007060005900005080000800400040900100700002040005030007'
  '400060070000000600030002001700008500010400000020950000000000705009100030003040080'
  '005300000800000020070010500400005300010070006003200080060500009004000030000009700'
);
 array test[11] (0 0 0 0 0 0 0 0 0 0 0 0);
 do i=1 to dim(args);
  rc=parseProblem(args[i]);
  array problem[9,9] /nosymbols;
  rc=read_array('sudoku',problem);
  put 'Problem= ' args[i];
  x=writeMatrix(problem);
  _time=time();
  if solveProblem(test[i],problem)=0 then put 'No Unique Solution Found';
  _diff=time()-_time;
  put 'Time Elapsed: ' _diff best. 'seconds';
  put _page_;
 end;
run;
```