Paper 206-2010

Recursive SAS Macro: A Fun Application

John Zheng, Baltimore, MD

ABSTRACT

This paper describes a simple recursive SAS macro solution to 4×4(16 pieces) Eternity II, an edge-matching puzzle constrained by the requirement to match adjacent edges. By this fun macro solution, the concept of recursion and useful applications of recursive algorithm using SAS Macro is presented.

INTRODUCTION

The Eternity II puzzle is an edge-matching puzzle, which involves placing 256 square puzzle pieces into a 16×16 grid so that adjacent edges match in both color and shapes (collectively called "colors" here). The full 256-piece Eternity II puzzle has been designed to be difficult to solve by brute-force computer search. Simpler versions with fewer pieces are also provided. For example, a 4×4 clue puzzle game can be found at http://uk.eternityii.com/try-eternity2-online/ and its interface is illustrated in figure 1.

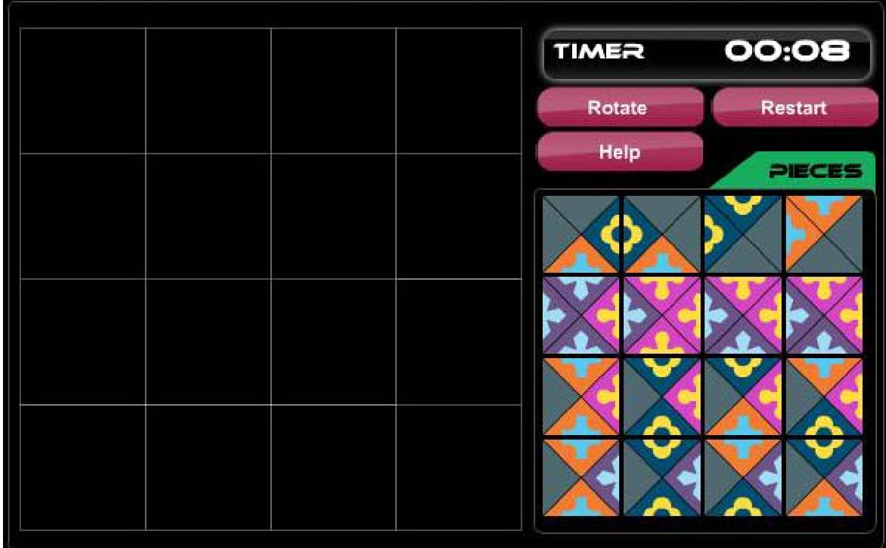


Figure 1: A 4×4 Eternity Puzzle Game

Each square puzzle piece consists of four triangles occupying four edges (left, top, right, and bottom). Each triangle is marked with a different color/shape. In the 4×4 puzzle, there are four colors, plus the grey color which is the only triangle can face the borders of the board. When a puzzle is completed, the color of the triangle on each edge (except the border-facing edge) must match precisely with the color of the triangle at the neighboring edge of an adjacent piece. One completed puzzle is shown in figure 2.

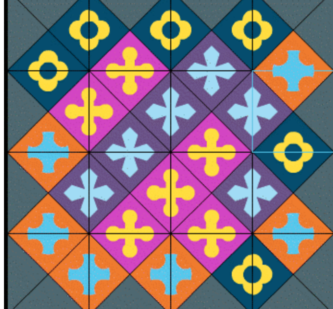


Figure 2: A solved Eternity II puzzle

Complex iterative algorithms can be used to solve problems such as the one presented here. However, creative use of recursive SAS macro, a special type of nested SAS macro, is proved to be more efficient. The key to recursive algorithm is to find the recursive function that can model both the original problem and a divided smaller problem. By self-calling, the recursive function successively divides the original problem into smaller problems till the smallest unit (the terminal level). The exit point of the recursive algorithm is triggered whenever an admissible match at the terminal level is found. In order to find an admissible match at the terminal level, a depth-first search (DFS) is used.

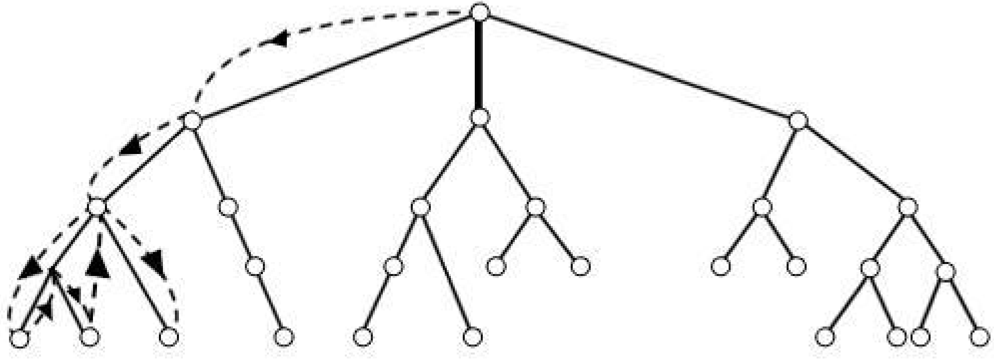


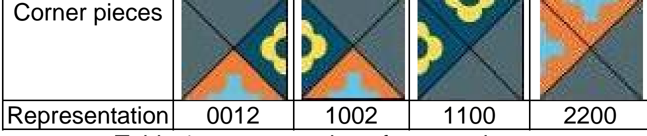
Figure 3: tree structure of depth-first searching

As shown in the above figure, a depth-first search (DFS) is essentially to start the search at the root of a tree structure and explores as far as possible before backtracking. Obviously the DFS can find all possible solutions to a problem after the entire tree is traversed. When only one solution is needed, we can also set the first solution as the exit point.

PROGRM DESIGN

Before we can use the power of recursive SAS macros to solve this problem, we first explain how pieces are represented internally. We use numbers 0 to 4 to represent five different colors (0-grey, 1-blue, 2-orange, 3-purple, and 4-pink). A piece with a certain configuration can be represented by color numbers of its four edges, starting from the left edge clock-wise. For example, a piece with a grey left edge, an orange top edge, a purple right edge, and a blue bottom edge is represented as 0231.

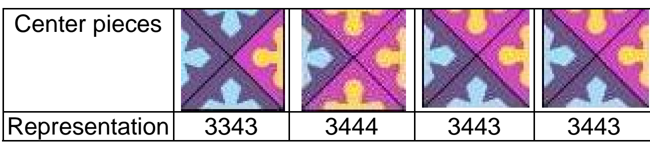
Because only grey color is permitted for edges facing borders of the board, we may categorize puzzle pieces by their number of grey edges. We call pieces with 2 grey edges "corner pieces," pieces with one grey edge "border pieces," and pieces with no grey edge "center pieces." We also give each piece a unique identification number. In figure 1, the four corner pieces are



Corner pieces				
Representation	0012	1002	1100	2200

Table 1: representation of corner pieces

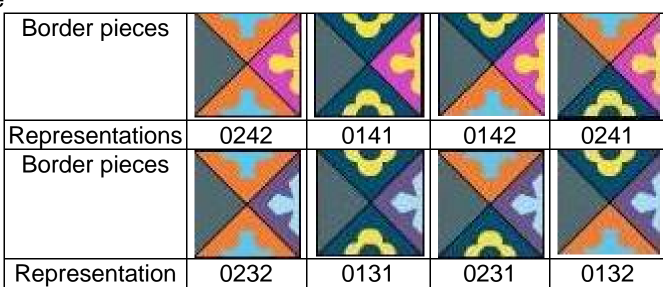
The four center pieces are



Center pieces				
Representation	3343	3444	3443	3443

Table 2: representation of center pieces

The eight border pieces are



Border pieces				
Representations	0242	0141	0142	0241
Border pieces				
Representation	0232	0131	0231	0132

Table 3: representation of border pieces

With the above representation of piece configurations, we must also represent two key elements of the game: the layout of the inventory board and the game board. We will use character strings to represent both. First, we will use numbers 1 to 16 to index 16 grids on the each board. The index for each grid is shown on an empty board in the following figure.

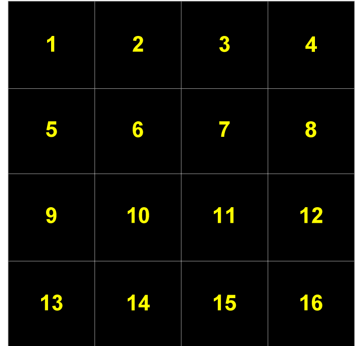


Figure 4. Grid indices

We simply use a string of piece numbers to represent the pieces on the inventory board. We use "…." to indicate a grid on the inventory board is empty. For example, the inventory board in figure 1 can be represented by the following character string

0012|1002|1100|3343|3444|3443|3443|2200|0242|0141|0142|0241|0232|0131|0231|0132

After taking the first piece out, its character string becomes

....|1002|1100|2200|0242|0141|0142|0241|0232|0131|0231|0132|3343|3444|3443|3443

The game board is represented differently from the inventory board. In the game board, we use missing value (i.e., ".") to indicate any color is allowed for the edge whereas a number 0 (1, 2, 3, or 4) indicates a grey (blue, orange, purple, and pink) edge is required for that piece. In other words, the numbers at a particular grid on the game board represents what is expected for that grid (unless it is already filled). Such a representation can let us quickly determine whether a piece is admissible for a grid on the game board. By this representation, an empty game board is represented by

00..|.0..|.0..|.00.|0...|....|....|..0.|0...|....|....|..0.|0..0|...0|...0|..00

The above character string reflects the requirement that border-facing edges must be grey edges.

Another important element of the recursive program is the representation of a move, i.e., an act of moving a certain piece from the inventory board to a particular grid on the game board. We come up a 3-number string, x-y-z, to represent such a move, where the first number (x) represents the index of target grid on the game board, the second number (y) represents the index of the source grid on the inventory board, and the last number (z) represents the rotation of the piece. We use the following numbers to represent how we rotate a piece (all rotations are clock-wise).

0	no rotation
1	90-degree rotation
2	180-degree rotation
3	270-degree rotation

Table 4: rotation indices

The following table illustrates how rotation changes the string associated with a piece

Rotation	0	1	2	3
Rotation Degree	0	90	180	270
Piece				
Representation	0132	2013	3201	1320

Table 5: an example of rotation & representation

Suppose we move the first piece in the inventory board and place it at the first grid of the game board. The move, the new inventory board status, and the new game board status are represented as respectively in table 6.

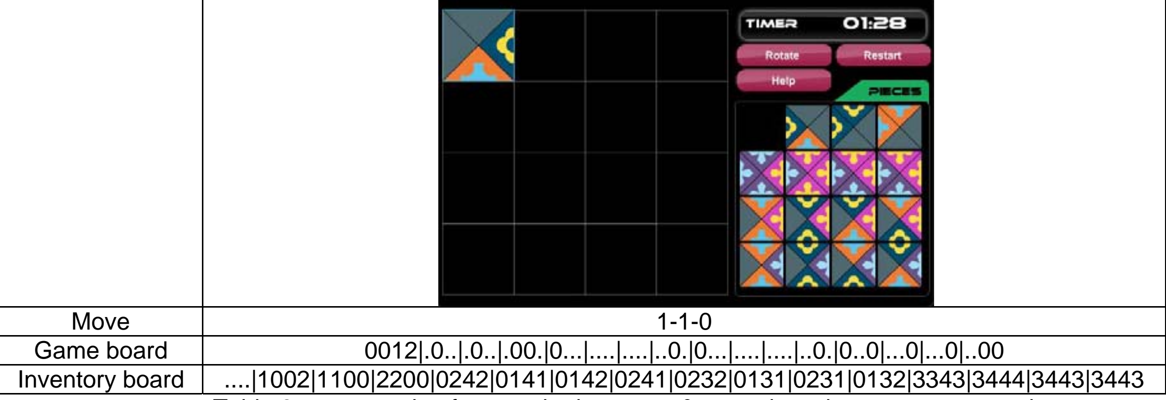
	
Move	1-1-0
Game board	0012|.0..|.0..|.00.|0...|....|....|..0.|0...|....|....|..0.|0..0|...0|...0|..00
Inventory board	....|1002|1100|2200|0242|0141|0142|0241|0232|0131|0231|0132|3343|3444|3443|3443

Table 6: an example of move, the inventory & game board status representation

PROGRAM IMPLEMENTATION

After the design, we now talk about the programming structure. First as mentioned that key to the recursive algorithm building is to find its recursive function. Before we get to programming implementation, below are the flowchart and pseudo code of recursive function used in our solution.

Figure 5: flowchart of the recursive function

```
Function PlaceOnePiece(GameBoard,  InventoryBoard){
       If the inventory board is not empty
              Get the next piece on the inventory board
              For each empty grid on the game board do
                     For each orientation of the piece do
                            If the piece fits in the grid
                                   Put the piece in the game board
                                   Call PlaceOnePiece(GameBoard,  InventoryBoard)
                            End if
                            Break if Found
                     Next orientation
                     Break if Found
              Next empty grid
       Else
              Print moves and game board
              set global variable Found to True
       End If
End Function
```

Figure 6: pseudo code of the recursive function

RECURISVE MACRO IMPLEMENTATION

The recursive macro is defined as:

```
%macro eternity(gboard=, iboard=, moves=);
```

in which the parameter **gboard** provides the current game board layout, whereas the parameter **iboard** provides the current inventory board layout. The last parameter **moves** stores the previous moves. Note that every time a recursive macro is invoked, the macro is provided a set of parameter values along with fresh local macro variables. The local macro variables within the nested macro are not visible to its calling macro because local variables can only be accessed within the macro where they are created. In this sense, recursive macros work like any other SAS macros except that the recursive macros apply the same logic at each turn of the problem.

An important macro variable is *solution,* which is the flag to indicate whether a solution is found.

```
%if %length(%sysfunc(compress(&board, .|))) <64 %then %let solution = 0;
```

The **solution** is set to **global** so that an upper level macro can learn that a solution has been found by some terminal level macro and stop trying the next move.

Game Board and Inventory Board

The game board and the inventory board are initialized before the recursive macro is called.

```
%global oiboarads ogboards solution;
%let oiboards =
0012|1002|1100|2200|0242|0141|0142|0241|0232|0131|0231|0132|3343|3444|3443|3443;
%let ogboards =
00..|.0..|.0..|.00.|0...|....|....|..0.|0...|....|....|..0.|0..0|...0|...0|..0;
```

In the macro, the local macro variables array ***gboard1-gboard16*** & ***iboard1-iboard16*** are declared to represent the layout of inventory board and the layout of the game board. Those macro variables array are **local** to the recursive macro, and are used to save the status of game and inventory boards during each recursive calling.

```
%local gboard1 gboard2 gboard3 gboard4 gboard5 gboard6 gboard7 gboard8 gboard9
gboard10 gboard11 gboard12 gboard13 gboard14 gboard15 gboard16;
%local iboard1 iboard2 iboard3 iboard4 iboard5 iboard6 iboard7 iboard8 iboard9
iboard10 iboard11 iboard12 iboard13 iboard14 iboard15 iboard16;
/*generate macro variables array*/
%do i = 1 %to 16;
       %let iboard&i=%scan(&iboard, &i, |);
       %let gboard&i=%scan(&gboard, &i, |);
%end;
```

Once the implementation of the board layout is established, the recursive macro will search for a legitimate move (i.e., to fit one piece from the inventory board into the game board). The search is exhaustive in that it will try every remaining piece on the inventory board, every vacant grid on the game board, and every rotation of a piece before a legitimate move is found. The outer loop is used to search over all the remaining pieces in the inventory board

```
%do iboardidx = 1 %to 16;
       %if not %length(%sysfunc(compress(&&iboard&iboardidx, .)))=4 %then %do;
              …
       %end;
%end;
```

The intermediate loop is for searching over all the remaining empty grids on the game board for a given inventory piece.

```
%do gboardidx = 1 %to 16;
  %if %length(%sysfunc(compress(&&gboard&gboardidx, .)))<4 %then %do;
       …
  %end;
%end;
```

And the inner loop is to for trying every possible rotation for a given inventory piece and a given empty grid on the game board

```
%do rotateidx = 1 %to 4;
    …
%end;
```

The 3-level loops exhaust all the possible moves at the current level given the moves made in the previous levels.

Matching

First step in matching is to create a virtual piece **nbgboard** for each empty grid on the game board by collecting information from its 4 neighboring grids. The matching is implemented by comparing 4 digits in the virtual piece with those of available pieces:

```
%do j = 1 %to 4;
    %if %substr(&piece,&j,1)=%substr(&nbgboard,&j,1)
              or %substr(&nbgboard,&j,1)=.
              and %length(%sysfunc(compress(&nbgboard, 0)))
                  =%length(%sysfunc(compress(&piece, 0))) %then
              %let vmth=&vmth.0;
        %else %let vmth=&vmth.1;
```

```
%end;
```

If no match is found, the loop will reach to its end and the macro will return to its calling macro, in which the next possible move at the higher level will be tried. In a special case, if we reach the terminal level (with only one piece left on the inventory board) and all 4 rotations turn out to be no match, we then reach a dead end. The current iteration then finishes and returns to the calling macro in which the next potential move at a higher level is checked. And backtrack will be explained with an example in the next section.

Recursive Function Implementation

If a match is found, the statements below will accept the move and save it into the macro variables ***localmoves***, which will be shortly passed on to the recursive call at the next level via the parameter *moves*

```
%if &rotateidx=4 %then %let ridx=0;
%else %let ridx=&rotateidx;
%let allmoves = &moves &gboardidx-&iboardidx-&ridx;
```

Besides saving the move into the parameter variable, statements below show how the new inventory board and the game board layout are first saved into local macro variable array **newiboard1- newiboard16** and **newgboard1-newgboard16**, and then passed on into parameter variable **iboards** and **gboards**.

```
%do i = 1 %to 16;
       %if &i = &iboardidx %then  %let newiboard&i= ....;
       %else %let newiboard&i= &&iboard&i;
       %if &i = &gboardidx %then  %let newgboard&i= &piece;
       %else %let newgboard&i= &&gboard&i;
%end;

%let iboards=&newiboard1;
%let gboards=&newgboard1;
%do i = 2 %to 16;
       %let iboards= &iboards|&&newiboard&i;
       %let gboards= &gboards|&&newgboard&i;
%end;
```

In this recursive structure, each macro will loop through all the possible moves at its level, given the moves at previous levels, until one move yields a solution. Then here comes the core conditional statement of the recursive implementation.

```
%if %length(%sysfunc(compress(&gboards, .|))) = 64 %then %let solution = 1;
%else %eternity(gboard=&gboards, iboard=&iboards,moves=&localmoves);
```

The ***%if*** statement checks whether the updated game board layout is a solution. If it is a valid solution, the global variable *solution* is set to 1, which will notify the recursive macros at all levels to stop searching and return to the higher levels immediately. If the updated game board layout is not yet a solution, the recursive macro calls itself with the updated inventory board and game board layout parameters to fit another piece into the game board.

IMPLEMENTATION OF BACKTRACK

As described in the previous section, our recursive macro is set up in such a way that it will call itself to fit in another piece after successfully fitting one piece. The critical question is what happens if it fails to fit in a piece. First, by the logic of our recursive macro, it will try to rotate the current piece and see if it fits the current empty grid (see the inner loop). If none of the rotation works, it will try to fit the piece in the next empty grid on the game board (including any rotation of the piece).

We will use the tree structure in figure 7 to show how the trackback is done in the depth-first searching recursive algorithm. And it is easy to see by the nested recursions, the searching path is from node 13.j through node 14.k and then to node 15.m, which is a dead end. The recursion unit 15.m will finish with no solution and the search program will “backtrack” to upper recursion unit 14.k, and then keep on looping into recursion unit 15.m+1. Once recursion unit 15.m+1 turns out to be a dead end, the recursion unit 14.k is found to be a dead loop. The searching program then will backtrack to recursion unit 13.j. The process keep going until a solution 16.n is found. The actual implementation is shown in figure 8.

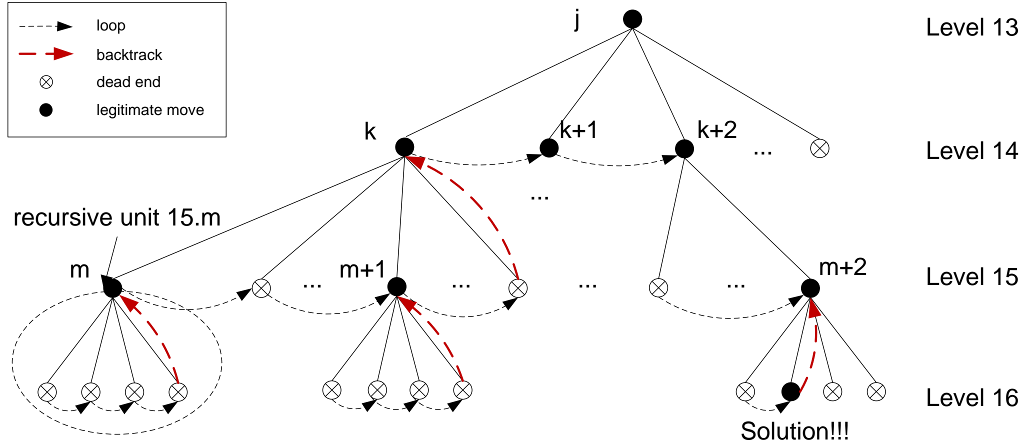


Figure 7: an example tree structure of how backtrack works

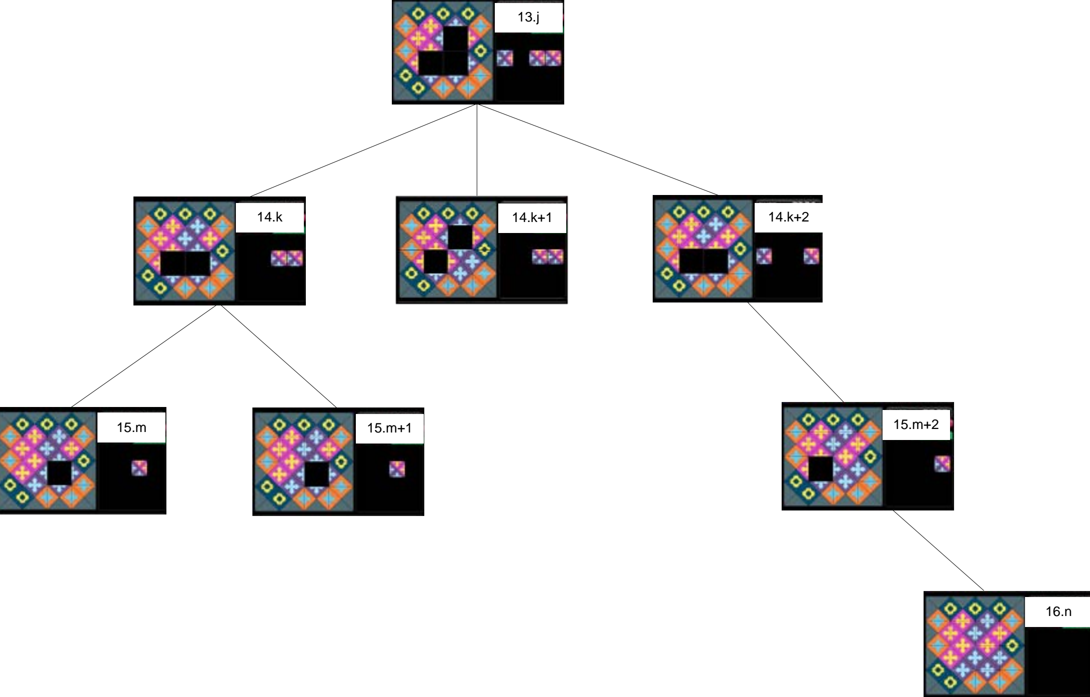


Figure 8: Moves made by the depth-first search and backtrack process

One thing to keep in mind is that not all initial layout of the inventory board are equal. Certain starting layout of the inventory board may take much longer time to find a solution than others. We can accelerate our search by employing search heuristics. In solving the Eternity I puzzle, mathematicians Alex Selby and Oliver Riordan employed a strategy of putting "bad" pieces first and "good" pieces last. We use a similar search heuristic here. It is easy to see that center pieces apply to more grids than border pieces, and so do border pieces than corner pieces. So our strategy is to place corner pieces first, followed by border pieces, and finally center pieces. To do so, we simply sort our inventory board by the order of corner pieces, border pieces, and center pieces. One example of the sorted inventory board layout is
0012|1002|1100|2200|0242|0141|0142|0241|0232|0131|0231|0132|3343|3444|3443|3443

With this inventory board layout, the search algorithm will place the 4 corner pieces first, then the 8 border pieces, and finally the 4 center pieces. By doing so, we reduce the searching possibilities to $4! \times 8! \times 4! \times 4^4 = 5.945 \times 10^9$. With our search heuristics, we can set the first found solution as the exit point. For example, with the rearranged inventory board show in the above, we find a solution after 41 steps in 32 seconds on a Pentium Core 2 Duo PC with windows XP operating system and 1 gigabyte memory. The solution involves the following moves
1-1-0, 4-2-0, 13-3-1, 16-4-0, 5-5-0, 2-6-1, 8-7-2, 9-8-0, 15-9-3, 3-10-1, 12-11-2, 14-12-3, 6-14-2, 7-15-1, 11-13-3, 10-16-3
and the final game board layout is
0012|1014|1013|1002|0242|4434|3344|4201|0241|4433|3433|3102|0110|1320|2320|2200.

CONCLUSION

This paper uses this puzzle solving example to show how recursive algorithm is implemented using SAS macro. Whenever dealing with problems that can be successively divided into smaller problems that resemble the original problem, we should take advantage of recursion technique. We can all appreciate how powerful and efficient recursion programming can be, as shown in this presentation.

REFERENCES

Tabladillo, M. (2005), “Macro Architecture in Pictures”, *SUGI Proceedings*, 2005.
Li H.(2005) “The Pegboard Game: A Recursive SAS® Macro Solution” *NESUG Proceedings*, 2005

ACKNOWLEDGMENTS

I would like to thank Xueqi Chen & Chao Liu for working the C++ version with me, also like to thank Houliang Li for his constructive suggestions.

CONTACT INFORMATION

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

John Zheng
8852 Town & Country Blvd,
Ellicott City, MD 21043
(571)-329-1496
hqzheng@gmail.com

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

APPENDIX

```
%global oiboards ogboards solution;
%let oiboards =
0012|1002|1100|2200|0242|0141|0142|0241|0232|0131|0231|0132|3343|3444|3443|3443;
%let ogboards =
00..|.0..|.0..|.00.|0...|....|....|..0.|0...|....|....|..0.|0..0|...0|...0|..00;

%macro eternity(gboard=,  iboard=, moves=);
%if %length(%sysfunc(compress(&gboard, .|))) <64 %then %let solution = 0;
%local iboards gboards piece iboardidx gboardidx rotateidx;
%local gboard1 gboard2 gboard3 gboard4 gboard5 gboard6 gboard7 gboard8 gboard9
gboard10 gboard11 gboard12 gboard13 gboard14 gboard15 gboard16;
%local iboard1 iboard2 iboard3 iboard4 iboard5 iboard6 iboard7 iboard8 iboard9
iboard10 iboard11 iboard12 iboard13 iboard14 iboard15 iboard16;

/*generate macro variables array*/
%do i = 1 %to 16;
  %let iboard&i=%scan(&iboard, &i, |);
  %let gboard&i=%scan(&gboard, &i, |);
%end;

/*searching across all available inventory board grids*/
%do iboardidx = 1 %to 16;
  %if not &solution and %length(%sysfunc(compress(&&iboard&iboardidx, .)))=4 %then
%do;
  /*current inventory board piece*/
  %let piece =&&iboard&iboardidx;

  /*searching across all empty game board grid*/
  %do gboardidx = 1 %to 16;
  %if not &solution and %length(%sysfunc(compress(&&gboard&gboardidx, .)))<4
        and %length(%sysfunc(compress(&&iboard&iboardidx, .)))=4 %then %do;

     /*create a virtual piece nbgboard by collecting the info from adjoining grids*/
```

```
%let nbgboard=;
%do j = 1 %to 4;
   %if %substr(&&gboard&gboardidx,&j,1)=0 %then %let nbgboard=&nbgboard.0;
  %else %do;
    %if &j=1 %then %do;
     %let nbgboardidx=%eval(&gboardidx-1);
     %let nbgboard=&nbgboard.%substr(&&gboard&nbgboardidx,3,1);
    %end;
    %if &j=2 %then %do;
     %let nbgboardidx=%eval(&gboardidx-4);
     %let nbgboard=&nbgboard.%substr(&&gboard&nbgboardidx,4,1);
    %end;
    %if &j=3 %then %do;
     %let nbgboardidx=%eval(&gboardidx+1);
     %let nbgboard=&nbgboard.%substr(&&gboard&nbgboardidx,1,1);
    %end;
    %if &j=4 %then %do;
     %let nbgboardidx=%eval(&gboardidx+4);
     %let nbgboard=&nbgboard.%substr(&&gboard&nbgboardidx,2,1);
     %put &nbgboardidx &gboardidx &&gboard&gboardidx;
    %end;
   %end;
%end;

/*rotate current piece 4 times to match with the virtual piece*/
%do rotateidx = 1 %to 4;
%if not &solution  and %length(%sysfunc(compress(&&gboard&gboardidx, .)))<4 %then
  %do;
   %let piece=%substr(&piece,4,1)%substr(&piece, 1, 3);

  /*matching the grid with current inventory board piece*/
  %let vmth=;
  %do j = 1 %to 4;
      %if %substr(&piece,&j,1)=%substr(&nbgboard,&j,1)
      or %substr(&nbgboard,&j,1)=.
      and %length(%sysfunc(compress(&nbgboard, 0)))
          =%length(%sysfunc(compress(&piece, 0))) %then %let vmth=&vmth.0;
    %else %let vmth=&vmth.1;
  %end;

  /*match found*/
   %if %length(%sysfunc(compress(&vmth, 0))) =0 %then %do;
      %if &rotateidx=4 %then %let ridx=0;
     %else %let ridx=&rotateidx;
     %let localmoves = &moves &gboardidx-&iboardidx-&ridx;

     %do i = 1 %to 16;
       %if &i = &iboardidx %then  %let newiboard&i= ....;
       %else %let newiboard&i= &&iboard&i;
       %if &i = &gboardidx %then  %let newgboard&i= &piece;
       %else %let newgboard&i= &&gboard&i;
     %end;

     %let iboards=&newiboard1;
     %let gboards=&newgboard1;
     %do i = 2 %to 16;
       %let iboards= &iboards|&&newiboard&i;
       %let gboards= &gboards|&&newgboard&i;
     %end;
      %put current moves are: &localmoves;
     %put current inventory board layout are: &iboards;
     %put current game board layout are: &gboards;
     %put ;
```

```
          /*check whether the puzzle is solved*/
          %if %length(%sysfunc(compress(&gboards, .|))) = 64 %then %do;
            %let solution = 1;
            %put Original inventory boards are: &oiboards;
            %put Final game boards is: &gboards;
            %put Successful moves are: &localmoves;
            %end;
          %else %eternity(gboard=&gboards, iboard=&iboards,moves=&localmoves);
       %end;
    %end;
    %end;
  %end;
  %end;
%end;
%end;
%mend;


%eternity(gboard=&ogboards, iboard=&oiboards);
```