Among many other worthwhile contents, the previous issue of Datafile featured an article that, being as truly fond of combinatorial puzzles as I am, immediately caught my fancy, namely “(--) sudoku (--)” by Peter Gatenby (V24N1 p48-50). Though it seems that sudoku puzzles aren’t exactly new, I simply hadn’t seen them before, and was intrigued by its very simple rules yet substantial challenge in degrees varying from relatively affordable to plainly exasperating.

Also, being an HP-calc fan I instantly saw that it would make a fair programming challenge. How difficult? Well, Mr. Gatenby’s interesting article describes his own attempt, and the perspective seems rather gloomy. He says, for example (the highlighting is mine):

“I have written a long and complicated hp49g+ program to solve sudoku puzzles. The printout on HP-IR printer strip is about one metre long and it seems pointless to take up pages of Datafile with a huge listing ...”

Wow. And for the HP49g+ no less, i.e.: the fastest, most powerful HP calculator ever. Mr. Gatenby doesn’t include any program code for the aforementioned reasons, but delves instead into a description of his program. He tells us that it “must run six ‘first-order’ solution procedures” but can encounter a so-called “thwarting” condition in which case, in Mr. Gatenby’s own words:

“The solver needs to use old-fashioned pencil and paper on a ‘2nd-order’ procedure to find a new entry, then set the program off again with an extra digit in place ...] Most puzzles are solved with 4 or fewer program runs but some need also one or more second-order interventions. I chickened out of trying to program any second-order procedures.”

In other words, his program may be unable to solve the puzzle all by itself and may require one or more human interventions to provide some new, human-computed digits and resume. Gloomy indeed. Alas, despite these simplifications:

“A single run at an early stage takes about four minutes on a 49g+ [...] My whole program is very bitty, with 18 subroutines in five levels [...]”

Thus, the task at hand seems to be very difficult indeed, specially as it takes a fast, state-of-the-art HP49g+ four minutes per run (several of them) of a long (1 metre) and complicated program (18 subroutines in 5 levels executing 6 first-order procedures), not to mention occasional human intervention(s).

Certainly, Mr. Gatenby’s achievement deserves all our respect, but ... Can we try and do better with an old-fashioned dinosaur of the 80’s like the HP-71B, say? You bet!

1 No HP-71B, HP-IL ROM or Math ROM? No problem. Search the Web for Emu71, a free emulator for Windows (>30X faster), or HP-71X, an excellent emulator (2X) for your HP48/49.
Introducing SUDOKU71, a Sudoku solver program

SUDOKU71 is a very short program (35 lines, 1300+ bytes) I’ve written for the HP-71B which can solve any Sudoku puzzle without requiring human intervention. It is not based on Mr. Gatenby’s program (which I’ve not seen) or ideas but on my own approach instead, which includes advanced techniques such as bitboards and making good use of the powerful capabilities provided by the HP-71B system to recursively search till a solution is found, limited only by available RAM and time.

In practice it’ll solve most typical newspaper Sudoku puzzles in reasonable times, Mr. G.’s examples are solved in 2 min 23 s and 59 sec, respectively. See Examples for puzzles having from the usual 30 to as few as only 19 initially filled-up cells.

Here’s the full listing. For a thorough explanation, see Program details below:

SUDOKU71 (1,325 Bytes)

```plaintext
10 ! *****<<< SUDOKU Solver v1.0 - (c) Valentin Albillo, 2005 >>>*****
12 DESTROY ALL @ OPTION BASE 1 @ STD @ DIM P(9,9),S(9,9),E(9),F,X,Y,Z
14 DISP "Initializing ..." @ FOR I=1 TO 9 @ K=3*((I-1) DIV 3)+1
16 FOR J=1 TO 9 @ S(I,J)=K+(J-1) DIV 3 @ NEXT J @ E(I)=2^I @ NEXT I
18 DISP "Enter puzzle:"; @ MAT INPUT P
20 INPUT "Max. Depth (1 -15) = ","2";X @ X=INT(MAX(MIN(X,15),1))
22 INPUT "Max. Width (2 -9) = ","2";Y @ Y=INT(MAX(MIN(Y,15),2))
24 INPUT "Verbose (Y/N) ? ","N";R$ @ Z=R$#"N" @ DISP "Solving ..."
26 CALL TRY(P,F,1,X,Y,Z,S,E)
28 IF F=2 THEN DISP "SOLVED!"
30 IF F=3 THEN DISP "Illegal?" ELSE IF F=4 THEN DISP "No solution found"
32 MAT DISP USING "DX";P @ END
34 !
36 SUB TRY(P(,),F,W,X,Y,Z,S(,),E()) @ INTEGER T(9,9),A(81,4)
38 DIM R(9),C(9),B(9),D,I,J,U,V,K,M,N,H,L @ FOR I=1 TO 9
40 FOR J=1 TO 9 @ IF P(I,J) THEN H=S(I,J) @ D=E(P(I,J)) @ GOSUB 78
42 NEXT J @ NEXT I @ M=0
44 F=2 @ K=0 @ MAT T=P @ FOR I=1 TO 9 @ U=R(I) @ FOR J=1 TO 9
46 V=C(J) @ IF P(I,J) THEN 58 ELSE K=K+1 @ H=S(I,J) @ IF F=2 THEN F=0
48 L=BINOR(BINOR(BINOR(U,V),B(H)),B(I)) @ IF L=1022 THEN F=3 @ MAT P=T @ END
50 D=BINAND(BINCMP(L,1023),-1) @ N=BIT(D,1)+BIT(D,2)+BIT(D,3)
52 N=N+BIT(D,4)+BIT(D,5)+BIT(D,6)+BIT(D,7)+BIT(D,8)+BIT(D,9)
54 IF N=1 THEN P(I,J)=LOG2(D) @ F=1 @ M=M+1 @ GOSUB 78 @ GOTO 58
56 A(K,1)=I @ A(K,2)=J @ A(K,3)=N @ A(K,4)=L
58 NEXT J @ NEXT I @ IF F=1 THEN 44
60 IF F=2 THEN END ELSE IF Z THEN DISP W;:"":M;"forced cells"
62 IF W=X THEN F=4 @ END
64 MAT P=T @ FOR U=1 TO K @ IF A(U,3)>Y THEN 76
66 I=A(U,1) @ J=A(U,2) @ L=A(U,4) @ FOR V=1 TO 9 @ IF BIT(L,V) THEN 74
68 P(I,J)=V @ DISP W;:"$:Try";I+J/10;":";V @ CALL TRY(P,F,W+1,X,Y,Z,S,E)
70 IF F=2 THEN END ELSE IF F=3 AND Z THEN DISP W;:"":(dead end)"
72 MAT P=T
74 NEXT V
76 NEXT U @ F=4 @ END
78 R(I)=R(I)+D @ C(J)=C(J)+D @ B(H)=B(H)+D @ RETURN
```

Notes: The program requires approximately 1444 + 1735*MD bytes of free RAM available, where MD is the maximum depth of the search. It also makes use of keywords from the Math ROM and HP-IL ROM. Emu71 executes the program 40-70X faster on a typical 2.4 Ghz PC.
Programming details & techniques

Note: A brief explanation of *sudoku* puzzles is in order: you’re given a 9x9 grid, each cell to be occupied by a *single* digit 1-9, such that *each* of the 9 columns, rows, and 3x3 non-overlapping blocks contain all digits without repetition. Initially some cells are already filled-in and you must fill the rest.

Though a capable solver, SUDOKU71 is a no-frills, didactic program, and as such it has a very basic shell providing minimal input/output capabilities around its central core (subprogram TRY), which is where all advanced techniques and optimizations reside. For production quality, more advanced input (saving, loading, editing puzzles) and output (formatted grids, choice of output device, statistics) is mandatory, but that’s just the icing of the cake and for the purposes of this article a more focused approach is best. We discuss now the diverse sections & techniques.

The Shell

The twelve lines 10-34 are the *shell*, whose purpose is to do some initialization, input the puzzle and search specifications, call the subprogram which actually does all the work, and report the results to the user. Firstly, it initializes two *constant* arrays later used in the solving process:

```
14 DISP "Initializing ..." @ FOR I=1 TO 9 @ K=3*((I-1) DIV 3)+1
16 FOR J=1 TO 9 @ S(I,J)=K+(J-1) DIV 3 @ NEXT J @ E(I)=2^I @ NEXT I
```

This is done for speed, as it’s *much* faster to pass the initialized arrays to the search subprogram (which then uses mere array references to retrieve values) instead of wasting time recomputing them every time they’re needed. The array contents are:

- **S(9,9)**: Given the row-column cell coordinates, retrieves its block number
- **E(9)**: Retrieves precalculated integer powers of 2

Now the shell prompts the user to enter the puzzle. Thanks to the Math ROM advanced matrix capabilities, this is done using a *single* **MAT INPUT** command:

```
18 DISP "Enter puzzle:"; @ MAT INPUT P
```

The shell asks the user now for the search parameters (default values are provided), and validates them (using **INT**, **MAX**, **MIN**) to ensure they’re integer values in some specific range:

```
20 INPUT "Max. Depth (1-15) = ","2";X @ X=INT(MAX(MIN(X,15),1))
22 INPUT "Max. Width (2-9) = ","2";Y @ Y=INT(MAX(MIN(Y,15),2))
24 INPUT "Verbose (Y/N) ? ","N";R$ @ Z=R$#"N" @ DISP "Solving ..."
```

Input completed, the shell calls now the *search subprogram*, **TRY**, passing it all required arrays and parameters, one of them by value (**1**), the rest by reference:

```
26 CALL TRY(P,F,1,X,Y,Z,S,E)
```

Upon returning from the subprogram, the shell finds out the kind of result obtained and informs the user. The grid (*fully solved, partially solved, or intact*) is output as well with a *single* **MAT DISP USING** statement, and the program ends:

```
28 IF F=2 THEN DISP "SOLVED!"
30 IF F=3 THEN DISP "Illegal?" ELSE IF F=4 THEN DISP "No solution found"
32 MAT DISP USING "DX";P @ END
```
The Search
The 22 lines (36 to 78) comprise the search subprogram, TRY. This is a recursive subprogram which performs a depth-first search to a specified Maximum Depth by calling itself till this depth is reached, while applying simple heuristics and efficient techniques to keep the search manageable and achieve good performance.

For each depth level, TRY attempts to correctly fill in as many cells as possible by determining which cells admit only a single, forced value. Each such cell is assigned that value, and as every extra cell filled in can result in other cells being now forced, the process is iterated till no more forced cells remain. If no empty cells are left, a solution has been found and no further search is necessary. Else, provided the specified Maximum Depth is 2 or more, the recursive search begins:

For each empty cell, TRY determines all legal values and then tentatively tests a maximum of N of them (as per the Maximum Width parameter) by updating the grid with each in turn and recursively calling itself to solve the new, easier puzzle (1 cell less). This goes on till one of the following stopping criteria is met:

- If no empty cells are left, a solution has been found and the search ends immediately, returning from all recursive calls back to the calling shell. The grid is updated with the solution for the shell to report to the user.
- If a cell is found which is impossible to fill, the puzzle is inconsistent and there’s no solution. The search ends immediately and returns back to the shell. The grid is restored to its initial state for the user to examine it (maybe a typo?)
- If the specified maximum depth/width have been reached without finding a solution, the search ends. The grid is returned with as many cells correctly filled-in as possible. The user might simply increase the depth/width and restart the search, see Test case #2 in Examples below.

Now for the details:
TRY begins by accepting all required parameters and dimensioning two arrays (both are specified as INTEGER to save RAM, specially important since TRY will call itself recursively and each call will generate additional instances of those arrays. Thus, it’s advisable to use INTEGER for the largest arrays):

36 SUB TRY(P(,),F,W,X,Y,Z,S(,),E()) @ INTEGER T(9,9),A(81,4)
The parameters are as follows:

- P(,) is the puzzle grid (input/output variable) passed by reference so that the resulting grid after the search can be returned back to the caller
- F is an output variable (thus passed by reference) which returns a value indicating the results of the search (2=Solved, 3=Illegal, 4=Unsolved)
- W is an input, the current depth of the search, always passed by value
- X is an input, the Maximum Depth of the search
- Y is an input, the Maximum Width of the search
- Z is an input, a flag controlling the verbosity of search messages
- S(,) is an input, a constant array with the mapping of the cells to blocks
- E() is an input, a constant array with precalculated integer powers of 2
Now other local small arrays and variables are dimensioned. For HP-71B’s arrays and variables, REAL is faster than INTEGER, so for speed we only use INTEGER for large arrays (in fact, all variables used in this program could be declared INTEGER, and you might want to do so if getting out of free RAM while running it):

```
38 DIM R(9), C(9), B(9), D, I, J, U, V, K, M, N, H, L @ FOR I = 1 TO 9
```

A loop is now entered to generate the bitboards for all rows/columns/blocks. The grid is scanned for occupied cells and for each of them a call is made to a subroutine which simply sets the bit representing the value assigned to the cell in the bitboards corresponding to the row, column and block where the cell resides:

```
40 FOR J = 1 TO 9 @ IF P(I, J) THEN H = S(I, J) @ D = E(P(I, J)) @ GOSUB 78
42 NEXT J @ NEXT I @ M = 0
```

Assorted variables are initialized and a copy of the current grid is made. Now we loop to scan for all empty cells, and among them determine and fill in all forced ones (a single legal value) and find out and record all legal values for the rest. This is efficiently accomplished using the generated bitboards (see Bitboards below):

```
44 F = 2 @ K = 0 @ MAT T = P @ FOR I = 1 TO 9 @ U = R(I) @ FOR J = 1 TO 9
46 V = C(J) @ IF P(I, J) THEN 58 ELSE K = K + 1 @ H = S(I, J) @ IF F = 2 THEN F = 0
48 L = BINIOR(BINIOR(U, V), B(H)) @ IF L = 1022 THEN F = 3 @ MAT P = T @ END
50 D = BINAND(BINCMP(L, 1023), -1) @ N = BIT(D, 1) + BIT(D, 2) + BIT(D, 3)
52 N = N + BIT(D, 4) + BIT(D, 5) + BIT(D, 6) + BIT(D, 7) + BIT(D, 8) + BIT(D, 9)
```

If there’s just a single legal value, the cell is filled in at once and the subroutine is called again to update the corresponding cell’s bitboards (row/column/block):

```
54 IF N = 1 THEN P(I, J) = LOG2(D) @ F = 1 @ M = M + 1 @ GOSUB 78 @ GOTO 58
```

Else, the empty cell’s data (row, column, number of legal values, and the legal values themselves in bitboard format) are recorded for eventual further search:

```
56 A(K, 1) = I @ A(K, 2) = J @ A(K, 3) = N @ A(K, 4) = L
```

At loop termination, we check if any forced cells were assigned, and if so we go back to repeat the loop till no forced cells remain. Else, if no empty cells either, the puzzle is solved and the search ends. If not, the number of forced cells is reported:

```
58 NEXT J @ NEXT I @ IF F = 1 THEN 44
60 IF F = 2 THEN END ELSE IF Z THEN DISP W;": ";M;"forced cells"
```

At this point, the grid has been updated with all forced cells, but there are still empty cells which admit 2 or more legal values. If we’re already at the maximum search depth, we search no more but immediately return to the caller:

```
62 IF W = X THEN F = 4 @ END
```

Else, further recursive search is possible. For each recorded cell we traverse the list of its legal values (up to N of them, where N is the Maximum Width), update the grid with each value in turn, and recursively call TRY to solve the updated puzzle:

```
64 MAT T = P @ FOR U = 1 TO K @ IF A(U, 3) > Y THEN 76
66 I = A(U, 1) @ J = A(U, 2) @ L = A(U, 4) @ FOR V = 1 TO 9 @ IF BIT(L, V) THEN 74
68 P(I, J) = V @ DISP W; ":::Try"; I + J / 10; "="; V @ CALL TRY(P, F, W + 1, X, Y, Z, S, E)
```

Upon returning from this call, we test the result. If F = 2, the updated puzzle was solved by the call and we return immediately to the caller. If F = 3, an inconsistency was detected, so we report a “dead end” to the user (if Verbose).
70 IF F=2 THEN END ELSE IF F=3 AND Z THEN DISP W;":    (dead end)"

If the call didn’t solve the updated puzzle, we simply backtrack to the current puzzle and try another legal value for the cell, then another cell. If we exhaust all legal values and cells, we report failure (F=4) and end the search at this depth:

72 MAT P=T
74 NEXT V
76 NEXT U @ F=4 @ END

This subroutine updates a cell’s bitboards by setting the bit for the corresponding cell’s value. Usually, BINIOR boolean commands would be necessary, but as the bit being set is guaranteed not to be set on entry, sum operators will do, and faster:

78 R(I)=R(I)+D @ C(J)=C(J)+D @ B(H)=B(H)+D @ RETURN

Bitboards

Previous experience with Computer Chess made it plain to me that bitboards would be a natural for this puzzle. A bitboard is a binary representation of some aspect of the puzzle which is boolean in nature, such as “Does Row 1 already contain a 7?”. Once you’ve generated the bitboards, all kinds of questions can be answered very quickly by simply performing boolean operations upon them. SUDOKU71 makes extensive use of this technique. For instance, for the puzzle shown, we obtain these row/column/block bitboards:

<table>
<thead>
<tr>
<th>Rows bitboards</th>
<th>Columns bitboards</th>
<th>Blocks bitboards</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 0 0 0 0 0 1</td>
<td>0 0 0 0 1 1 1 1 0 0</td>
<td>0 0 0 1 1 1 0 0 0 0</td>
</tr>
<tr>
<td>0 0 1 1 0 0 0 0 0 0</td>
<td>0 0 0 0 0 0 1 0 0 1</td>
<td>0 0 1 0 0 1 0 0 0 0</td>
</tr>
<tr>
<td>0 1 0 0 1 1 0 0 0 0</td>
<td>1 1 0 0 0 0 0 0 0 0</td>
<td>1 0 0 1 0 0 1 0 0 0</td>
</tr>
<tr>
<td>0 0 0 0 0 0 0 0 1 1</td>
<td>0 1 0 1 1 1 1 0 0 0</td>
<td>0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>0 1 0 0 1 1 0 1 1 1</td>
<td>0 1 1 1 1 1 1 0 0 1</td>
<td>1 0 0 0 0 1 0 0 1</td>
</tr>
<tr>
<td>0 0 0 0 0 0 0 0 1 0</td>
<td>0 0 1 0 0 1 0 0 0 0</td>
<td>0 1 1 0 1 0 0 1</td>
</tr>
<tr>
<td>1 0 0 0 1 0 0 0 0 0</td>
<td>0 1 0 1 0 0 0 0 0 0</td>
<td>0 0 0 1 1 1 0 0 0</td>
</tr>
</tbody>
</table>

Using them, we can discover that cell 1,6 is forced to have the value 7 as follows:

\[ R(1) = \text{Bitboard of Row #1 :} 1 1 0 0 0 0 0 0 1 \text{ (digits used in Row #1=1,2,9)} \]
\[ C(6) = \text{Bitboard of Col #6 :} 0 0 1 1 0 0 1 0 0 1 \text{ (digits used in Column #6)} \]
\[ B(2) = \text{Bitboard of Block #2 :} 1 0 0 1 1 0 0 1 0 \text{ (digits used in Block #2)} \]
\[ L = R(1) \text{ OR } C(6) \text{ OR } B(2) : 1 1 1 1 1 1 0 1 1 \text{ (digits used in any of them)} \]
\[ D = \text{Complement of L :} 0 0 0 0 0 0 1 0 0 \text{ (digits legal for cell(1,6) = 7)} \]

This gives us all the legal values for the cell: just the value 7. The HP-IL ROM binary operations are used for manipulating the bitboards: BINIOR (OR), BINCMP (Complement), BINAND (to mask out the pertinent 9 bits), and BIT (to count the number of values set in a bitboard. Also, LOG2 from the Math ROM is used to extract the value (7) from the final bitboard (D). For maximum efficiency, all bitboards are updated dynamically each time a cell is assigned a value.
Usage instructions

• The following are sample inputs and outputs. To start the program, press:

```
[RUN]   --> Initializing ...

--> Enter puzzle:P(1,1)?
```

• Enter the contents of all cells (0 if empty), left to right, top to bottom, separated by commas. You can enter up to 48 cells at a time, but entering just a single row per prompt will make it probably easier for you to keep track. For example:

```
P(2,1)? 4,5,0,0,0,0,0,0,6 [ENTER]
```

```
P(2,1)? 0,0,0,0,3,0,1,0,7 [ENTER]
```

```
P(9,1)? 7,0,0,0,0,0,0,6,4 [ENTER]
```

• Now enter the Maximum Depth and Maximum Width of the search (or accept the default values) and specify whether you want Verbose output or not:

```
Max. Depth (1-15) = 2 [ENTER]
Max. Width (2-9) = 2 [ENTER]
Verbose (Y/N) ? N [ENTER]
```

• The search will proceed unattended until a solution is found or it exhausts Max. Depth/Width without finding one (see Examples for more sample outputs):

```
Solving ...
(depth level#) :>Try (cell) = (digit)
```

then eventually, the search results are reported, which will be one of these:

```
SOLVED! The subsequent grid is a full solution to the puzzle.
Illegal? The puzzle is inconsistent and has no solution.
No solution found The grid is output with as many correct digits filled in as found. Increase Max.Depth/Width, no need to re-enter the puzzle, just: CONT 20[ENTER]
```

Notes:

• Running time increases exponentially with Maximum Depth/Width, so it’s advisable to start with the lowest values (Max. Depth = 1, Max. Width=2) and increase them if no solution is found. Usually (but not always), it’s best to restrict Width to the range 2-4 and increase Depth instead. See Examples.

• Verbose output includes the number of digits forced at each node of the search as well as indicating when it has encountered a dead end and it’s backtracking.

• The program doesn’t alter the DELAY setting, use the one that suits you best.

• Should you miss the final outputs, you can re-output them by executing this right from the keyboard:

```
RUN 28 [ENTER]
```
Examples

Test case #1: 30 cells

>RUN
Initializing ...
Enter puzzle:
P(1,1)? 0,5,0,0,0,1,4,0,0 [ENTER]
P(2,1)? 2,0,3,0,0,0,7,0,0 [ENTER]
P(3,1)? 0,7,0,3,0,0,1,8,2 [ENTER]
P(4,1)? 0,0,4,0,5,0,0,0,7 [ENTER]
P(5,1)? 0,0,0,1,0,3,0,0,0 [ENTER]
P(6,1)? 8,0,0,0,2,0,6,0,0 [ENTER]
P(7,1)? 1,8,5,0,0,6,0,9,0 [ENTER]
P(8,1)? 0,0,2,0,0,0,8,0,3 [ENTER]
P(9,1)? 0,0,6,4,0,0,0,7,0 [ENTER]
Max. Depth (1-15) = 1 [ENTER] 6 5 8 2 7 1 4 3 9
Max. Width (2-9) = 2 [ENTER] 2 1 3 9 4 7 5 6
Verbose (Y/N) ? N [ENTER] 4 7 9 3 6 5 1 8 2
Solving ... 5 6 7 1 4 3 9 2 8
SOLVED! { HP71B: 55 seconds }
{ Emu71: 1 second }

Notes: This is a very simple case, and no recursion is needed at all (Depth =1)

Test case #2: 26 cells

>RUN
Initializing ...
Enter puzzle:
P(1,1)? 4,5,0,0,0,0,0,0,6 [ENTER]
P(2,1)? 0,0,3,0,0,1,0,0,7 [ENTER]
P(3,1)? 0,0,0,2,3,0,0,0,0 [ENTER]
P(4,1)? 0,0,0,4,0,2,5,0,0 [ENTER]
P(5,1)? 0,0,9,3,0,2,1,0,0 [ENTER]
P(6,1)? 0,8,1,0,7,0,0,0,0 [ENTER]
P(7,1)? 0,0,0,5,8,0,0,0,0 [ENTER]
P(8,1)? 9,0,0,7,0,0,8,0,0 [ENTER]
P(9,1)? 7,0,0,0,0,0,6,4 [ENTER]
Max. Depth (1-15) = 1 [ENTER] 4 5 2 8 9 7 3 1 6
Max. Width (2-9) = 2 [ENTER] 0 0 3 0 0 1 0 0 7
Verbose (Y/N) ? Y [ENTER] 0 0 0 0 2 3 0 0 0
Solving ... 0 0 9 3 0 2 1 0 8
1 : 10 forced cells
No solution found { 1 min 11 sec }

Notes: This case isn’t that simple, and trying to avoid recursion (Depth=1) doesn’t produce a full solution, though it was still able to fill in 10 forced cells. We simply
repeat the search, this time with **Depth=2**, which succeeds wholesale. No need to re-enter the puzzle, just type right from the command prompt:

>CONT 20

Max. Depth (1-15) = 2 [ENTER]
Max. Width (2-9) = 2 [ENTER]
Verbose (Y/N) ? Y [ENTER]

Solving ... 4 5 2 8 9 7 3 1 6
1 : 10 forced cells 8 9 3 6 5 1 4 2 7
1 :>Try 2.1 = 6 1 7 6 4 2 3 9 8 5
1 : (dead end) 6 3 7 1 4 8 2 5 9
1 :>Try 2.1 = 8 5 4 9 3 6 2 1 7 8
2 : 0 forced cells 2 8 1 9 7 5 6 4 3
1 :>Try 2.2 = 6 3 2 4 5 8 6 7 9 1
1 : (dead end) 9 6 5 7 1 4 8 3 2
1 :>Try 2.2 = 9 7 1 8 2 3 9 5 6 4

SOLVED!  ( HP71B: 2 min 29 sec )  ( Emu71: 4 seconds )

**Test case #3: 24 cells**

>RUN

Initializing ...

Enter puzzle:

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1</td>
<td>7</td>
<td>6</td>
<td>2</td>
<td>5</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6</td>
</tr>
</tbody>
</table>

Max. Depth (1-15) = 3 [ENTER]  SOLVED!  ( HP71B: 6 min 55 sec )
Max. Width (2-9) = 3 [ENTER]  ( Emu71: 11 seconds )
Verbose (Y/N) ? N [ENTER]

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

**Notes:** This is harder, so we need to increase **both** Depth and Width to 3, (3-3) from now on. (2-9) or (3-2) fails but (5-2) (trading Width for Depth) also succeeds.

A good strategy is to begin with (1-2) and increase them if no solution is found. For puzzles up to **24 cells** it’s best to increase **just** the Depth leaving the Width fixed at 2. Over **26 cells**, it’s best to first increase the Width, and only increase the Depth if even setting Width to 9 doesn’t succeed. For this puzzle, (3-3) to (3-9) all succeed equally fast while (5-2) also does but **33 times (!) slower.**
Test case #4: 22 cells (!)

> RUN
Initializing ...
Enter puzzle:
P(1,1)? 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 [ENTER]
P(2,1)? 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 [ENTER]
P(3,1)? 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 [ENTER]
P(4,1)? 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 [ENTER]
P(5,1)? 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 [ENTER]
P(6,1)? 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 [ENTER]
P(7,1)? 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 [ENTER]
P(8,1)? 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 [ENTER]
P(9,1)? 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 [ENTER]

Max. Depth (1-15) = 2 [ENTER] SOLVED! { HP71B: 19 min 58 sec }
Max. Width (2-9) = 3 [ENTER] { Emu71: 21 seconds }
Verbose (Y/N) ? N [ENTER]

Solving:
1 :> Try 1.2 = 3
1 :> Try 1.2 = 8
1 :> Try 1.2 = 9
...
1 :> Try 6.3 = 9
1 :> Try 6.8 = 1

Notes: Nice-looking puzzle, (2-3) solves it fastest. (3-2) does, too, but 4X slower.

Test case #5: 19 cells (!!)

> RUN
Initializing ...
Enter puzzle:
P(1,1)? 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 [ENTER]
P(2,1)? 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 [ENTER]
P(3,1)? 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 [ENTER]
P(4,1)? 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 [ENTER]
P(5,1)? 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 [ENTER]
P(6,1)? 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 [ENTER]
P(7,1)? 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 [ENTER]
P(8,1)? 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 [ENTER]
P(9,1)? 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 [ENTER]

Max. Depth (1-15) = 5 [ENTER] SOLVED! { HP71B: 7 hr 58 min }
Max. Width (2-9) = 2 [ENTER] { Emu71: 6 min 36 sec }
Verbose (Y/N) ? N [ENTER]

Solving ...
1 :> Try 4.3 = 1
2 :> Try 2.3 = 5
3 :> Try 1.3 = 4
...
4 :> Try 2.3 = 9
4 :> Try 3.5 = 6

Notes: The very hardest nut (!) but (5-2) cracks it. (3-4) would, too, albeit slower.