DOMINOKU – Swiftly Solving Domino Grids

© 2021 Valentín Albillo

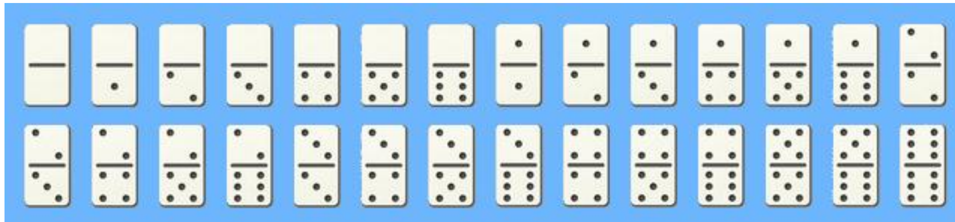
1. Introduction

Welcome to a new article, this time featuring a program I wrote years ago for the *HP-71B* to solve a newspaper-style domino puzzle which goes by assorted names (*"Domino Logic"*, *"- Hunt"*, *"- Dilemma"*, *"Dominosa"*, etc.) but I'm informally calling my solver *"Dominoku"* because this puzzle looks to me like a *"reverse Sudoku"* in the sense that in *Sudoku* you're given a fixed *9x9* grid with some of the cells having numbers *1-9* already in place and you must find out the missing ones, i.e.: *"given grid → find numbers"*, while here you're given all 56 numbers *(pips)* corresponding to the full set of 28 dominoes already placed on a grid, and you must find the internal boundaries of the grid, i.e.: *"given numbers → find grid"*, or in other words, the exact location for each domino.

The resulting program, as was the case with my *Sudoku* solver, is a nice example of how recursion can be extremely useful to simplify the programming by implementing a search procedure which calls itself to go deeper and deeper until the puzzle is completely *solved* or else determined to be *inconsistent* and thus have no solution.

2. The Puzzle

You're given the external boundary of a grid containing this set of 28 dominoes *("bones")* in some arrangement,



all of them represented by their *pips*, but their outlines aren't given and the solver must find them. This is, the solver must place every domino on top of two horizontally- or vertically-adjacent numbers on the grid, corresponding to its pips, such that in the end all 28 dominoes are placed, with no overlaps and neither unplaced dominoes nor uncovered locations left, like for instance this *7x8* rectangular grid puzzle and its solution:

1	4	4	4	4	4	0	0
1	2	1	6	6	2	2	4
1	2	0	0	0	6	6	6
5	2	0	2	0	0	2	2
1	3	3	3	3	5	5	5
4	3	3	3	6	6	5	5
4	5	1	1	1	6	5	3

→

Unlike the *9x9* square grid of *Sudoku*, the grid here can be any shape and even have holes inside (in which case simply embed it on a bigger square/rectangle) as long as it has *exactly* 56 occupied locations (*1x1* cells) and no more than *9* columns/rows, which may include empty locations as well. See ***Examples 2***, ***4***, ***5*** and ***7*** below.

3. Program Listing

```
 1  DESTROY ALL @ INTEGER F,C @ INPUT "Rows,Cols ?: ";F,C
 2  F=ABS(F) @ C=ABS(C) @ IF F>9 OR C>9 THEN "Rows and Cols must be <=9" @ END
 3  IF F*C<56 THEN DISP "Grid size too small to place all 28 dominoes" @ END
 4  OPTION BASE 0 @ INTEGER D(F+1,C+1) @ SFLAG -1 @ K=0 @ X=INF @ MAT D=(X)
 5  OPTION BASE 1 @ INTEGER B(28,15),U(F,C) @ MAT INPUT U @ FOR I=1 TO F
 6  FOR J=1 TO C @ D(I,J)=U(I,J) @ NEXT J @ NEXT I @ DESTROY U
 7  N=0 @ FOR I=1 TO F @ FOR J=1 TO C @ N=N+(D(I,J)#X) @ NEXT J @ NEXT I
 8  IF N#56 THEN DISP "There must be exactly 56 available cells, not";N @ END
 9  FOR I=0 TO 6 @ FOR J=I TO 6 @ K=K+1 @ B(K,1)=I @ B(K,2)=J @ NEXT J @ NEXT I
10  SETTIME 0 @ CALL SOLVEP(D,B,1,R) @ DISP @ CFLAG 5 @ CFLAG -1
11  IF R= 0 THEN DISP "Solved in";TIME;"seconds !" @ CALL PGRID(B,F,C)
12  IF R= 1 THEN DISP "Unsolved: recursion failed"
13  IF R=-1 THEN DISP "Inconsistent: unplayable domino"
14  IF R=-2 THEN DISP "Inconsistent: unplayable location"

15  SUB FPOS(B(,),P,D(,)) @ FOR K=P TO UBND(B,1) @ X=B(K,1) @ Y=B(K,2) @ Z=X=Y
16  N=0 @ FOR I=1 TO UBND(D,1)-1 @ FOR J=1 TO UBND(D,2)-1 @ IF D(I,J)#X THEN 21
17  H=10*I+J @ M=101*H @ IF D(I,J+1)=Y THEN N=N+1 @ B(K,N+3)=M+1
18  IF D(I+1,J)=Y THEN N=N+1 @ B(K,N+3)=M+10
19  IF Z THEN 21 ELSE IF D(I,J-1)=Y THEN N=N+1 @ B(K,N+3)=M-1
20  IF D(I-1,J)=Y THEN N=N+1 @ B(K,N+3)=M-10
*21 NEXT J @ NEXT I @ B(K,3)=N @ NEXT K

22  SUB SRTB(B(,),P) @ L=UBND(B,1)
23  FOR I=P TO L @ FOR J=I+1 TO L @ U=B(I,3) @ V=B(J,3) @ IF U<=V THEN 25
24  FOR K=1 TO 3+MAX(U,V) @ H=B(I,K) @ B(I,K)=B(J,K) @ B(J,K)=H @ NEXT K
*25 NEXT J @ NEXT I

26  SUB FDOM(B(,),P,D(,),R) @ L=UBND(B,1) @ F=UBND(D,1)-1 @ C=UBND(D,2)-1
27  CFLAG 5 @ FOR I=1 TO F @ FOR J=1 TO C @ X=D(I,J) @ IF X=INF THEN 32
28  N=0 @ A=I @ Z=J+1 @ GOSUB 33 @ Z=J-1 @ GOSUB 33 @ IF N>1 THEN 32
29  Z=J @ A=I+1 @ GOSUB 33 @ IF N>1 THEN 32
30  A=I-1 @ GOSUB 33 @ IF N=0 THEN R=-2 @ END
31  IF N=1 AND B(H,3)#1 THEN B(H,3)=1 @ B(H,4)=100*S+T @ SFLAG 5
*32 NEXT J @ NEXT I @ END
*33 Y=D(A,Z) @ IF Y=INF THEN RETURN
34  FOR K=P TO L @ M=B(K,1) @ W=B(K,2)
35  IF M=X AND W=Y THEN N=N+1 @ H=K @ S=10*I+J @ T=10*A+Z @ RETURN
36  IF M=Y AND W=X THEN N=N+1 @ H=K @ T=10*I+J @ S=10*A+Z @ RETURN
37  NEXT K @ RETURN

38  SUB PGRID(B(,),F,C) @ X=2*F+1 @ Y=4*C+1 @ DIM G$(X)[Y] @ FOR I=1 TO X
39  IF MOD(I,2) THEN G$(I)=RPT$(".",Y) ELSE G$(I)=RPT$(":   ",C)&":"
40  NEXT I @ FOR I=1 TO UBND(B,1) @ P$=STR$(B(I,4))
41  F1=2*VAL(P$[1,1]) @ C1=4*VAL(P$[2,2])-1 @ G$(F1)[C1,C1]=STR$(B(I,1))
42  F2=2*VAL(P$[3,3]) @ C2=4*VAL(P$[4,4])-1 @ G$(F2)[C2,C2]=STR$(B(I,2))
43  IF F1=F2 THEN C1=(C1+C2)/2 @ G$(F1)[C1,C1]="-"
44  IF C1=C2 THEN F1=(F1+F2)/2 @ G$(F1)[C1-2,C1+2]=": - :"
45  NEXT I @ FOR I=1 TO X @ DISP G$(I) @ NEXT I

46  SUB PLCB(D(,),B(,),K,I) @ P$=STR$(B(K,I+3))
47  D(VAL(P$[1,1]),VAL(P$[2,2]))=INF @ D(VAL(P$[3,3]),VAL(P$[4,4]))=INF

48  SUB SOLVEP(D(,),B(,),P,R) @ L=UBND(B,1) @ N=P
*49 CALL FPOS(B,P,D) @ CALL SRTB(B,P) @ DISP "."; @ IF B(P,3)=0 THEN R=-1 @ END
50  IF B(P,3)=1 THEN GOSUB 53
51  CALL FDOM(B,P,D,R) @ IF R=-2 THEN END ELSE IF FLAG(5) THEN CALL SRTB(B,P) @ GOSUB 53
52  IF P=N THEN 55 ELSE IF P<=L THEN N=P @ GOTO 49 ELSE R=0 @ END
*53 FOR K=P TO L @ IF B(K,3)=1 THEN P=P+1 @ CALL PLCB(D,B,K,1)
54  NEXT K @ RETURN
*55 OPTION BASE 0 @ INTEGER D2(0,0) @ OPTION BASE 1 @ INTEGER B2(1,1)
56  FOR K=P TO L @ FOR I=1 TO B(K,3) @ MAT D2=D @ MAT B2=B @ P2=P+1
57  CALL PLCB(D2,B,K,I) @ B2(K,3)=1 @ B2(K,4)=B(K,I+3) @ CALL SRTB(B2,P) @ R=0
58  CALL SOLVEP(D2,B2,P2,R) @ IF R=0 THEN MAT D=D2 @ MAT B=B2 @ P=P2 @ END
59  NEXT I @ NEXT K @ R=1
```

4. Program Characteristics

This *BASIC* program is 59 lines long (~ 200 statements, 2,505 bytes) and for speed and convenience it uses several matrix keywords from the ***Math Pac*** *ROM*. which must be available (either physically plugged in or as a virtual *ROM* image.) It also uses keywords from the *STRINGLX LEX* file, which must be available as well. If unavailable, those keywords can be easily replaced by simple equivalent *BASIC* constructs, albeit at a speed and size penalty.

4.1. Program structure

The program consists of a *main* section and *6 subprograms*, namely:

- **Main** section

  It inputs the number of row/columns of the grid to solve, validates them and declares and dimensions the arrays used by the program, then asks for the grid's contents (the numbers representing the pips for each domino, or **X** for any empty locations) and checks that exactly 56 non-empty locations have been entered.

  Once the puzzle is correctly entered, it performs some additional initialization and calls the **SOLVEP** subprogram to solve the puzzle. Upon return the result is checked and the puzzle is either declared *solved*, in which case the timing is printed, as well as the solved grid via a call to subprogram **PGRID**, or else the puzzle is declared *inconsistent* either because there's at least one domino which can't be placed on any location or there's a location where no domino can be placed, and in both cases there's no solution possible.

  The result can also indicate that the recursion *failed* to solve the puzzle, but this isn't expected to ever happen when attempting to solve a legal, actually solvable puzzle.

- Subprogram **FPOS** *(Find Positions)*

  Computes and fills up the array *("bones")* of legal locations for each domino (from a given index up) with the coordinates of all locations where each domino can be legally placed, as well as their number.

- Subprogram **SRTB** *(Sort Bones)*

  Sorts the *bones* array from a given index up in ascending order by the number of legal positions for each domino.

- Subprogram **FDOM** *(Find Dominoes)*

  For each location on the grid tries to find a (unique) domino which can be legally placed at that location, or the fact that none can.

- Subprogram **PGRID** *(Print Grid)*

  Prints the solved puzzle, showing all the grid internal boundaries, i.e., the outlines of all 28 dominoes placed in their correct locations and orientations.

- Subprogram **PLCB** *(Place Bone)*

  Updates the grid by marking the locations occupied by a given (forced) domino as *"unavailable"*.

- Subprogram **SOLVEP** *(Solve Puzzle)*

  Recursive subprogram which does all the solving. First, it obtains the legal locations for all the dominoes from some index up by calling **FPOS**, which are then sorted in ascending order by the number of legal locations via a call to **SRTB**. If the lowest number of locations is ***0*** then it sets the result as *"Inconsistent: unplayable domino"* and returns immediately to the previous level. If it is ***1***, then there's a forced location for this domino and all others which also have just one possible location so it marks all those locations as *"unavailable"* by calling **PLCB** in a loop.

Next, a call to **FDOM** finds those locations that admit just one or no dominoes at all. If the latter, then sets the result as *"Inconsistent: unplayable location"* and returns immediately to the previous level. If the former, then there's a forced domino for one or more locations so it calls **SRTB** and marks all those locations as *"unavailable"* by calling **PLCB** in a loop, as for the previous case.

Now it checks whether the puzzle is already fully solved, in which case it sets the result as *"Solved"* and returns immediately to the previous level. If not, it checks whether some new forced locations and/or forced dominoes were indeed found and if yes it goes again to try and find additional ones in the updated, fuller grid. On the other hand, if none were found (and so the current attempt was fruitless and the grid remains unchanged), it proceeds to tentatively try placing every still-pending domino in every legal location for that domino, one by one, and recursively calls itself (**SOLVEP**) to try and solve the new resulting grid.

Upon returning from the recursive call the result is checked and if the puzzle was solved then the arrays and index are updated and it returns immediately to the previous level. Else, the next location for the current domino is tried, then the next pending domino. If eventually all are exhausted without success, then the result is set *as "Unsolved: recursion failed"* before returning to the previous level. However, this is not expected to ever occur for legal, actually solvable puzzles.

4.2. Data structures

The main data structures are two permanent arrays (**D**, **B**) and a temporary auxiliar one (**U**), defined as follows:

OPTION BASE 0 @ INTEGER D(F+1,C+1) *holds the FxC grid contents, includes the external borders*

The external borders and any empty locations inside the grid are marked as "*unavailable"* for placing dominoes, as are the locations of all forced dominoes eventually found by the search, upon finding them.

OPTION BASE 1 @ INTEGER B(28,15) *holds all 28 dominoes and their possible locations; is kept sorted by #locations*

B(k,1)	*pips in the first end of domino **k***	*e.g.:*	*1*
B(k,2)	*pips in the second end of domino **k***	*e.g.:*	*5*
B(k,3)	*number **n** of legal locations for domino **k*** ($n \leq 12$)	*e.g.:*	*3*
B(k,4)	*coordinates* $r_1c_1r_2c_2$ *for location #**1** of domino **k***	*e.g.:*	*1222*
...	...		
B(k,3+n)	*coordinates* $r_1c_1r_2c_2$ *for location #**n** of domino **k***	*e.g.:*	*7161*

OPTION BASE 1 @ INTEGER U(F,C) *holds the initial user-supplied grid contents; copied to D, then destroyed*

4.3. Memory requirements

The memory required to successfully solve a puzzle depends on the size of the grid and on the maximum depth (*MaxDepth*) necessary to recursively solve it. For the typical *7x8* rectangular grid, such as the one in ***Example 1***, the memory required is ~ ***1,776 + 1,730 * (MaxDepth - 1)*** *bytes*, where the minimum *MaxDepth* is ***1*** if recursion isn't needed and the maximum depends on the particular puzzle.

For instance, solving ***Example 1*** goes up to *MaxDepth* ***9***, which means *~15,616 bytes* must be available to successfully solve it[1] (plus the 2,505 bytes required to load the program's source code), thus some additional memory (*RAM* modules, physical or virtual) might need to be added to the basic *16 Kb HP-71B*.

Most other puzzles aren't that demanding e.g.: ***Example 2*** is solved at *MaxDepth* ***1*** and thus only needs ~ *1,885 bytes* available, while ***Example 6*** requires *MaxDepth* ***7***, which means *~12,119 bytes* must be available. If you get an **Insufficient Memory** error message while solving a puzzle you'll probably need to make extra *RAM* available but see the last paragraph in Section ***Where to go now*** below.

---

[1] The high *MaxDepth* is due to ***Example 1*** having many solutions *(36)*, which makes it difficult to find forced dominoes or locations because many dominoes can go in several locations and many locations admit several dominoes, so we have the seemingly paradoxical fact that the "easiest" puzzle is the hardest to solve in terms of *MaxDepth* required (and thus, *RAM*) !

5. Usage instructions

To solve a puzzle, such as for instance the following *8x8* grid with an 8-cell hole inside, proceed as follows:

4	**2**	**3**	**0**	**3**	**6**	**4**	**2**
4	**2**	**5**	**0**	**1**	**5**	**0**	**1**
2	**3**	**4**	–	**4**	**6**	**4**	**0**
6	**6**	**3**	–	–	–	**2**	**0**
1	**5**	–	–	–	**3**	**1**	**1**
0	**4**	**5**	**5**	–	**1**	**3**	**1**
4	**1**	**6**	**2**	**6**	**6**	**3**	**0**
2	**6**	**3**	**5**	**0**	**5**	**5**	**2**

RUN → *Rows,Cols ?:*

It is asking for the number of rows and columns, which must be ≤ 9, else you'll get the error message "**Rows and Cols must be <=9**" and the program ends.

8,8 **END LINE** →[1] *U(1,1)?*

Now it is asking for the grid contents, which you must enter as integers ***0-6*** separated by commas and using **x** (or **X**) for any empty locations, like this:

```
4,2,3,0,3,6,4,2,4,2,5,0,1,5,0,1,2,3,4,x,4,6,4,0,6,6,3,x,x,x,2,0  END LINE  →  U(5,1)?
1,5,x,x,x,3,1,1,0,4,5,5,x,1,3,1,4,1,6,2,6,6,3,0,2,6,3,5,0,5,5,2  END LINE  →  .......
```

Notice that you can enter as many elements per line as you want (as long as they fit in 95 characters) and then press **END LINE** to assign them and continue entering more when asked for the next element to enter. Above, we entered the first 4 rows at once, then the last 4 rows, using **x** to specify the empty locations.

Once all grid elements have been entered, the program checks that there are exactly 56 non-empty locations, else it displays "**There must be exactly 56 available cells, not *nn***" and the program ends. If all's Ok, the program proceeds to find a solution, showing dots as the search progresses, and once found it outputs the timing and the solution just found and program execution ends:

```
→  Solved in 3.82 seconds !
→   ...............................
    : 4 : 2 : 3 – 0 : 3 – 6 : 4 : 2 :
    : – : – :..............: – : – :
    : 4 : 2 : 5 – 0 : 1 – 5 : 0 : 1 :
    ...............................
    : 2 – 3 : 4 :   : 4 – 6 : 4 : 0 :
    .......: – :...........: – : – :
    : 6 – 6 : 3 :   :   :   : 2 : 0 :
    ...............................
    : 1 : 5 :   :   :   : 3 – 1 : 1 :
    : – : – :..................: – :
    : 0 : 4 : 5 : 5 :   : 1 : 3 : 1 :
    .......: – : – :...: – : – :....
    : 4 – 1 : 6 : 2 : 6 : 6 : 3 : 0 :
    ...............: – :.......: – :
    : 2 – 6 : 3 – 5 : 0 : 5 – 5 : 2 :
    ...............................
```

If no solution was found, you'll get one of these messages instead:

"Inconsistent: unplayable domino"

One of more dominoes wouldn't fit at any available locations during the search.

"Inconsistent: unplayable location"

One of more locations wouldn't admit any pending dominoes during the search.

"Unsolved: recursion failed"

The recursion completed without finding any solution.

In all three cases, either the puzzle is *unsolvable* or you made an error while entering the grid's contents. Use the *command stack* to check what you actually entered and see if there are any typos, and if yes then run again the program and use the *command stack* to correct the typos on the fly while you're reentering the grid's elements.

[1] If the grid dimensions are too small to place all dominoes, it outputs **Grid size too small to place all 28 dominoes** and ends.

6. Examples

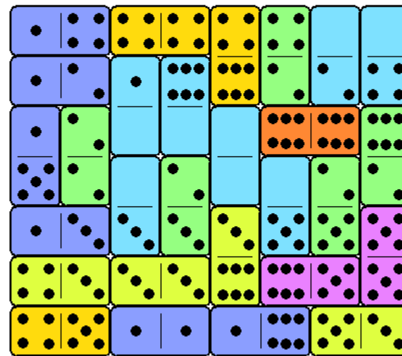
Note: *All runtimes in this article are for Jean-François Garnier's* ***Emu71/DOS*** *running on a 2.4 Ghz single-core CPU under Windows XP.*

6.1 Example 1:

1	**4**	**4**	**4**	**4**	**4**	**0**	**0**
1	**2**	**1**	**6**	**6**	**2**	**2**	**4**
1	**2**	**0**	**0**	**0**	**6**	**6**	**6**
5	**2**	**0**	**2**	**0**	**0**	**2**	**2**
1	**3**	**3**	**3**	**3**	**5**	**5**	**5**
4	**3**	**3**	**3**	**6**	**6**	**5**	**5**
4	**5**	**1**	**1**	**1**	**6**	**5**	**3**

```
[RUN] →
Rows,Cols ?:  7,8 [END LINE]
U(1,1)?  1,4,4,4,4,4,0,0,1,2,1,6,6,2,2,4,1,2,0,0,0,6,6,6 [END LINE]
U(4,1)?  5,2,0,2,0,0,2,2,1,3,3,3,3,5,5,5,4,3,3,3,6,6,5,5
         ,4,5,1,1,1,6,5,3 [END LINE]
→  .........
→  Solved in 4.15 seconds !
```
{ *17' 51" on a physical HP-71B, MaxDepth = 9* }

```
.................................
: 1 - 4 : 4 - 4 : 4 : 4 : 0 : 0 :
................: - : - : - : - :
: 1 - 2 : 1 : 6 : 6 : 2 : 2 : 4 :
........: - : - :................
: 1 : 2 : 0 : 0 : 0 : 6 - 6 : 6 :
: - : - :.......: - :.......: - :
: 5 : 2 : 0 : 2 : 0 : 0 : 2 : 2 :
........: - : - :...: - : - :....
: 1 - 3 : 3 : 3 : 3 : 5 : 5 : 5 :
................: - :.......: - :
: 4 - 3 : 3 - 3 : 6 : 6 - 5 : 5 :
.................................
: 4 - 5 : 1 - 1 : 1 - 6 : 5 - 3 :
.................................
```
→

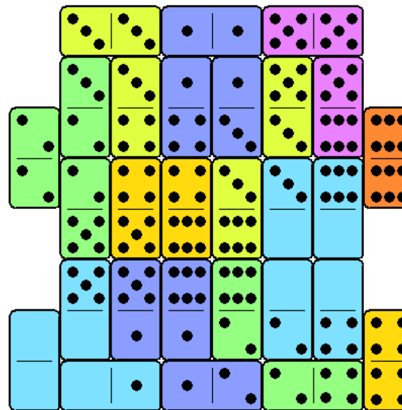


6.2 Example 2:

–	**3**	**3**	**1**	**1**	**5**	**5**	–
–	**3**	**3**	**1**	**1**	**5**	**5**	–
2	**2**	**4**	**4**	**3**	**3**	**6**	**6**
2	**2**	**4**	**4**	**3**	**3**	**6**	**6**
–	**5**	**5**	**6**	**6**	**0**	**0**	–
–	**5**	**5**	**6**	**6**	**0**	**0**	–
0	**0**	**1**	**1**	**2**	**2**	**4**	**4**
0	**0**	**1**	**1**	**2**	**2**	**4**	**4**

```
[RUN] →
Rows,Cols ?:  8,8 [END LINE]
U(1,1)?  x,3,3,1,1,5,5,x,x,3,3,1,1,5,5,x,2,2,4,4,3,3,6,6,2,2,4,4,3,3,6,6 [EoL]
U(5,1)?  x,5,5,6,6,0,0,x,x,5,5,6,6,0,0,x,0,0,1,1,2,2,4,4,0,0,1,1,2,2,4,4 [EoL]
→  .........
→  Solved in 0.82 seconds !
```
{ *3' 24" on a physical HP-71B, MaxDepth = 1* }

```
.................................
:   : 3 - 3 : 1 - 1 : 5 - 5 :   :
.................................
:   : 3 : 3 : 1 : 1 : 5 : 5 :   :
....: - : - : - : - : - : - :....
: 2 : 2 : 4 : 4 : 3 : 3 : 6 : 6 :
: - :.......................: - :
: 2 : 2 : 4 : 4 : 3 : 3 : 6 : 6 :
....: - : - : - : - : - : - :....
:   : 5 : 5 : 6 : 6 : 0 : 0 :   :
.................................
:   : 5 : 5 : 6 : 6 : 0 : 0 :   :
....: - : - : - : - : - : - :....
: 0 : 0 : 1 : 1 : 2 : 2 : 4 : 4 :
: - :.......................: - :
: 0 : 0 - 1 : 1 - 2 : 2 - 4 : 4 :
.................................
```
→



6.3 Example 3:

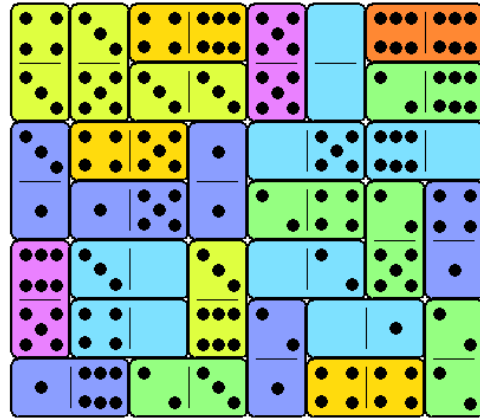
4	3	4	6	5	0	6	6
3	5	3	3	5	0	2	6
3	4	5	1	0	5	6	0
1	1	5	1	2	4	2	4
6	3	0	3	0	2	5	1
5	4	0	6	2	0	1	2
1	6	2	3	1	4	4	2

```
[RUN] →
Rows,Cols ?:  7,8 [END LINE]
U(1,1)?  4,3,4,6,5,0,6,6,3,5,3,3,5,0,2,6,3,4,5,1,0,5,6,0 [END LINE]
U(4,1)?  1,1,5,1,2,4,2,4,6,3,0,3,0,2,5,1,5,4,0,6,2,0,1,2
         ,1,6,2,3,1,4,4,2 [END LINE]
→  .........
→  Solved in 6.39 seconds !
```

{ *27' 40" on a physical HP-71B, MaxDepth = 9* }

```
.................................
: 4 : 3 : 4 - 6 : 5 : 0 : 6 - 6 :
: - : - :.......: - : - :.......
: 3 : 5 : 3 - 3 : 5 : 0 : 2 - 6 :
.................................
: 3 : 4 - 5 : 1 : 0 - 5 : 6 - 0 :
: - :.......: - :................
: 1 : 1 - 5 : 1 : 2 - 4 : 2 : 4 :
.........................: - : - :
: 6 : 3 - 0 : 3 : 0 - 2 : 5 : 1 :
: - :.......: - :................
: 5 : 4 - 0 : 6 : 2 : 0 - 1 : 2 :
.................: - :.......: - :
: 1 - 6 : 2 - 3 : 1 : 4 - 4 : 2 :
.................................
```

→



6.4 Example 4:

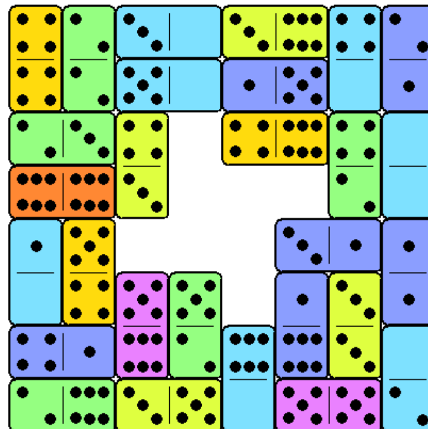
4	2	3	0	3	6	4	2
4	2	5	0	1	5	0	1
2	3	4	–	4	6	4	0
6	6	3	–	–	–	2	0
1	5	–	–	–	3	1	1
0	4	5	5	–	1	3	1
4	1	6	2	6	6	3	0
2	6	3	5	0	5	5	2

```
[RUN] →
Rows,Cols ?:  8,8 [END LINE]
U(1,1)?  4,2,3,0,3,6,4,2,4,2,5,0,1,5,0,1,2,3,4,x,4,6,4,0,6,6,3,x,x,x,2,0 [EoL]
U(5,1)?  1,5,x,x,x,3,1,1,0,4,5,5,x,1,3,1,4,1,6,2,6,6,3,0,2,6,3,5,0,5,5,2 [EoL]
→  .........
→  Solved in 3.82 seconds !
```

{ *16' 32" on a physical HP-71B, MaxDepth = 2* }

```
.................................
: 4 : 2 : 3 - 0 : 3 - 6 : 4 : 2 :
: - : - :.................: - : - :
: 4 : 2 : 5 - 0 : 1 - 5 : 0 : 1 :
.................................
: 2 - 3 : 4 :   : 4 - 6 : 4 : 0 :
.........: - :.............: - : - :
: 6 - 6 : 3 :   :   :   : 2 : 0 :
.................................
 : 1 : 5 :   :   :   : 3 - 1 : 1 :
: - : - :.......................: - :
: 0 : 4 : 5 : 5 :   : 1 : 3 : 1 :
.........: - : - :...: - : - :....
: 4 - 1 : 6 : 2 : 6 : 6 : 3 : 0 :
.................: - :.......: - :
: 2 - 6 : 3 - 5 : 0 : 5 - 5 : 2 :
.................................
```

→



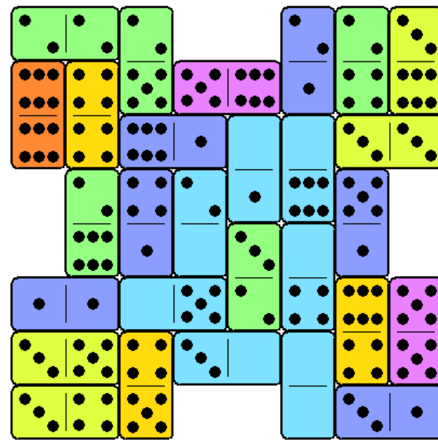
6.5 Example 5:

2	2	2	–	–	2	2	3
6	4	5	5	6	1	4	6
6	4	6	1	0	0	3	3
–	2	4	2	1	6	5	–
–	6	1	0	3	0	1	–
1	1	0	5	2	4	6	5
3	5	4	3	0	0	4	5
3	4	5	–	–	0	3	1

RUN →
Rows,Cols ?: 8,8 **END LINE**
U(1,1)? 2,2,2,x,x,2,2,3,6,4,5,5,6,1,4,6,6,4,6,1,0,0,3,3,x,2,4,2,1,6,5,x **EoL**
U(5,1)? x,6,1,0,3,0,1,x,1,1,0,5,2,4,6,5,3,5,4,3,0,0,4,5,3,4,5,x,x,0,3,1 **EoL**
→ ....
→ *Solved in* ***2.84*** *seconds !* { *12' 14" on a physical HP-71B, MaxDepth = 1* }

```
.................................
: 2 - 2 : 2 :   :   : 2 : 2 : 3 :
........: - :.......: - : - : - :
: 6 : 4 : 5 : 5 - 6 : 1 : 4 : 6 :
: - : - :........................
: 6 : 4 : 6 - 1 : 0 : 0 : 3 - 3 :
................: - : - :........
:   : 2 : 4 : 2 : 1 : 6 : 5 :   :
....: - : - : - :.......: - :....
:   : 6 : 1 : 0 : 3 : 0 : 1 :   :
................: - : - :........
: 1 - 1 : 0 - 5 : 2 : 4 : 6 : 5 :
........................: - : - :
: 3 - 5 : 4 : 3 - 0 : 0 : 4 : 5 :
........: - :.......: - :........
: 3 - 4 : 5 :   :   : 0 : 3 - 1 :
.................................
```

→



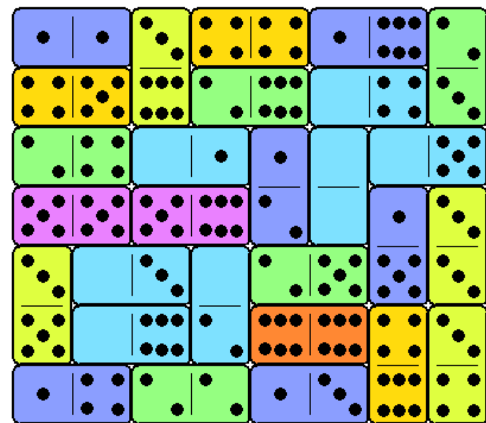
6.6 Example 6:

1	1	3	4	4	1	6	2
4	5	6	2	6	0	4	3
2	4	0	1	1	0	0	5
5	5	5	6	2	0	1	3
3	0	3	0	2	5	5	3
5	0	6	2	6	6	4	3
1	4	2	2	1	3	6	4

RUN →
Rows,Cols ?: 7,8 **END LINE**
U(1,1)? 1,1,3,4,4,1,6,2,4,5,6,2,6,0,4,3,2,4,0,1,1,0,0,5,5,5,5,6,2,0,1,3 **EoL**
U(5,1)? 3,0,3,0,2,5,5,3,5,0,6,2,6,6,4,3,1,4,2,2,1,3,6,4 **EoL**
→ ....
→ *Solved in* ***8.75*** *seconds !* { *37' 42" on a physical HP-71B, MaxDepth = 7* }

```
.................................
: 1 - 1 : 3 : 4 - 4 : 1 - 6 : 2 :
........: - :...............: - :
: 4 - 5 : 6 : 2 - 6 : 0 - 4 : 3 :
.................................
: 2 - 4 : 0 - 1 : 1 : 0 : 0 - 5 :
................: - : - :........
: 5 - 5 : 5 - 6 : 2 : 0 : 1 : 3 :
........................: - : - :
: 3 : 0 - 3 : 0 : 2 - 5 : 5 : 3 :
: - :.......: - :................
: 5 : 0 - 6 : 2 : 6 - 6 : 4 : 3 :
........................: - : - :
: 1 - 4 : 2 - 2 : 1 - 3 : 6 : 4 :
.................................
```

→

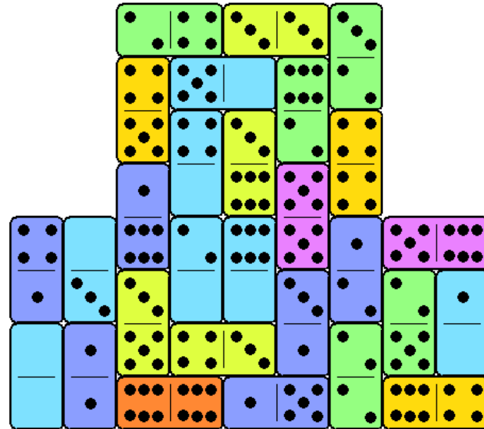


6.7 Example 7:

–	–	2	4	3	3	3	–	–
–	–	4	5	0	6	2	–	–
–	–	5	4	3	2	4	–	–
–	–	1	0	6	5	4	–	–
4	0	6	2	6	5	1	5	6
1	3	3	0	0	3	2	2	1
0	1	5	4	3	1	2	5	0
0	1	6	6	1	5	2	6	4

[RUN] →
Rows,Cols ?: 8,9 **[END LINE]**
U(1,1)? x,x,2,4,3,3,3,x,x,x,x,4,5,0,6,2,x,x,x,x,5,4,3,2 **[EoL]**
U(3,7)? 4,x,x,x,x,1,0,6,5,4,x,x,4,0,6,2,6,5,1,5,6,1,3,3 **[EoL]**
U(6,4)? 0,0,3,2,2,1,0,1,5,4,3,1,2,5,0,0,1,6,6,1,5,2,6,4 **[EoL]**
→ ....
→ *Solved in* ***3.06*** *seconds !* { *13' 11" on a physical HP-71B, MaxDepth = 1* }

```
.....................................
:   :   : 2 - 4 : 3 - 3 : 3 :   :   :
........................: - :........
:   :   : 4 : 5 - 0 : 6 : 2 :   :   :
........: - :.......: - :............
:   :   : 5 : 4 : 3 : 2 : 4 :   :   :
............: - : - :...: - :........
:   :   : 1 : 0 : 6 : 5 : 4 :   :   :
........: - :.......: - :............
: 4 : 0 : 6 : 2 : 6 : 5 : 1 : 5 - 6 :
: - : - :...: - : - :...: - :........
: 1 : 3 : 3 : 0 : 0 : 3 : 2 : 2 : 1 :
........: - :.......: - :...: - : - :
: 0 : 1 : 5 : 4 - 3 : 1 : 2 : 5 : 0 :
: - : - :...............: - :........
: 0 : 1 : 6 - 6 : 1 - 5 : 2 : 6 - 4 :
.....................................
```

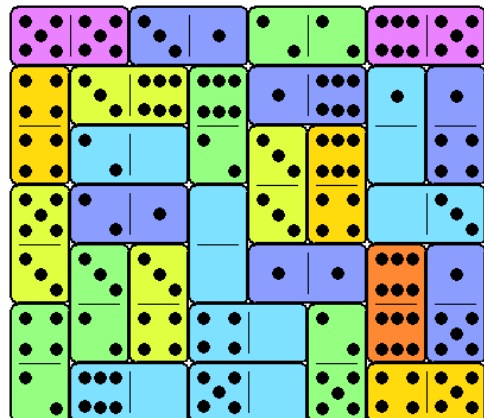


6.8 Example 8[1]:

5	5	3	1	2	2	6	5
4	3	6	6	1	6	1	1
4	2	0	2	3	6	0	4
5	2	1	0	3	4	0	3
3	3	3	0	1	1	6	1
4	2	4	4	0	2	6	5
2	6	0	5	0	5	4	5

[RUN] →
Rows,Cols ?: 7,8 **[END LINE]**
U(1,1)? 5,5,3,1,2,2,6,5,4,3,6,6,1,6,1,1,4,2,0,2,3,6,0,4 **[EoL]**
U(4,1)? 5,2,1,0,3,4,0,3,3,3,3,0,1,1,6,1,4,2,4,4,0,2,6,5,2,6,0,5,0,5,4,5 **[EoL]**
→ ....
→ *Solved in* ***5.79*** *seconds !* { *24' 52" on a physical HP-71B, MaxDepth = 3* }

```
.................................
: 5 - 5 : 3 - 1 : 2 - 2 : 6 - 5 :
.................................
: 4 : 3 - 6 : 6 : 1 - 6 : 1 : 1 :
: - :.......: - :.......: - : - :
: 4 : 2 - 0 : 2 : 3 : 6 : 0 : 4 :
................: - : - :........
: 5 : 2 - 1 : 0 : 3 : 4 : 0 - 3 :
: - :.......: - :................
: 3 : 3 : 3 : 0 : 1 - 1 : 6 : 1 :
....: - : - :...........: - : - :
: 4 : 2 : 4 : 4 - 0 : 2 : 6 : 5 :
: - :...............: - :........
: 2 : 6 - 0 : 5 - 0 : 5 : 4 - 5 :
.................................
```



[1] This example is a very good demonstration of the solving process, as it requires recursion up to *Depth 3* and at the very beginning *(Depth 1)* there aren't any forced dominoes or locations so it has to immediately go to *Depth 2*, where it does manage to place some forced dominoes but eventually finds an unplayable domino which requires backtracking to *Depth 1* to try some other dominoes and locations.

Where to go now

As is, this program does achieve its goal, namely to solve legal domino puzzles suitably fast and efficiently. However, this doesn't mean it is the ultimate word on the subject and, as usual, I've *intentionally* left room for improvement so that *you* can contribute extra optimizations and features if you feel so inclined, among them:

- The ***Main*** section does some validations on the input data, such as ensuring that *Rows* and *Cols* are positive integers ≤ *9*, and that the grid has exactly 56 locations available for placing the 28 dominoes. This could be extended to also check that the grid contents are *legal*: only values *0-6* are permitted and exactly 8 of each.

- Currently the program solves puzzles containing the *full* set of 28 dominoes. It could be generalized to also solve puzzles using a given *subset*, such as the following ten dominoes to be placed on a *4x5* grid, say.

- It could also be extended to accept grids with more than *9* rows/columns but this would require changing the declaration of array **`B`** from **`INTEGER`** to **`REAL`**, adjusting the representation of locations to use 2 digits for the row/column coordinates of the location instead of one, and making the appropriate changes everywhere (**`PGRID`**, for instance). Also, the **`REAL`** array **`B`** would now occupy *twice* the memory as the **`INTEGER`** one did.

- The program outputs just the first solution it finds but some puzzles may have more than one (e.g.: ***Example 3*** has *36*) so it could be extended to continue the search to find and output all additional solutions, if any.

- The **`SRTB`** *(Sort Bones)* subprogram uses a very simple sorting algorithm, as the number of dominoes to sort is initially just 28 and further it steadily decreases as the search progresses and more dominoes are forced and thus need no further sorting. Thus, a more complex sort seems unwarranted for that few elements but it might be worth a try to tentatively implement a better sort procedure and see if it actually does speed the search up.

- This program uses a straightforward solving algorithm that I concocted from scratch, which essentially tries to mimic what I would do to manually solve the puzzle: find dominoes that can be placed only on a single location, find locations where only one domino would fit, place them on the grid, rinse and repeat. Occasionally I'd get into an impossible situation where a domino couldn't be placed anywhere or a location wouldn't accept any domino and in such cases I'd have to backtrack and tentatively try other dominoes and locations. That's how I would solve it by hand and that's what my program does. Nice and easy.

  Alas, this doesn't mean that other algorithms wouldn't do better or that this one can't be improved. E.g.: when there are very few dominoes remaining it makes a final recursive call to place them but a faster procedure could be implemented at the current depth to place them all at once, which would also save *RAM*. There are other possible optimizations which would save some subprogram calls and result in a faster search.

  Finally, the solving procedure is quite memory-hungry because it creates new local copies of arrays **`B`** and **`D`** at each depth in order to backtrack almost instantly in case of reaching a dead end. However, while this saves time it's also quite costly in terms of *RAM*. Implementing some other efficient backtracking mechanism that would work on the original arrays **`B`** and **`D`** instead of making local copies would save lots of *RAM*.

- The program can be converted to run on other handheld calculators or pocket computers with reasonable ease so you might want to try adapting it to run in vintage *SHARP* or *Casio* pocket computers, and also you can convert it to *RPN*, *RPL*, *FORTH*, *Hppl*, etc. For the case of *Hppl*, it could be extended to provide a *color graphic output* of the solved puzzle similar to the ones featured in the ***Examples***. It would be fun !

- The *bones* array is conservatively declared as **`INTEGER B(28,15)`** to cater for some domino having up to 12 legal locations where to place it, but I've never seen a puzzle where a domino could be legally placed in more than **8** locations so if you're starved for *RAM* you might declare the array as **`INTEGER B(28,11)`** saving *336* bytes per *Depth* level. If you get a **`Subscript`** error for some puzzle, simply increase the **`11`** to **`12`** or **`13`**.

Copyrights

Copyright for this article and its contents is retained by the author. Permission to use it for non-profit purposes is granted as long as the contents aren't modified in any way and the copyright is acknowledged.

For the purposes of this copyright, the definition of *non-profit* does *not* include publishing this article in any media for which a subscription fee is asked and thus such use is *strictly disallowed* without explicit written permision granted by the author.