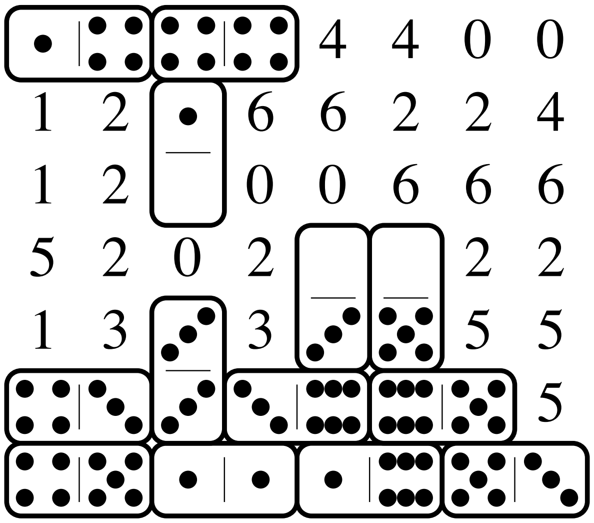
Three Ways to Solve Domino Grids

Kenneth E. Caviness

This article develops and compares three methods for solving domino grid puzzles: a “human-type” algorithm, a brute-force method, and a scheme using a generalized odometer.

A Puzzle Is Presented, and a “Human-Type” Solution Algorithm Developed

When my son brought home a paper from school with a 7×8 grid of numbers on it, I was immediately interested. The goal: cover the puzzle with all the dominoes from the “bone pile,” making sure that each number of the puzzle is covered by the same number on a domino. Many similar puzzles can be found online and in puzzle collections: see [1, 2, 3, 4, 5] for several online resources, which are the source of some of the examples considered here.



▲ **Figure 1.** A partially solved domino grid, with almost half of the 28 dominoes placed on the underlying puzzle grid.

□ Puzzle Construction

■ *Board*

Our first task is to represent the board.

```
m9 = {
   {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}};
```

■ *Bone Pile*

Next, we need the bone pile, the list of available dominoes. In this case, the bone pile consists of all 28 dominoes from the double zero to double six, but the definition is generally valid for any non-negative number n, for a total of $n(n+1)/2$ dominoes.

```
bones[n_Integer?NonNegative] :=
  bones[n] = Union[Sort /@ Tuples[Range[0, n], 2]];

bones[6]

{{0, 0}, {0, 1}, {0, 2}, {0, 3}, {0, 4}, {0, 5}, {0, 6},
 {1, 1}, {1, 2}, {1, 3}, {1, 4}, {1, 5}, {1, 6}, {2, 2},
 {2, 3}, {2, 4}, {2, 5}, {2, 6}, {3, 3}, {3, 4}, {3, 5},
 {3, 6}, {4, 4}, {4, 5}, {4, 6}, {5, 5}, {5, 6}, {6, 6}}
```

■ *Locations*

Find possible locations for a given piece `{a, b}`.

```
find[m_?MatrixQ, {a_, b_}] := Sort@Join[
   {#, # + {0, 1}} & /@ Position[Partition[#, 2, 1] & /@ m,
     Alternatives[#, Reverse[#]] &@{a, b}],
   {#, # + {1, 0}} & /@
    (Reverse /@ Position[Partition[#, 2, 1] & /@ Transpose[m],
       Alternatives[#, Reverse[#]] &@{a, b}])
  ]
```

This is the workhorse of the entire solution, first dividing the puzzle into pairs along each row and looking for matches to the given pair, then repeating the process on the transposed matrix (i.e. along the columns of the original grid) and noting the locations of any matches found. The location of the pair in the partition gives the location of the first half domino in the original grid, but adding the appropriate offset gives the location of the second half domino as well, and both are included as a domino location in the list of locations found.

■ *Displaying Dominoes*

Now for functions to highlight the dominoes within a puzzle. The function `frameDomino` generates the options to include in the `Frame` option of `Grid`.

```
frameDomino[{{x1_Integer, y1_Integer},
   {x2_Integer, y2_Integer}}] :=
 Sequence[
  {x1, y1} → Directive[Thin, GrayLevel[0.8]],
  {x2, y2} → Directive[Thin, GrayLevel[0.8]],
  {{x1, x2}, {y1, y2}} → Directive[Thick, Black]]
```

The function `displayPuzzle` accepts a puzzle grid (a matrix) and a domino list (a list of location pairs) and displays the puzzle grid with frames around the dominoes indicated in the list.

```
displayPuzzle[puzzle[m_?MatrixQ, l_List, x___], opts___] :=
  Grid[m, Alignment → {Center, Center},
   FrameStyle → Directive[Black, Thick],
   Frame → {False, False,
     Prepend[frameDomino /@ l,
      {1, #} & /@ Dimensions[m] → White]}, opts,
   ItemSize → All, Spacings → 0.5 {1, 1}];
```

For example, there are two possible locations for the domino {0, 3} in the `m9` puzzle.

```
find[m9, {0, 3}]
```

```
{{{4, 3}, {5, 3}}, {{4, 5}, {5, 5}}}
```

```
displayPuzzle[puzzle[m9, find[m9, {0, 3}]]]
```

```
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
```

A `puzzle` object `puzzle[m_?MatrixQ, filled_List, bones_List]` takes three arguments.

1. The matrix `m` contains the puzzle to be solved, a 2D array of integers.
2. The filled locations list `filled` is a list of coordinate pairs: $\{\{x_1, y_1\}, \{x_1, y_2\}\}$, where either $x_1 = x_2$ and $y_1 = y_2 \pm 1$ or $y_1 = y_2$ and $x_1 = x_2 \pm 1$.
3. The bone pile `bones`, the list of unplayed dominoes, consists of a list of pairs of integers.

The `Format` command defines how to format a puzzle: the puzzle matrix has its filled list of dominoes framed, and a tooltip shows the bone pile, if any.

```
Format[puzzle[m_?MatrixQ, filled_List, bones_: {}]] ^:=
 If[Length@bones == 0,
  displayPuzzle[puzzle[m, filled]],
  Tooltip[displayPuzzle[puzzle[m, filled]],
   Row[{"bone pile: ", ToString[bones]}]]
 ]
```

■ *Initial Puzzle*

This section shows examples of various puzzles; mouse over a puzzle to see the bone pile. In this puzzle, no dominoes have been played yet.

```
puzzle[m9, {}, bones[6]]
```

```
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
```

Here two pieces have been removed from the bone pile and placed on the board.

```
puzzle[m9, {{{2, 1}, {2, 2}}, {{3, 2}, {4, 2}}},
 Complement[bones[6], {{1, 2}, {2, 2}}]]
```

```
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
```

□ Solution Tools

■ *Hide Board Positions Already Filled*

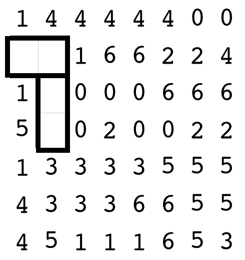
To ensure that the squares filled by already placed pieces are no longer included, make a version of the board with the affected squares blanked out.

```
hideFilled[m_?MatrixQ, l : {{{_, _}, {_, _}} ...}] :=
 ReplacePart[m, Flatten[l, 1] → ""]
```

```
hideFilled[m9, {{{2, 1}, {2, 2}}, {{3, 2}, {4, 2}}}]
```

```
{{1, 4, 4, 4, 4, 4, 0, 0},
 {, , 1, 6, 6, 2, 2, 4}, {1, , 0, 0, 0, 6, 6, 6},
 {5, , 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}}
```

```
puzzle[
 hideFilled[m9, {
    {{2, 1}, {2, 2}},
    {{3, 2}, {4, 2}}
   }],
 {{{2, 1}, {2, 2}},
  {{3, 2}, {4, 2}}}]
```



```
1 4 4 4 4 4 0 0
    1 6 6 2 2 4
1   0 0 0 6 6 6
5   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
```

■ *Forced Locations*

This function finds the forced locations; only one piece can possibly go into a forced location.

```
forcedLocations[m_?MatrixQ] :=
 Module[{neighbor, shadow, map},
  neighbor =
   ArrayPad[
    Sum[RotateRight[ArrayPad[m /. {_Integer → 1, "" → 0}, 1],
      r], {r, {{-1, 0}, {0, -1}, {0, 1}, {1, 0}}}], -1];
  shadow = m /. {_Integer → 0, "" → ∞};
  map = neighbor + shadow;
  If[! FreeQ[map, 0], Return[False]]; (* Error,
  some location is unplayable *)
  find[map, {1, _Integer}]
 ]
```

Find the forced locations after two particular dominoes have been played.

```
forcedLocations[hideFilled[m9, {
    {{2, 1}, {2, 2}},
    {{3, 2}, {4, 2}}
   }]]
```

```
{{{1, 1}, {1, 2}}, {{3, 1}, {4, 1}}}
```

The forced locations are shown empty.

```
puzzle[hideFilled[m9, {{{2, 1}, {2, 2}}, {{3, 2}, {4, 2}}}],
 {{{2, 1}, {2, 2}}, {{3, 2}, {4, 2}}} ⋃ %]
```

```
1 4 4 4 4 4 0 0
    1 6 6 2 2 4
1   0 0 0 6 6 6
5   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
```

■ *Solution Method*

1. Select the pieces that fit in forced locations.
   - If there are any squares where no pieces can be placed, the puzzle is unsolvable, so quit.
2. Use `find` to return a list of all possible locations for playable pieces, and select the pieces that have only one possible location:
   - If any piece has no possible placement, the puzzle is unsolvable, so quit.
   - If all the pieces have multiple possible locations, take the first piece with the minimum number of locations, and do all options on duplicate boards.

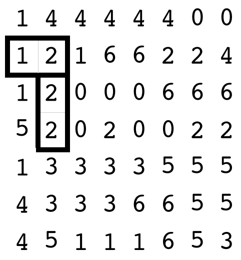
```
debug = True;

step[l_List] := Flatten[step /@ Flatten[{l}]]

step[puzzle[m0_?MatrixQ, f0_List, b0_List]] :=
 Module[{m, fl, found, freq, minfreq, ours, filled = f0,
    bones = b0, fp},
   If[Length[bones] == 0, Return[puzzle[m0, f0, b0]]];
   (* already done! *)
   m = hideFilled[m0, filled];
   If[(fl = forcedLocations[m]) === False,
    If[debug, Print["Impossible location"]]; Return[{}]];

   fp = Sort /@ Apply[m[[##]] &, fl, {2}];
   If[debug, Print["forcedLocations: ", fl, "; pieces: ",
     fp]];

   If[Length[fl] > 0,
    If[Length[Union @@ fl] ≠ 2 Length[fl],
     If[debug, Print["Overlapping forced locations"]];
     Return[{}]];
    If[Length[Complement[fp, bones]] > 0,
     If[debug,
      Print[
       "Forced location requires piece already played"];
      Return[{}]]];
    If[Length[Union[fp]] ≠ Length[fp],
     If[debug, Print["Forced locations require same piece"];
      Return[{}]]];
    Return[puzzle[m0, Join[filled, fl],
      Complement[bones, Sort /@ Apply[m⟦##⟧ &, fl, {2}]]]]
   ];
   (* look at options for the pieces still in the
    bone pile *)
   found = find[m, #] & /@ bones;
   (* find all possible locations for all remaining
    pieces *)
   freq = Length /@ found;
   minfreq = Min[freq];
   ours = Position[freq, _? (# == minfreq &)];
   If[debug, Print["freq: ", freq, "\nminfreq: ",
     minfreq, "; ours: ", ours]];

   Switch[minfreq,
    0, If[debug, Print["unplayable piece(s): ",
      bones⟦#⟧ & @@@ ours]]; Return[{}],
    (* some piece unplayable, no solution! *)

    1, ours = Flatten[Position[freq, 1]];
    (* list of indices of all pieces that have only
     1 possible location *)
    filled = Join[filled, Flatten[found⟦ours⟧, 1]];
    (* add the locations of these pieces to the answer *)
    bones = Delete[bones, List /@ ours];
    (* delete these pieces from the bone pile *)
    Return[puzzle[m0, filled, bones]],

    _, ours = First@First@Position[freq, Min[freq]];
    (* index of first piece that has minumum number
     of locations *)
    If[debug, Print["reduced ours list: ", ours]];
    bones = Delete[bones, ours];
    (* delete this piece from the bone pile *)
    If[debug, Print["found⟦ours⟧: ", found⟦ours⟧]];
    puzzle[m0, #, bones] & /@
     (Append[filled, #] & /@ found⟦ours⟧)
   ](* for each location of this piece,
   add to (a copy of) the answer *)
  ]
```

```
solvePuzzle[l_] := Flatten[{FixedPoint[step, l]}] /.
  puzzle[m_, s_, r___] :→ puzzle[m, Sort[s], r]
(* for consistency,
return a list of solutions with the set of steps
 sorted for easy comparison *)
```

■ *Examples*

In this artificial case, there are two forced locations: in each, only one piece can be placed.

```
puzzle[m9, {{{2, 1}, {2, 2}}, {{3, 2}, {4, 2}}},
 Complement[bones[6], {{1, 2}, {2, 2}}]]
```

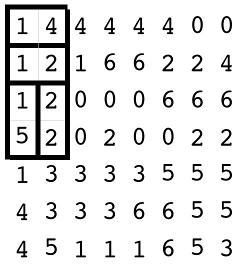


```
1 4 4 4 4 4 0 0
1 2 1 6 6 2 2 4
1 2 0 0 0 6 6 6
5 2 0 2 0 0 2 2
1 3 3 3 3 5 5 5
4 3 3 3 6 6 5 5
4 5 1 1 1 6 5 3
```

The function `step` finds the forced locations and fills them in with the appropriate dominoes taken from the bone pile. Mouse over to see that these dominoes have been removed from the bone pile.

```
step[%]
```

```
forcedLocations: {{{1, 1}, {1, 2}}, {{3, 1}, {4, 1}}}
 ; pieces: {{1, 4}, {1, 5}}
```



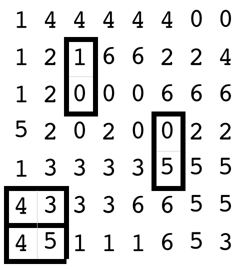
```
1 4 4 4 4 4 0 0
1 2 1 6 6 2 2 4
1 2 0 0 0 6 6 6
5 2 0 2 0 0 2 2
1 3 3 3 3 5 5 5
4 3 3 3 6 6 5 5
4 5 1 1 1 6 5 3
```

At the beginning, there are no forced locations, but there are four forced pieces: pieces that can only be placed in one location in the puzzle: {0, 1}, {0, 5}, {3, 4}, and {4, 5}. The `step` function plays all four at once.

```
step@puzzle[m9, {}, bones[6]]
```

```
forcedLocations: {}; pieces: {}
```

```
freq: {6, 1, 7, 2, 2, 1, 4, 4, 3, 3, 3,
  3, 3, 4, 2, 3, 3, 5, 8, 1, 4, 2, 5, 1, 3, 6, 3, 5}
minfreq: 1; ours: {{2}, {6}, {20}, {24}}
```



```
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
```

■ *Solve It!*

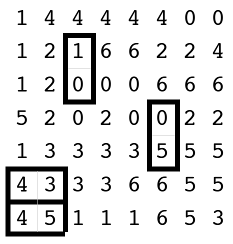
We are ready to solve the whole puzzle. The next command prints the current state, takes one step, and repeats until the bone pile is empty.

```
NestWhile[(Print[#]; step[#]) &, puzzle[m9, {}, bones[6]],
 Last[Last[#]] ≠ {} &]
```

```
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
```

```
forcedLocations: {}; pieces: {}
```

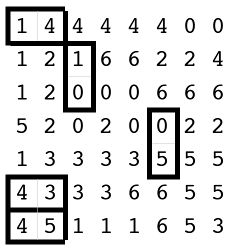
```
freq: {6, 1, 7, 2, 2, 1, 4, 4, 3, 3, 3,
  3, 3, 4, 2, 3, 3, 5, 8, 1, 4, 2, 5, 1, 3, 6, 3, 5}
minfreq: 1; ours: {{2}, {6}, {20}, {24}}
```



```
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
```

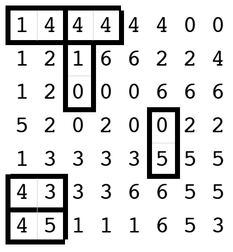
```
forcedLocations: {}; pieces: {}
```

```
freq: {3, 5, 2, 2, 3, 4, 2, 3,
  1, 2, 2, 4, 2, 3, 3, 5, 6, 2, 2, 4, 3, 5, 2, 5}
minfreq: 1; ours: {{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
```

```
forcedLocations: {{{1, 3}, {1, 4}}}; pieces: {{4, 4}}
```



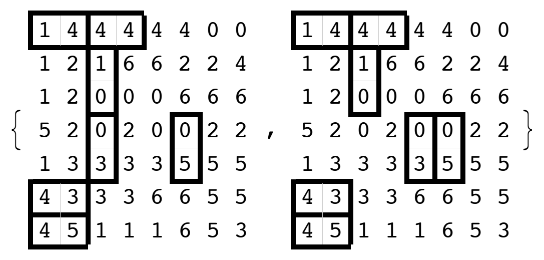
```
1 4 4 4 4 4 0 0
1 2 1 6 6 2 2 4
1 2 0 0 0 6 6 6
5 2 0 2 0 0 2 2
1 3 3 3 3 5 5 5
4 3 3 3 6 6 5 5
4 5 1 1 1 6 5 3
```

```
forcedLocations: {}; pieces: {}
```

```
freq: {3, 5, 2, 2, 3, 3, 2, 3, 2, 2, 4, 2, 2, 3, 5, 6, 2, 2, 2, 5, 2, 5}
minfreq: 2; ours:
 {{3}, {4}, {7}, {9}, {10}, {12}, {13}, {17}, {18}, {19}, {21}}
```

```
reduced ours list: 3
```

```
found⟦ours⟧: {{{4, 3}, {5, 3}}, {{4, 5}, {5, 5}}}
```



```
  1 4 4 4 4 4 0 0     1 4 4 4 4 4 0 0
  1 2 1 6 6 2 2 4     1 2 1 6 6 2 2 4
  1 2 0 0 0 6 6 6     1 2 0 0 0 6 6 6
{ 5 2 0 2 0 0 2 2  ,  5 2 0 2 0 0 2 2 }
  1 3 3 3 3 5 5 5     1 3 3 3 3 5 5 5
  4 3 3 3 6 6 5 5     4 3 3 3 6 6 5 5
  4 5 1 1 1 6 5 3     4 5 1 1 1 6 5 3
```

```
forcedLocations: {}; pieces: {}
```

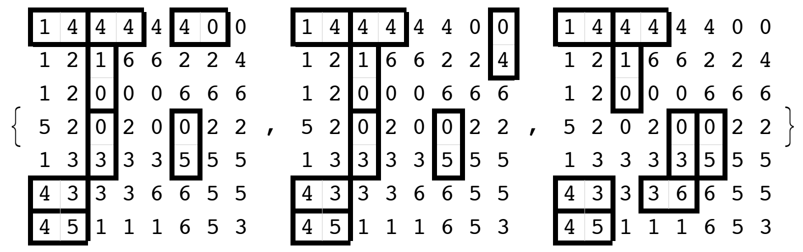
```
freq: {3, 3, 2, 3, 3, 2, 3, 2, 2, 4, 2, 2, 3, 5, 3, 2, 2, 2, 5, 2, 5}
minfreq: 2; ours:
 {{3}, {6}, {8}, {9}, {11}, {12}, {16}, {17}, {18}, {20}}
```

```
reduced ours list: 3
```

```
found⟦ours⟧: {{{1, 6}, {1, 7}}, {{1, 8}, {2, 8}}}
```

```
forcedLocations: {}; pieces: {}
```

```
freq: {2, 4, 2, 3, 3, 2, 3, 2, 2, 4, 2, 2, 3, 5, 5, 2, 1, 2, 5, 2, 5}
minfreq: 1; ours: {{17}}
```



```
forcedLocations: {{{1, 5}, {2, 5}}, {{1, 8}, {2, 8}}}
 ; pieces: {{4, 6}, {0, 4}}
```

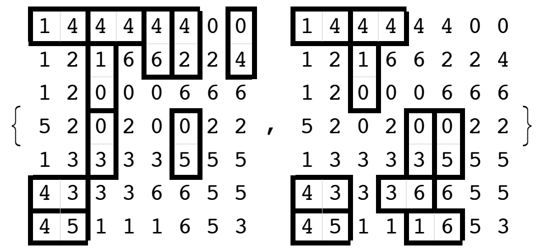
```
Forced location requires piece already played
```

```
forcedLocations: {}; pieces: {}
```

```
freq: {2, 3, 3, 3, 2, 3, 2, 2, 4, 2, 1, 3, 5, 3, 2, 2, 1, 5, 2, 5}
minfreq: 1; ours: {{11}, {17}}
```

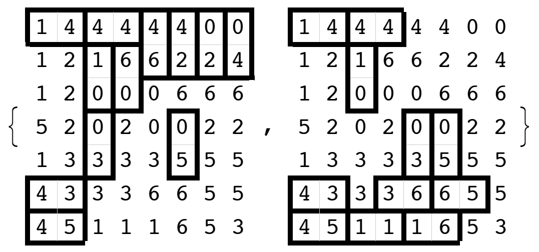
```
forcedLocations: {}; pieces: {}
```

```
freq: {2, 4, 2, 3, 3, 2, 2, 2, 1, 4, 2, 2, 3, 5, 3, 2, 2, 5, 2, 4}
minfreq: 1; ours: {{9}}
```



```
forcedLocations: {{{1, 7}, {2, 7}}, {{2, 4}, {3, 4}}}
 ; pieces: {{0, 2}, {0, 6}}
```

```
forcedLocations: {{{6, 6}, {6, 7}}, {{7, 3}, {7, 4}}}
 ; pieces: {{5, 6}, {1, 1}}
```

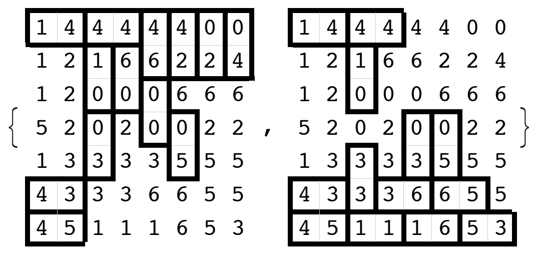


```
  1 4 4 4 4 4 0 0     1 4 4 4 4 4 0 0
  1 2 1 6 6 2 2 4     1 2 1 6 6 2 2 4
  1 2 0 0 0 6 6 6     1 2 0 0 0 6 6 6
{ 5 2 0 2 0 0 2 2  ,  5 2 0 2 0 0 2 2 }
  1 3 3 3 3 5 5 5     1 3 3 3 3 5 5 5
  4 3 3 3 6 6 5 5     4 3 3 3 6 6 5 5
  4 5 1 1 1 6 5 3     4 5 1 1 1 6 5 3
```

```
forcedLocations: {}; pieces: {}
```

```
freq: {1, 3, 2, 3, 2, 2, 3, 2, 3, 2, 3, 2, 2, 5, 2, 4}
minfreq: 1; ours: {{1}}
```

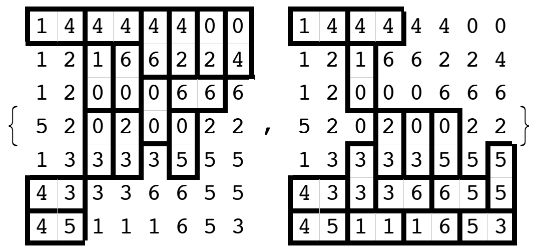
```
forcedLocations: {{{5, 3}, {6, 3}}, {{7, 7}, {7, 8}}}
 ; pieces: {{3, 3}, {3, 5}}
```



```
  1 4 4 4 4 4 0 0     1 4 4 4 4 4 0 0
  1 2 1 6 6 2 2 4     1 2 1 6 6 2 2 4
  1 2 0 0 0 6 6 6     1 2 0 0 0 6 6 6
{ 5 2 0 2 0 0 2 2  ,  5 2 0 2 0 0 2 2 }
  1 3 3 3 3 5 5 5     1 3 3 3 3 5 5 5
  4 3 3 3 6 6 5 5     4 3 3 3 6 6 5 5
  4 5 1 1 1 6 5 3     4 5 1 1 1 6 5 3
```

```
forcedLocations: {{{3, 6}, {3, 7}}, {{4, 4}, {5, 4}}}
 ; pieces: {{6, 6}, {2, 3}}
```

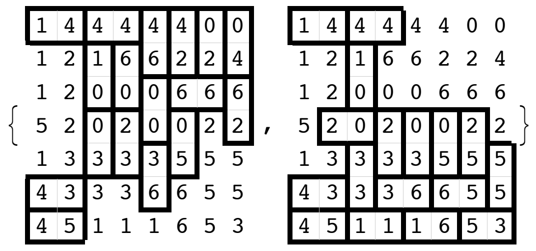
```
forcedLocations: {{{4, 4}, {5, 4}}, {{5, 8}, {6, 8}}}
 ; pieces: {{2, 3}, {5, 5}}
```



```
  1 4 4 4 4 4 0 0     1 4 4 4 4 4 0 0
  1 2 1 6 6 2 2 4     1 2 1 6 6 2 2 4
  1 2 0 0 0 6 6 6     1 2 0 0 0 6 6 6
{ 5 2 0 2 0 0 2 2  ,  5 2 0 2 0 0 2 2 }
  1 3 3 3 3 5 5 5     1 3 3 3 3 5 5 5
  4 3 3 3 6 6 5 5     4 3 3 3 6 6 5 5
  4 5 1 1 1 6 5 3     4 5 1 1 1 6 5 3
```

```
forcedLocations: {{{3, 8}, {4, 8}}, {{5, 5}, {6, 5}}}
 ; pieces: {{2, 6}, {3, 6}}
```

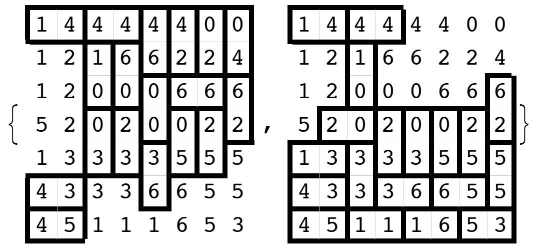
```
forcedLocations: {{{4, 2}, {4, 3}}, {{4, 7}, {5, 7}}}
 ; pieces: {{0, 2}, {2, 5}}
```



```
  1 4 4 4 4 4 0 0     1 4 4 4 4 4 0 0
  1 2 1 6 6 2 2 4     1 2 1 6 6 2 2 4
  1 2 0 0 0 6 6 6     1 2 0 0 0 6 6 6
{ 5 2 0 2 0 0 2 2 ,   5 2 0 2 0 0 2 2 }
  1 3 3 3 3 5 5 5     1 3 3 3 3 5 5 5
  4 3 3 3 6 6 5 5     4 3 3 3 6 6 5 5
  4 5 1 1 1 6 5 3     4 5 1 1 1 6 5 3
```

```
forcedLocations: {{{4, 7}, {5, 7}}}; pieces: {{2, 5}}
```

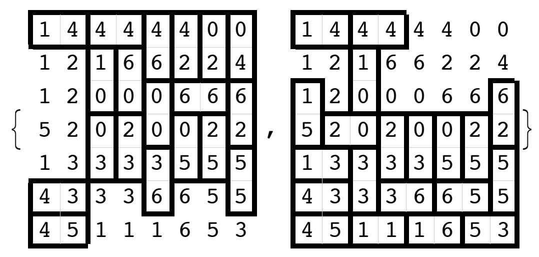
```
forcedLocations: {{{3, 8}, {4, 8}}, {{5, 1}, {5, 2}}}
 ; pieces: {{2, 6}, {1, 3}}
```



```
  1 4 4 4 4 4 0 0     1 4 4 4 4 4 0 0
  1 2 1 6 6 2 2 4     1 2 1 6 6 2 2 4
  1 2 0 0 0 6 6 6     1 2 0 0 0 6 6 6
{ 5 2 0 2 0 0 2 2 ,   5 2 0 2 0 0 2 2 }
  1 3 3 3 3 5 5 5     1 3 3 3 3 5 5 5
  4 3 3 3 6 6 5 5     4 3 3 3 6 6 5 5
  4 5 1 1 1 6 5 3     4 5 1 1 1 6 5 3
```

```
forcedLocations: {{{5, 8}, {6, 8}}}; pieces: {{5, 5}}
```

```
forcedLocations: {{{3, 1}, {4, 1}}}; pieces: {{1, 5}}
```

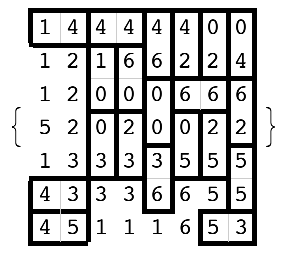


```
  1 4 4 4 4 4 0 0     1 4 4 4 4 4 0 0
  1 2 1 6 6 2 2 4     1 2 1 6 6 2 2 4
  1 2 0 0 0 6 6 6     1 2 0 0 0 6 6 6
{ 5 2 0 2 0 0 2 2  ,  5 2 0 2 0 0 2 2 }
  1 3 3 3 3 5 5 5     1 3 3 3 3 5 5 5
  4 3 3 3 6 6 5 5     4 3 3 3 6 6 5 5
  4 5 1 1 1 6 5 3     4 5 1 1 1 6 5 3
```

```
forcedLocations: {{{7, 7}, {7, 8}}}; pieces: {{3, 5}}
```

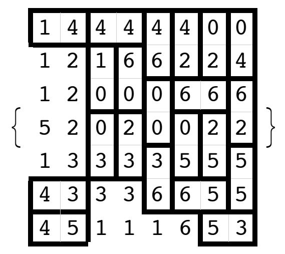
```
forcedLocations: {{{2, 1}, {2, 2}}, {{2, 2}, {3, 2}}}
 ; pieces: {{1, 2}, {2, 2}}
```

```
Overlapping forced locations
```



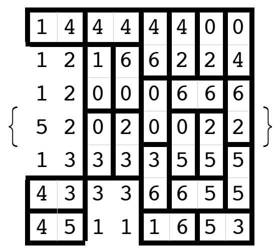
```
  1 4 4 4 4 4 0 0
  1 2 1 6 6 2 2 4
  1 2 0 0 0 6 6 6
{ 5 2 0 2 0 0 2 2 }
  1 3 3 3 3 5 5 5
  4 3 3 3 6 6 5 5
  4 5 1 1 1 6 5 3
```

```
forcedLocations: {{{6, 6}, {6, 7}}}; pieces: {{5, 6}}
```



```
  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
```

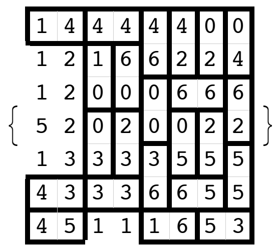
```
forcedLocations: {{{7, 5}, {7, 6}}}; pieces: {{1, 6}}
```



```
  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
```

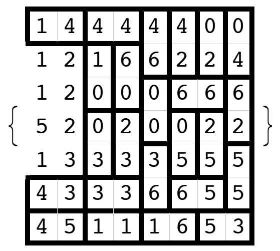
```
forcedLocations: {}; pieces: {}
```

```
freq: {2, 2, 3, 2, 2, 1}
minfreq: 1; ours: {{6}}
```



```
  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
```

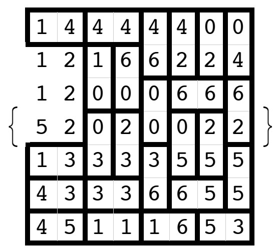
```
forcedLocations: {{{7, 3}, {7, 4}}}; pieces: {{1, 1}}
```



```
  1 4 4 4 4 4 0 0
  1 2 1 6 6 2 2 4
  1 2 0 0 0 6 6 6
{ 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
```

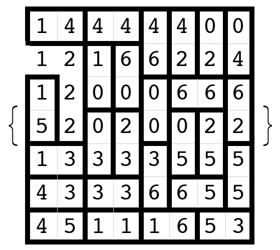
```
forcedLocations: {}; pieces: {}
```

```
freq: {2, 1, 2, 2}
minfreq: 1; ours: {{2}}
```

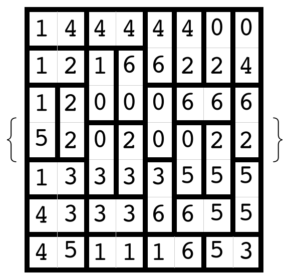


```
forcedLocations: {}; pieces: {}
```

```
freq: {2, 1, 2}
minfreq: 1; ours: {{2}}
```



```
forcedLocations: {{{2, 1}, {2, 2}}, {{3, 2}, {4, 2}}}
 ; pieces: {{1, 2}, {2, 2}}
```



Along the way, multiple partial solutions had to be considered when no forced locations or forced pieces were found, but in the end all but one solution were dropped because of inconsistency. The comments were left in to show the forced locations or forced pieces at each step, but now we turn them off.

```
debug = False;
```

■ *Alternative Display Function*

There is no reason not to make a prettier display function to show the dominoes with their customary pips (or dots), rather than showing only the grid numbers. We can represent the pip positions by matrices, some of which can be easily created by built-in matrix commands. Since the pip positions of double-9 and double-6 domino sets are consistent, let us build the larger set here. (A double-12 set would require adjusting the pip positions.)

```
dieMatrix[9] = Table[1, {3}, {3}];
dieMatrix[2] = DiagonalMatrix[{1, 0, 1}];
dieMatrix[3] = IdentityMatrix[3];
dieMatrix[6] = Table[If[j == 2, 0, 1], {j, 3}, {3}];
MatrixForm /@ {dieMatrix[9], dieMatrix[2], dieMatrix[3],
  dieMatrix[6]}
```

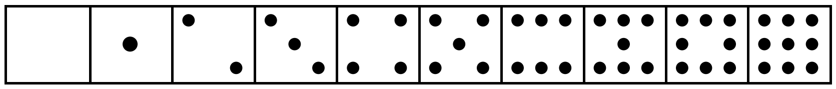
$$\left\{\begin{pmatrix}1&1&1\\1&1&1\\1&1&1\end{pmatrix},\begin{pmatrix}1&0&0\\0&0&0\\0&0&1\end{pmatrix},\begin{pmatrix}1&0&0\\0&1&0\\0&0&1\end{pmatrix},\begin{pmatrix}1&1&1\\0&0&0\\1&1&1\end{pmatrix}\right\}$$

The other matrices could be built by hand or using `SparseArray` or `Table` with appropriate criteria, but it is easy to create them by addition and subtraction.

```
dieMatrix[0] = dieMatrix[2] - dieMatrix[2];
dieMatrix[1] = dieMatrix[3] - dieMatrix[2];
dieMatrix[4] = dieMatrix[2] + Reverse[dieMatrix[2]];
dieMatrix[5] = dieMatrix[4] + dieMatrix[1];
dieMatrix[7] = dieMatrix[6] + dieMatrix[1];
dieMatrix[8] = dieMatrix[9] - dieMatrix[1];
```

A pip will be placed on a half-domino square wherever the matrix had a 1.

```
Do[
  diePips[n] =
   Point[Position[Reverse@Transpose@dieMatrix[n], 1] / 4.0],
  {n, 0, 9}];
Grid[
 {Graphics[{PointSize[.25], diePips[#]},
    ImageSize → 25 {1, 1}] & /@ Range[0, 9]}, Dividers → All]
```

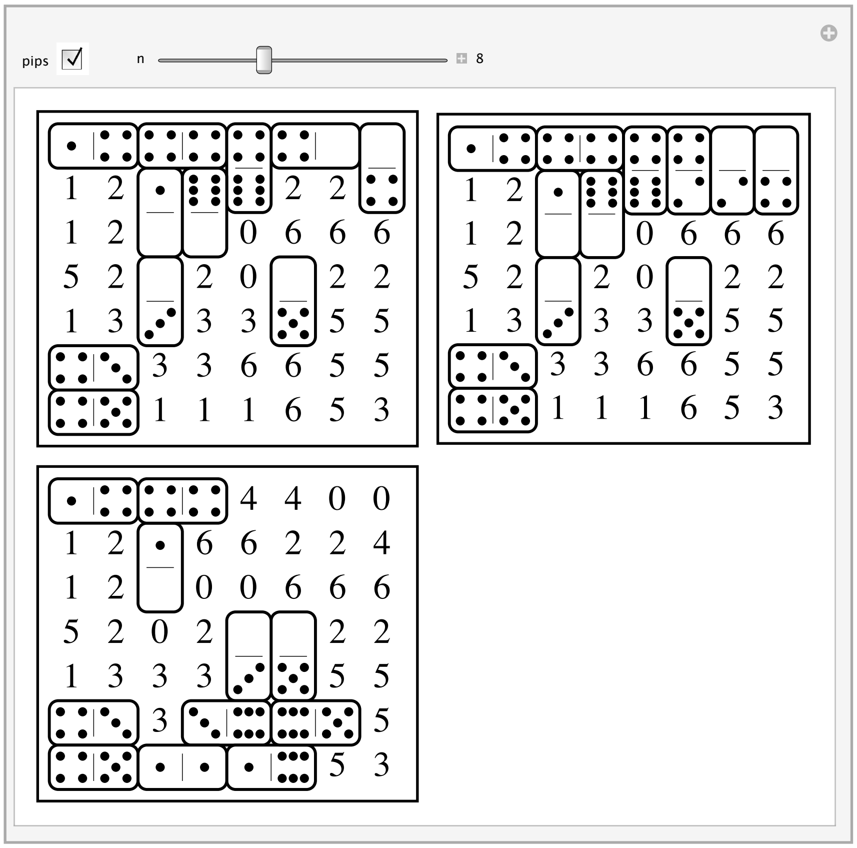


The function `displayDottedPuzzle` creates a graphical display of the puzzle, optionally replacing numbers by half-domino faces for any locations listed in the “filled list,” outlining any placed dominoes in a way similar to `displayPuzzle`.

```
rot270[{x_, y_}] := {y, -x}

outlinePair[p : {{x1_, y1_}, {x2_, y2_}}] :=
 Module[{midpt = N@Mean[p] + 0.5, ofst},
  ofst = 0.3 If[x1 == x2, {1, 0}, {0, 1}];
  {EdgeForm[{Black, Thick}],
   FaceForm[White, Opacity[0.5]],
   Rectangle[{Min[x1, x2], Min[y1, y2]},
    {Max[x1, x2], Max[y1, y2]} + 1, RoundingRadius → .2],
   Line[{midpt - ofst, midpt + ofst}]}
 ]

Options[displayDottedPuzzle] = {ShowPips → True};

SyntaxInformation[displayDottedPuzzle] =
  {"ArgumentsPattern" → {_, OptionsPattern[]}};

displayDottedPuzzle[puzzle[m_?MatrixQ, p_List, x___],
  OptionsPattern[]] :=
 Module[{ϵ = 0.1, sd, pv, ph},
  sd = If[OptionValue[ShowPips] === True, True, False, True];
  Graphics[Flatten[{
     PointSize → .025,
     outlinePair /@ Map[rot270, p, {2}],
     Black,
     MapIndexed[
      Text[Style[#1, Large], rot270[#2] + .5 {1, 1}] &,
      If[sd, ReplacePart[# , Flatten[p, 1] -> ""], #] &@m,
      {2}],
     If[sd,
      pv = Select[p, Subtract @@ # == {-1, 0} &];
      ph = Complement[p, pv];
      {Translate[diePips[m〚Sequence @@ #〛], rot270 [#]] & /@
         Flatten[ph, 1],
       Translate[Rotate[diePips[m〚Sequence @@ #〛], π / 2],
          rot270 [#]] & /@ Flatten[pv, 1]},
      {}]
    }],
   PlotRange →
    ({1 + {-ϵ, #〚2〛 + ϵ}, {-#〚1〛 - ϵ, ϵ}} &@ Dimensions[m]),
   ImageSize → 248]
 ]
```

■ *Animate the Solution*

```
With[
 {soln =
   Most@FixedPointList[step, {puzzle[m9, {}, bones[6]]}]},
 Manipulate[
  Pane[
   Grid[
    Partition[
     Framed[displayDottedPuzzle[#, ShowPips → pips]] & /@
      soln〚n〛,
     2, 2, 1, {}],
    Spacings → {1, 1}
   ],
   ImageSize → {530, 475}],
  Grid[{{Control[{pips, {True, False}}], Spacer[20],
     Control[{{n, 8}, 1, Length[soln], 1,
       Appearance → "Labeled"}] }}],
  SaveDefinitions → True
 ]
]
```



□ Thoughts

The method described here can be thought of as “human-type,” since it uses intelligently chosen criteria for deciding which step to perform and which option to try next. The criteria used can be summarized as follows:

1. Seek forced locations: if any locations can take none of the available dominoes, abandon the partial solution currently being constructed; if any locations can take exactly one available domino (and not the same one), fill all of these “forced locations.”
2. Else seek forced dominoes: if any of the available dominoes cannot be placed on the board, abandon the partial solution currently being constructed; if any of the available dominoes can only be placed in one location on the board, play all of these "forced dominoes."
3. Else for a minimal case, place one domino in all possible locations, making separate copies of the puzzle for each case.
4. Repeat until no further changes occur.

A human can make more complicated arguments eliminating some options; for examples, see the explanations at the sites [1, 2, 3, 4, 5]. (But not all suggested solving strategies turn out to be useful. One common idea, placing the “double” dominoes first, can easily be defeated by a clever puzzle designer.) The order is arbitrary and might be modified, but is far faster than the more simplistic, brute-force method presented in the following section.

■ Brute-Force Collision Method

Here is a list of all possible locations of all dominoes in our original puzzle.

```
(options = find[m9, #] & /@ bones[6]) // Column

{{{1, 7}, {1, 8}}, {{3, 3}, {3, 4}}, {{3, 3}, {4, 3}},
 {{3, 4}, {3, 5}}, {{3, 5}, {4, 5}}, {{4, 5}, {4, 6}}}
{{{2, 3}, {3, 3}}}
{{{1, 7}, {2, 7}}, {{3, 2}, {3, 3}},
 {{3, 4}, {4, 4}}, {{4, 2}, {4, 3}},
 {{4, 3}, {4, 4}}, {{4, 4}, {4, 5}}, {{4, 6}, {4, 7}}}
{{{4, 3}, {5, 3}}, {{4, 5}, {5, 5}}}
{{{1, 6}, {1, 7}}, {{1, 8}, {2, 8}}}
{{{4, 6}, {5, 6}}}
{{{2, 4}, {3, 4}}, {{2, 5}, {3, 5}},
 {{3, 5}, {3, 6}}, {{3, 6}, {4, 6}}}
{{{1, 1}, {2, 1}}, {{2, 1}, {3, 1}},
 {{7, 3}, {7, 4}}, {{7, 4}, {7, 5}}}
{{{2, 1}, {2, 2}}, {{2, 2}, {2, 3}}, {{3, 1}, {3, 2}}}
{{{5, 1}, {5, 2}}, {{6, 3}, {7, 3}}, {{6, 4}, {7, 4}}}
```

```
{{{1, 1}, {1, 2}}, {{1, 3}, {2, 3}}, {{5, 1}, {6, 1}}}
{{{3, 1}, {4, 1}}, {{4, 1}, {5, 1}}, {{7, 2}, {7, 3}}}
{{{2, 3}, {2, 4}}, {{6, 5}, {7, 5}}, {{7, 5}, {7, 6}}}
{{{2, 2}, {3, 2}}, {{2, 6}, {2, 7}},
 {{3, 2}, {4, 2}}, {{4, 7}, {4, 8}}}
{{{4, 2}, {5, 2}}, {{4, 4}, {5, 4}}}
{{{1, 2}, {2, 2}}, {{1, 6}, {2, 6}}, {{2, 7}, {2, 8}}}
{{{4, 1}, {4, 2}}, {{4, 7}, {5, 7}}, {{4, 8}, {5, 8}}}
{{{2, 5}, {2, 6}}, {{2, 6}, {3, 6}},
 {{2, 7}, {3, 7}}, {{3, 7}, {4, 7}}, {{3, 8}, {4, 8}}}
{{{5, 2}, {5, 3}}, {{5, 2}, {6, 2}},
 {{5, 3}, {5, 4}}, {{5, 3}, {6, 3}}, {{5, 4}, {5, 5}},
 {{5, 4}, {6, 4}}, {{6, 2}, {6, 3}}, {{6, 3}, {6, 4}}}
{{{6, 1}, {6, 2}}}
{{{5, 5}, {5, 6}}, {{6, 2}, {7, 2}},
 {{6, 8}, {7, 8}}, {{7, 7}, {7, 8}}}
{{{5, 5}, {6, 5}}, {{6, 4}, {6, 5}}}
{{{1, 2}, {1, 3}}, {{1, 3}, {1, 4}},
 {{1, 4}, {1, 5}}, {{1, 5}, {1, 6}}, {{6, 1}, {7, 1}}}
{{{7, 1}, {7, 2}}}
{{{1, 4}, {2, 4}}, {{1, 5}, {2, 5}}, {{2, 8}, {3, 8}}}
{{{5, 6}, {5, 7}}, {{5, 7}, {5, 8}}, {{5, 7}, {6, 7}},
 {{5, 8}, {6, 8}}, {{6, 7}, {6, 8}}, {{6, 7}, {7, 7}}}
{{{5, 6}, {6, 6}}, {{6, 6}, {6, 7}}, {{7, 6}, {7, 7}}}
{{{2, 4}, {2, 5}}, {{3, 6}, {3, 7}},
 {{3, 7}, {3, 8}}, {{6, 5}, {6, 6}}, {{6, 6}, {7, 6}}}
```

The number of options for the pieces varies wildly.

```
Length /@ options
```

```
{6, 1, 7, 2, 2, 1, 4, 4, 3, 3, 3, 3, 3,
 4, 2, 3, 3, 5, 8, 1, 4, 2, 5, 1, 3, 6, 3, 5}
```

(You can easily verify that in this puzzle, all the double dominoes have between four and eight possible placement options, making “place doubles first” a poor strategy in this case.) Taking all possible options for all the pieces gives a very large number.

```
Times @@ %
```

```
20 316 635 136 000
```

Too many cases to consider! But this method would work, theoretically: Use `Outer` to get all combinations of choices of these options and then use `Select` on those that have no overlapping dominoes. Here do only the first three dominoes.

```
Flatten[Outer[List, Sequence @@ options[[ ;; 3]], 1],
 2]
```

```
{{{{1, 7}, {1, 8}}, {{2, 3}, {3, 3}}, {{1, 7}, {2, 7}}},
 {{{1, 7}, {1, 8}}, {{2, 3}, {3, 3}}, {{3, 2}, {3, 3}}},
 {{{1, 7}, {1, 8}}, {{2, 3}, {3, 3}}, {{3, 4}, {4, 4}}},
 {{{1, 7}, {1, 8}}, {{2, 3}, {3, 3}}, {{4, 2}, {4, 3}}},
 {{{1, 7}, {1, 8}}, {{2, 3}, {3, 3}}, {{4, 3}, {4, 4}}},
 {{{1, 7}, {1, 8}}, {{2, 3}, {3, 3}}, {{4, 4}, {4, 5}}},
 {{{1, 7}, {1, 8}}, {{2, 3}, {3, 3}}, {{4, 6}, {4, 7}}},
 {{{3, 3}, {3, 4}}, {{2, 3}, {3, 3}}, {{1, 7}, {2, 7}}},
 {{{3, 3}, {3, 4}}, {{2, 3}, {3, 3}}, {{3, 2}, {3, 3}}},
 {{{3, 3}, {3, 4}}, {{2, 3}, {3, 3}}, {{3, 4}, {4, 4}}},
 {{{3, 3}, {3, 4}}, {{2, 3}, {3, 3}}, {{4, 2}, {4, 3}}},
 {{{3, 3}, {3, 4}}, {{2, 3}, {3, 3}}, {{4, 3}, {4, 4}}},
 {{{3, 3}, {3, 4}}, {{2, 3}, {3, 3}}, {{4, 4}, {4, 5}}},
 {{{3, 3}, {3, 4}}, {{2, 3}, {3, 3}}, {{4, 6}, {4, 7}}},
 {{{3, 3}, {4, 3}}, {{2, 3}, {3, 3}}, {{1, 7}, {2, 7}}},
 {{{3, 3}, {4, 3}}, {{2, 3}, {3, 3}}, {{3, 2}, {3, 3}}},
 {{{3, 3}, {4, 3}}, {{2, 3}, {3, 3}}, {{3, 4}, {4, 4}}},
 {{{3, 3}, {4, 3}}, {{2, 3}, {3, 3}}, {{4, 2}, {4, 3}}},
 {{{3, 3}, {4, 3}}, {{2, 3}, {3, 3}}, {{4, 3}, {4, 4}}},
 {{{3, 3}, {4, 3}}, {{2, 3}, {3, 3}}, {{4, 4}, {4, 5}}},
 {{{3, 3}, {4, 3}}, {{2, 3}, {3, 3}}, {{4, 6}, {4, 7}}},
 {{{3, 4}, {3, 5}}, {{2, 3}, {3, 3}}, {{1, 7}, {2, 7}}},
 {{{3, 4}, {3, 5}}, {{2, 3}, {3, 3}}, {{3, 2}, {3, 3}}},
 {{{3, 4}, {3, 5}}, {{2, 3}, {3, 3}}, {{3, 4}, {4, 4}}},
 {{{3, 4}, {3, 5}}, {{2, 3}, {3, 3}}, {{4, 2}, {4, 3}}},
 {{{3, 4}, {3, 5}}, {{2, 3}, {3, 3}}, {{4, 3}, {4, 4}}},
 {{{3, 4}, {3, 5}}, {{2, 3}, {3, 3}}, {{4, 4}, {4, 5}}},
 {{{3, 4}, {3, 5}}, {{2, 3}, {3, 3}}, {{4, 6}, {4, 7}}},
 {{{3, 5}, {4, 5}}, {{2, 3}, {3, 3}}, {{1, 7}, {2, 7}}},
 {{{3, 5}, {4, 5}}, {{2, 3}, {3, 3}}, {{3, 2}, {3, 3}}},
 {{{3, 5}, {4, 5}}, {{2, 3}, {3, 3}}, {{3, 4}, {4, 4}}},
 {{{3, 5}, {4, 5}}, {{2, 3}, {3, 3}}, {{4, 2}, {4, 3}}},
 {{{3, 5}, {4, 5}}, {{2, 3}, {3, 3}}, {{4, 3}, {4, 4}}},
 {{{3, 5}, {4, 5}}, {{2, 3}, {3, 3}}, {{4, 4}, {4, 5}}},
 {{{3, 5}, {4, 5}}, {{2, 3}, {3, 3}}, {{4, 6}, {4, 7}}},
 {{{4, 5}, {4, 6}}, {{2, 3}, {3, 3}}, {{1, 7}, {2, 7}}},
 {{{4, 5}, {4, 6}}, {{2, 3}, {3, 3}}, {{3, 2}, {3, 3}}},
 {{{4, 5}, {4, 6}}, {{2, 3}, {3, 3}}, {{3, 4}, {4, 4}}},
 {{{4, 5}, {4, 6}}, {{2, 3}, {3, 3}}, {{4, 2}, {4, 3}}},
 {{{4, 5}, {4, 6}}, {{2, 3}, {3, 3}}, {{4, 3}, {4, 4}}},
 {{{4, 5}, {4, 6}}, {{2, 3}, {3, 3}}, {{4, 4}, {4, 5}}},
 {{{4, 5}, {4, 6}}, {{2, 3}, {3, 3}}, {{4, 6}, {4, 7}}}}
```

```
Length[%]
```

```
42
```

```
Select[%%, Length[Union @@ ##] == 2 Length[##] &]
```

```
{{{{1, 7}, {1, 8}}, {{2, 3}, {3, 3}}, {{3, 4}, {4, 4}}},
 {{{1, 7}, {1, 8}}, {{2, 3}, {3, 3}}, {{4, 2}, {4, 3}}},
 {{{1, 7}, {1, 8}}, {{2, 3}, {3, 3}}, {{4, 3}, {4, 4}}},
 {{{1, 7}, {1, 8}}, {{2, 3}, {3, 3}}, {{4, 4}, {4, 5}}},
 {{{1, 7}, {1, 8}}, {{2, 3}, {3, 3}}, {{4, 6}, {4, 7}}},
 {{{3, 4}, {3, 5}}, {{2, 3}, {3, 3}}, {{1, 7}, {2, 7}}},
 {{{3, 4}, {3, 5}}, {{2, 3}, {3, 3}}, {{4, 2}, {4, 3}}},
 {{{3, 4}, {3, 5}}, {{2, 3}, {3, 3}}, {{4, 3}, {4, 4}}},
 {{{3, 4}, {3, 5}}, {{2, 3}, {3, 3}}, {{4, 4}, {4, 5}}},
 {{{3, 4}, {3, 5}}, {{2, 3}, {3, 3}}, {{4, 6}, {4, 7}}},
 {{{3, 5}, {4, 5}}, {{2, 3}, {3, 3}}, {{1, 7}, {2, 7}}},
 {{{3, 5}, {4, 5}}, {{2, 3}, {3, 3}}, {{3, 4}, {4, 4}}},
 {{{3, 5}, {4, 5}}, {{2, 3}, {3, 3}}, {{4, 2}, {4, 3}}},
 {{{3, 5}, {4, 5}}, {{2, 3}, {3, 3}}, {{4, 3}, {4, 4}}},
 {{{3, 5}, {4, 5}}, {{2, 3}, {3, 3}}, {{4, 6}, {4, 7}}},
 {{{4, 5}, {4, 6}}, {{2, 3}, {3, 3}}, {{1, 7}, {2, 7}}},
 {{{4, 5}, {4, 6}}, {{2, 3}, {3, 3}}, {{3, 4}, {4, 4}}},
 {{{4, 5}, {4, 6}}, {{2, 3}, {3, 3}}, {{4, 2}, {4, 3}}},
 {{{4, 5}, {4, 6}}, {{2, 3}, {3, 3}}, {{4, 3}, {4, 4}}}}
```

```
Length[%]
```

```
19
```

Using the first three dominoes, there are $6 \times 1 \times 7 = 42$ possibilities, reduced to 19 after elimination of conflicts. Placing the first 13 dominoes involves considering 653184 cases, of which only four have no conflicts.

```
Times @@ (Length /@ options[[ ;; 13]])
```

```
653 184
```

```
Select[
 Flatten[Outer[List, Sequence @@ options[[ ;; 13]], 1], 13 - 1],
 Length[Union @@ ##] == 2 Length[##] &
]
```

```
{{{{3, 5}, {4, 5}}, {{2, 3}, {3, 3}},
   {{1, 7}, {2, 7}}, {{4, 3}, {5, 3}}, {{1, 8}, {2, 8}},
   {{4, 6}, {5, 6}}, {{2, 4}, {3, 4}}, {{1, 1}, {2, 1}},
   {{3, 1}, {3, 2}}, {{6, 4}, {7, 4}}, {{5, 1}, {6, 1}},
   {{7, 2}, {7, 3}}, {{6, 5}, {7, 5}}},
  {{{3, 5}, {4, 5}}, {{2, 3}, {3, 3}}, {{1, 7}, {2, 7}},
   {{4, 3}, {5, 3}}, {{1, 8}, {2, 8}}, {{4, 6}, {5, 6}},
   {{2, 4}, {3, 4}}, {{1, 1}, {2, 1}}, {{3, 1}, {3, 2}},
   {{6, 4}, {7, 4}}, {{5, 1}, {6, 1}}, {{7, 2}, {7, 3}},
   {{7, 5}, {7, 6}}}, {{{3, 5}, {4, 5}},
   {{2, 3}, {3, 3}}, {{1, 7}, {2, 7}}, {{4, 3}, {5, 3}},
   {{1, 8}, {2, 8}}, {{4, 6}, {5, 6}}, {{2, 4}, {3, 4}},
   {{7, 3}, {7, 4}}, {{2, 1}, {2, 2}}, {{5, 1}, {5, 2}},
   {{1, 1}, {1, 2}}, {{3, 1}, {4, 1}}, {{6, 5}, {7, 5}}},
  {{{3, 5}, {4, 5}}, {{2, 3}, {3, 3}}, {{1, 7}, {2, 7}},
   {{4, 3}, {5, 3}}, {{1, 8}, {2, 8}},
   {{4, 6}, {5, 6}}, {{2, 4}, {3, 4}}, {{7, 3}, {7, 4}},
   {{2, 1}, {2, 2}}, {{5, 1}, {5, 2}}, {{1, 1}, {1, 2}},
   {{3, 1}, {4, 1}}, {{7, 5}, {7, 6}}}}
```

```
Length[%]
```

```
4
```

So the following code should work, but will take an unreasonable amount of time and memory. It is beautifully simple and short, but do not run it, as it probably would not finish in our lifetimes!

```
Select[
 Flatten[Outer[List, Sequence @@ options, 1],
  Length[options] - 1], Length[Union @@ ##] == 2 Length[##] &
]
```

■ Odometer Method

“To a hammer, everything looks like a nail.” A few years ago, I worked out an exhaustive search-and-collision detection algorithm based on a idea of a generalized odometer, and since then I have seen applications for it everywhere. It works here, too.

■ *Method*

Create a 28-digit generalized odometer, whose n^{th} digit refers to which option we are trying for the n^{th} domino. All digits start as 1; incrementing the odometer does not in general occur at the right end, but at the first digit (from the left) whose domino placement conflicts with that of any previous domino. A digit “rolls over” when it is incremented past its maximum value and must be reset to 1. Whenever a digit rolls over, also increment the digit to its left, just as in a real odometer. Each odometer digit has a separate maximum determined by the number of options available for that domino. When the first digit finally rolls over, all solutions have been found. We also accelerate the procedure by sorting the domino option list in increasing length.

```
(options = Sort[
    find[m9, #] & /@ bones[6] ,
    Length[#1] ≤ Length[#2] &]) // Column
```

```
{{{2, 3}, {3, 3}}}
{{{4, 6}, {5, 6}}}
{{{6, 1}, {6, 2}}}
{{{7, 1}, {7, 2}}}
{{{4, 3}, {5, 3}}, {{4, 5}, {5, 5}}}
{{{1, 6}, {1, 7}}, {{1, 8}, {2, 8}}}
{{{4, 2}, {5, 2}}, {{4, 4}, {5, 4}}}
{{{5, 5}, {6, 5}}, {{6, 4}, {6, 5}}}
{{{2, 1}, {2, 2}}, {{2, 2}, {2, 3}}, {{3, 1}, {3, 2}}}
{{{5, 1}, {5, 2}}, {{6, 3}, {7, 3}}, {{6, 4}, {7, 4}}}
{{{1, 1}, {1, 2}}, {{1, 3}, {2, 3}}, {{5, 1}, {6, 1}}}
{{{3, 1}, {4, 1}}, {{4, 1}, {5, 1}}, {{7, 2}, {7, 3}}}
{{{2, 3}, {2, 4}}, {{6, 5}, {7, 5}}, {{7, 5}, {7, 6}}}
{{{1, 2}, {2, 2}}, {{1, 6}, {2, 6}}, {{2, 7}, {2, 8}}}
{{{4, 1}, {4, 2}}, {{4, 7}, {5, 7}}, {{4, 8}, {5, 8}}}
{{{1, 4}, {2, 4}}, {{1, 5}, {2, 5}}, {{2, 8}, {3, 8}}}
{{{5, 6}, {6, 6}}, {{6, 6}, {6, 7}}, {{7, 6}, {7, 7}}}
{{{2, 4}, {3, 4}}, {{2, 5}, {3, 5}},
 {{3, 5}, {3, 6}}, {{3, 6}, {4, 6}}}
{{{1, 1}, {2, 1}}, {{2, 1}, {3, 1}},
 {{7, 3}, {7, 4}}, {{7, 4}, {7, 5}}}
{{{2, 2}, {3, 2}}, {{2, 6}, {2, 7}},
 {{3, 2}, {4, 2}}, {{4, 7}, {4, 8}}}
```

```
{{{5, 5}, {5, 6}}, {{6, 2}, {7, 2}},
 {{6, 8}, {7, 8}}, {{7, 7}, {7, 8}}}
{{{2, 5}, {2, 6}}, {{2, 6}, {3, 6}},
 {{2, 7}, {3, 7}}, {{3, 7}, {4, 7}}, {{3, 8}, {4, 8}}}
{{{1, 2}, {1, 3}}, {{1, 3}, {1, 4}},
 {{1, 4}, {1, 5}}, {{1, 5}, {1, 6}}, {{6, 1}, {7, 1}}}
{{{2, 4}, {2, 5}}, {{3, 6}, {3, 7}},
 {{3, 7}, {3, 8}}, {{6, 5}, {6, 6}}, {{6, 6}, {7, 6}}}
{{{1, 7}, {1, 8}}, {{3, 3}, {3, 4}}, {{3, 3}, {4, 3}},
 {{3, 4}, {3, 5}}, {{3, 5}, {4, 5}}, {{4, 5}, {4, 6}}}
{{{5, 6}, {5, 7}}, {{5, 7}, {5, 8}}, {{5, 7}, {6, 7}},
 {{5, 8}, {6, 8}}, {{6, 7}, {6, 8}}, {{6, 7}, {7, 7}}}
{{{1, 7}, {2, 7}}, {{3, 2}, {3, 3}},
 {{3, 4}, {4, 4}}, {{4, 2}, {4, 3}},
 {{4, 3}, {4, 4}}, {{4, 4}, {4, 5}}, {{4, 6}, {4, 7}}}
{{{5, 2}, {5, 3}}, {{5, 2}, {6, 2}},
 {{5, 3}, {5, 4}}, {{5, 3}, {6, 3}}, {{5, 4}, {5, 5}},
 {{5, 4}, {6, 4}}, {{6, 2}, {6, 3}}, {{6, 3}, {6, 4}}}
```

```
odometer = Table[1, {Length[options]}]
```

```
{1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}
```

```
odometermax = Length /@ options
```

```
{1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 3,
 3, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 6, 6, 7, 8}
```

Notice that the first four odometer digits can only be 1; each starts at 1 and has a maximum of 1.

```
incOdometer[n0_Integer: Length[odometer]] := Module[{n = n0},
  If[n ≤ 0 || n > Length[odometer],
   If[debug, Print["Error: odometer index out of bounds: ",
     n, "∉[0,", Length[odometer], "]"]];
   Return[False]
  ];
  odometer⟦n⟧++; (* increment the requested digit *)
  While[odometer⟦n⟧ > odometermax⟦n⟧, (* digit over max *)
   n--; (* back up one digit *)
   If[n == 0, Return[False]]; (* done! *)
   odometer⟦n⟧++(* increment this digit, too *)
  ];
  odometer = PadRight[odometer⟦ ;; n⟧, Length[odometer], 1];
  (* restart all digits to the right *)
  True
 ]
```

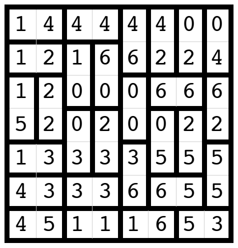
To see or use the parts of options specified by the odometer, we use the function `MapThread`.

```
MapThread[Part, {options, odometer}]

{{{2, 3}, {3, 3}}, {{4, 6}, {5, 6}},
 {{6, 1}, {6, 2}}, {{7, 1}, {7, 2}},
 {{4, 3}, {5, 3}}, {{1, 6}, {1, 7}}, {{4, 2}, {5, 2}},
 {{5, 5}, {6, 5}}, {{2, 1}, {2, 2}}, {{5, 1}, {5, 2}},
 {{1, 1}, {1, 2}}, {{3, 1}, {4, 1}}, {{2, 3}, {2, 4}},
 {{1, 2}, {2, 2}}, {{4, 1}, {4, 2}}, {{1, 4}, {2, 4}},
 {{5, 6}, {6, 6}}, {{2, 4}, {3, 4}}, {{1, 1}, {2, 1}},
 {{2, 2}, {3, 2}}, {{5, 5}, {5, 6}}, {{2, 5}, {2, 6}},
 {{1, 2}, {1, 3}}, {{2, 4}, {2, 5}}, {{1, 7}, {1, 8}},
 {{5, 6}, {5, 7}}, {{1, 7}, {2, 7}}, {{5, 2}, {5, 3}}}
```

Here is the program that more or less immediately returns the answer(s).

```
odometer = Table[1, {Length[options]}];
done = False;
While[! done,
 Do[
  If[k > Length[odometer],
   Print["odometer reading: ", odometer];
   Print[
    puzzle[m9, MapThread[Part, {options, odometer}]]];
   incOdometer[];
   Break[]
  ];
  If[Length[Union @@ ##] ≠ 2 Length[##] &[
     MapThread[Part, {options, odometer}]〚;; k〛],
   done = ! incOdometer[k]; Break[]],
  {k, 1, Length[odometer] + 1}]
]
```

```
odometer reading: {1, 1, 1, 1, 1, 2, 2, 1, 1,
  1, 1, 1, 3, 2, 2, 2, 2, 1, 3, 3, 4, 5, 2, 2, 5, 4, 1, 8}
```



As expected, there is only one odometer reading that works; that is, only one choice of domino placements solves the puzzle. The generalized odometer method works best for situations with a large number of variables taking on values that can be calculated in advance, particularly if the possible values are the same for all variables or vary in a way that can be easily specified. Here the options have to be recomputed for each new puzzle, making it less efficient than the previous method.

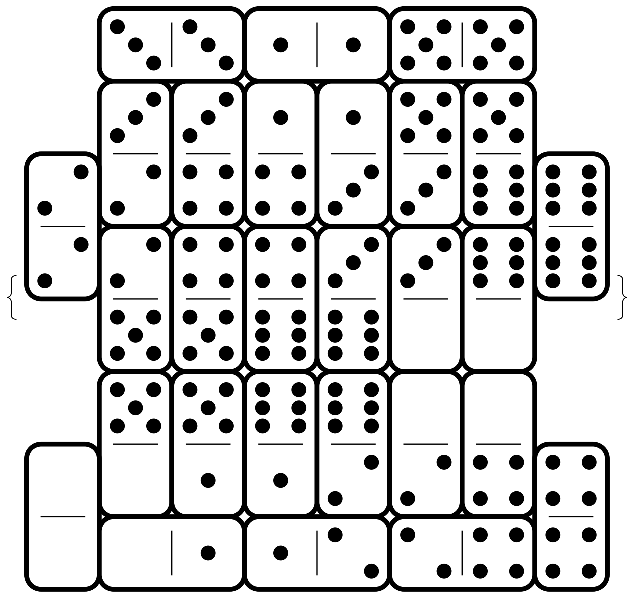
■ Solving Other Boards

■ *Quadrilles*

A “quadrilles” puzzle [5], an idea credited to French mathematician Edouard Lucas, can be divided into 2×2 blocks, each containing the same number. Since the following figure does not completely fill a rectangular array, we add empty strings.

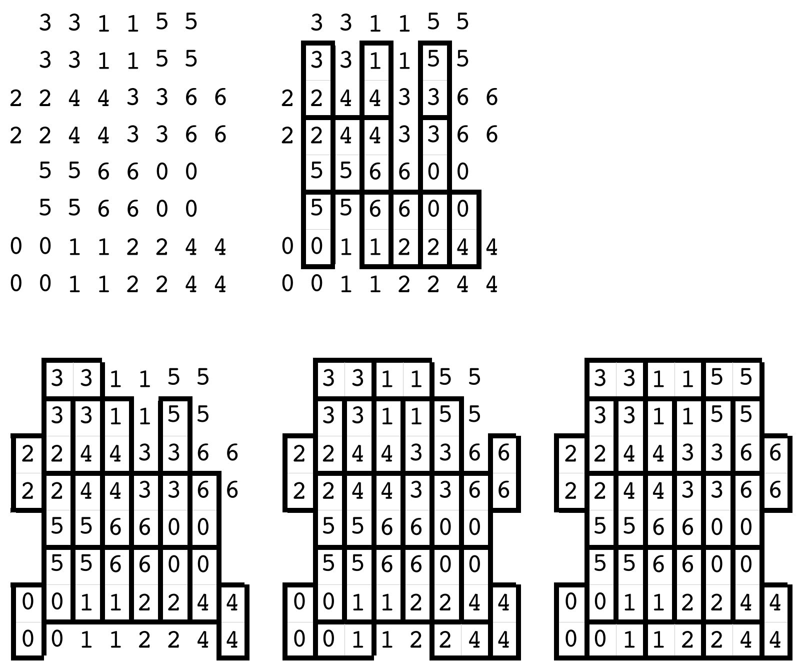
```
quadrille = {
   {"", 3, 3, 1, 1, 5, 5, ""},
   {"", 3, 3, 1, 1, 5, 5, ""},
   {2, 2, 4, 4, 3, 3, 6, 6},
   {2, 2, 4, 4, 3, 3, 6, 6},
   {"", 5, 5, 6, 6, 0, 0, ""},
   {"", 5, 5, 6, 6, 0, 0, ""},
   {0, 0, 1, 1, 2, 2, 4, 4},
   {0, 0, 1, 1, 2, 2, 4, 4}
  };
```

```
displayDottedPuzzle /@
 solvePuzzle[puzzle[quadrille, {}, bones[6]]]
```



This particular quadrille has only one solution. At each step there are a large number of forced locations or pieces, and all 28 dominoes are placed in only four iterations.

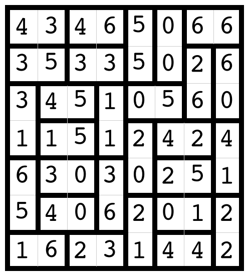
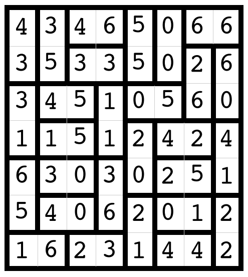
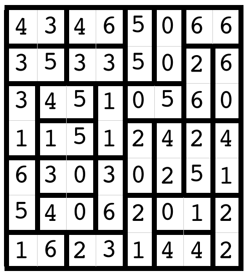
```
Row[
 Most@FixedPointList[step, puzzle[quadrille, {}, bones[6]]],
 "    "]
```

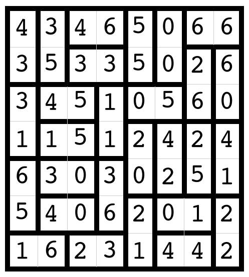
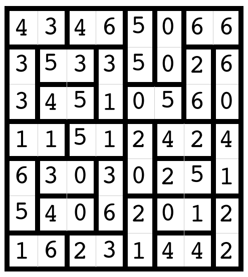
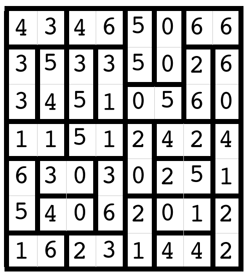


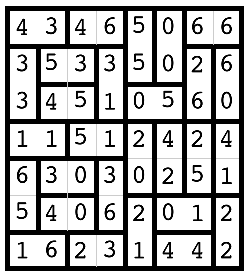
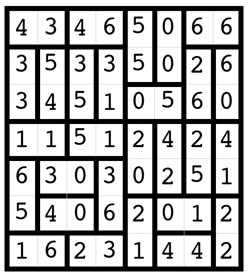
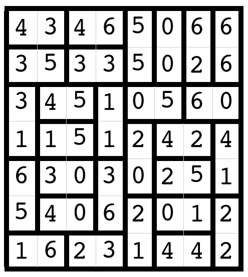
■ A "Too-Easy" Puzzle

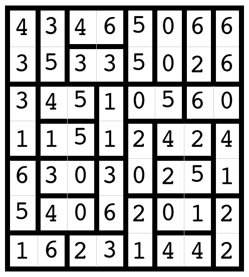
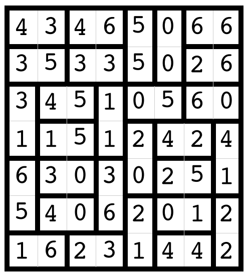
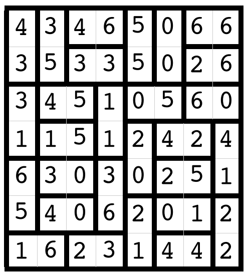
Now for a puzzle with so many different ways to solve it that one feels that almost anything will work [5]!

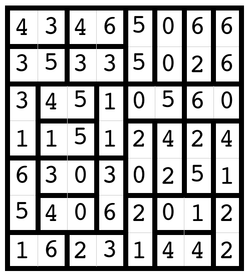
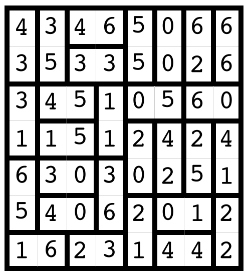
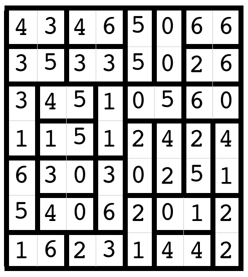
```
solvePuzzle @ puzzle[{
   {4, 3, 4, 6, 5, 0, 6, 6},
   {3, 5, 3, 3, 5, 0, 2, 6},
   {3, 4, 5, 1, 0, 5, 6, 0},
   {1, 1, 5, 1, 2, 4, 2, 4},
   {6, 3, 0, 3, 0, 2, 5, 1},
   {5, 4, 0, 6, 2, 0, 1, 2},
   {1, 6, 2, 3, 1, 4, 4, 2}
  }, {}, bones[6]]
```

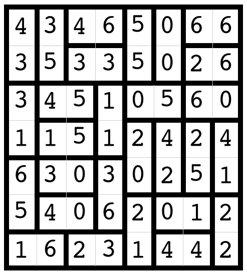
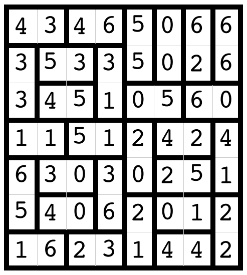
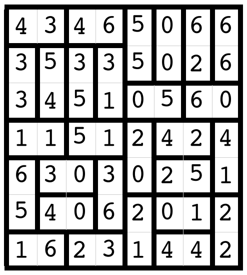
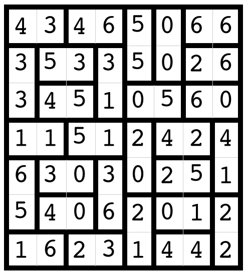
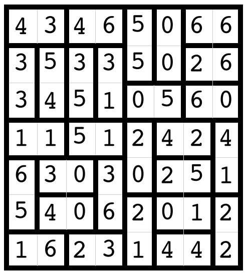
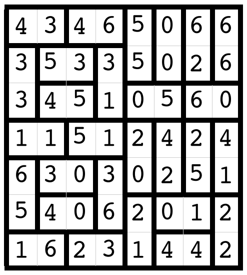
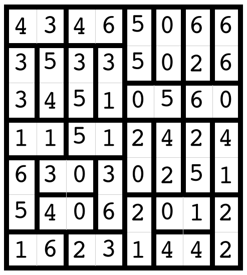
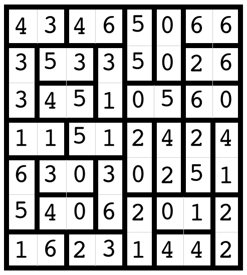
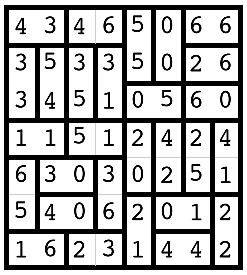
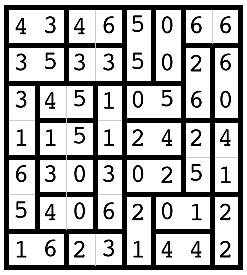
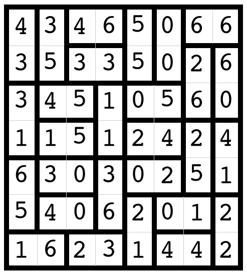
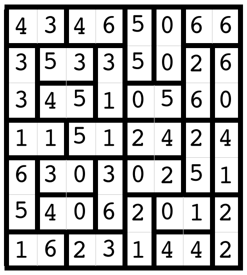
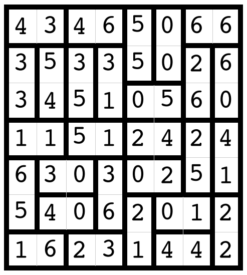
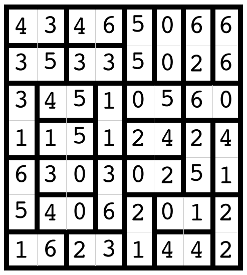
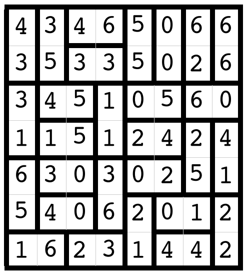
{
4 3 4 6 5 0 6 6
3 5 3 3 5 0 2 6
3 4 5 1 0 5 6 0
1 1 5 1 2 4 2 4
6 3 0 3 0 2 5 1
5 4 0 6 2 0 1 2
1 6 2 3 1 4 4 2
,

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
,

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
,

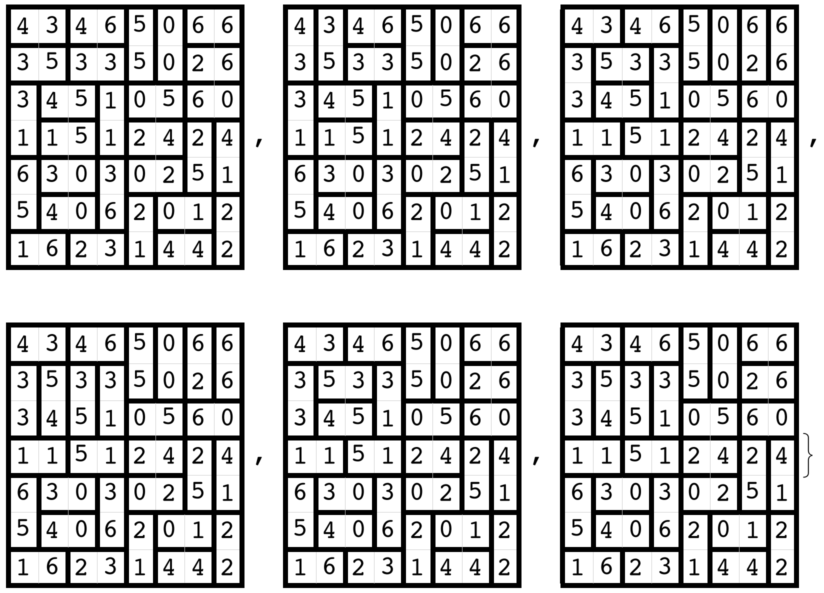

4 3 4 6 5 0 6 6
3 5 3 3 5 0 2 6
3 4 5 1 0 5 6 0
1 1 5 1 2 4 2 4
6 3 0 3 0 2 5 1
5 4 0 6 2 0 1 2
1 6 2 3 1 4 4 2
,

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
,

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
,


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
,

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
,

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
,


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
,

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
,

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
,


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
,

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
,

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
,


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
,

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
,

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
,

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
,

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
,

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
,

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
,

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
,

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
,

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
,

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
,

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
,

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
,

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
,

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
,



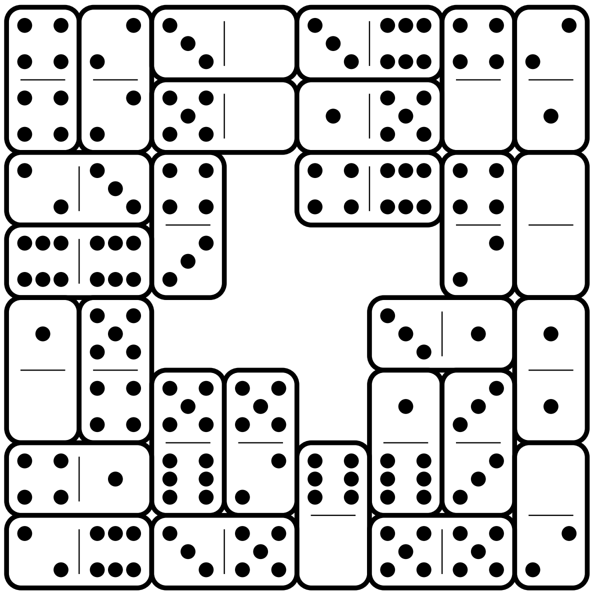
```
Length[%]
```

```
36
```

■ *A Puzzle with a Hole in the Middle*

If a puzzle is nonrectangular or has intentional gaps in it, such as the one shown below [4], simply embed it in a larger rectangle, and indicate the gaps by empty strings.

```
displayDottedPuzzle /@ solvePuzzle @ puzzle[{
      {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}
     }, {}, bones[6]] // Row
```



■ Last Thoughts

It seems likely that the online or downloadable domino puzzle generators effectively lay out the dominoes to create a grid that is guaranteed to be solvable. But even if all puzzles presented can be solved, a number of questions spring to mind:

For given grid dimensions, how many different solutions are there? (The three methods derived above solve individual puzzles, but what if the numbers are rearranged in a given grid in all possible ways?)

For given grid dimensions, what fraction of the possible puzzles has only one solution, and in general, for all k, what fraction of the puzzles has k solutions? What is the largest number of solutions possible?

Bear in mind that in the sense of the functions developed here, a "solution" is a merely a list of domino locations, so different puzzles of the same dimensions can have the same solution just by permuting the underlying grid numbers or rearranging them in other valid ways. In the interest of increased clarity, define a *solution schema* as a layout of dominoes face-down on a board. Now we can talk about the number of possible distinct schemas for a given puzzle grid.

What about writing a program that generates all solution schemas for a given board, ignoring the numbers? This could be done by modifying either the function `solvePuzzle` or the function `odometerSolve`, neither of which can quite do the job as written. (Yes, I did try them on a board filled with 0 entries, but they would need to be tweaked to expect a bone pile of double-zero dominoes.)

Finally, it is interesting that the first solution method worked so well, basically following how a human would decide which domino to play next. The code for the brute-force method is the simplest, but impractical without massive parallel processing. The odometer method works well, but here not as fast as the "human" method, and in any case may not be as transparent to the reader. There is more than one way to solve a puzzle! And if you spend much time thinking about a puzzle, other methods and other questions will probably occur to you.

■ Acknowledgments

I thank my colleagues at Southern Adventist University who have encouraged me, the folks at Wolfram Research who have occasionally helped me, and Claryce, who has put up with me in all my most puzzling moods.

■ References

[1] E. W. Weisstein. "Domino Tiling" from Wolfram *MathWorld*—A Wolfram Web Resource. mathworld.wolfram.com/DominoTiling.html.

[2] Domino-Games.com. "Domino Puzzles." (Sep 4, 2014) www.domino-games.com/domino-puzzles.html.

[3] "Dominosa." (Sep 4, 2014) www.puzzle-dominosa.com.

[4] Yoogi Games. "Domino Puzzle Puzzles." (Sep 4, 2014) syndicate.yoogi.com/domino.

[5] J. Köller. "Domino Puzzles." (Sep 4, 2014) www.mathematische-basteleien.de/dominos.htm.

K. E. Caviness, "Three Ways to Solve Domino Grids," *The Mathematica Journal*, 2014.
dx.doi.org/doi:10.3888/tmj.16-10.

About the Author

Ken Caviness teaches at Southern Adventist University, a liberal arts university near Chattanooga. Since obtaining a Ph.D. in physics (relativity and nuclear physics) from the University of Massachusetts Lowell, he has taught math and physics in Rwanda, Texas, and Tennessee. His interests include both computer and human languages (including Esperanto), and he has used *Mathematica* since Version 1, both professionally and recreationally.

Kenneth E. Caviness
Department of Physics & Engineering
Southern Adventist University
PO Box 370
Collegedale, TN 37315-0370
caviness@southern.edu