让计算机解决和产生数独

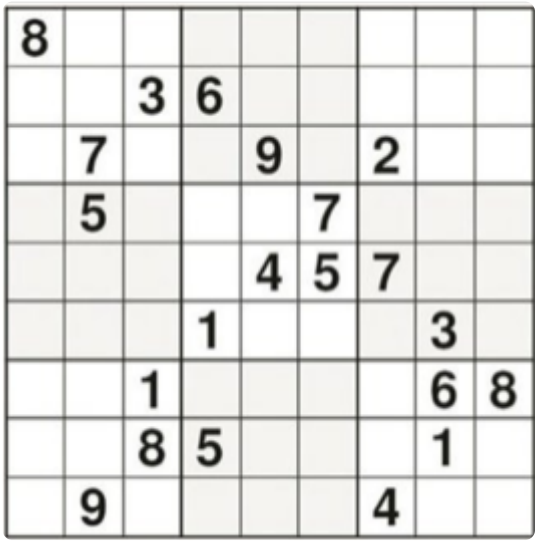
Wed, Oct 21, 2015 in Interesting

> 人每天都得熬过一段时间，独自躲在密室中，扬起头45度角，眼神涣散地等待……在单调乏味的如厕时段，你不妨带上卷数独卫生纸。数独简单易玩，可以让你一边新陈代谢，一边训练大脑，人生大事、游戏通关两不误！

在开始阅读这篇文章之前，您可能需要对下列知识有所了解：

- 回溯算法
- 数独的规则以及一些初等解决方法
- 部分的python 2.x语法（我们将要选择python解决这个问题）
- 一道相关的题目：https://oj.leetcode.com/problems/sudoku-solver/
- Peter Norvig 的研究：http://norvig.com/sudoku.html

如何解数独



这，是一个数独，如果之前我们对数独有过了解的话，我们就会知道，要解出这样一个数独，就必须将数字填入这些方格内，并且使得每一行，每一列，每一个大方格都含且只含有一个数字，这自然不必多说，但是想要电脑高效地解决这个问题却无从下手。我们要解决的，就是这个令人无从下手的问题。

让计算机读懂数独

计算机并非天生就能读懂数独的，我们需要将所需要解的数独转换成计算机能够接受的格式，这里，我们有两种选择：一维数组，二维数组，或者字典（在phthon里的dic），在这里，我们选择二维数组的形式也就是下面这个样子：

```
1. [".....7..9",".4..812..","...9...1.","..53...7
   2","293....5.",".....53..","8...23...","7...
   5..4.","531.7...."]
```

上面这个python二维数组代表了这样一个数独：

```
3 1 2 |5 4 7 |8 6 9
9 4 7 |6 8 1 |2 3 5
6 5 8 |9 3 2 |7 1 4
------+------+------
1 8 5 |3 6 4 |9 7 2
2 9 3 |7 1 8 |4 5 6
4 7 6 |2 9 5 |3 8 1
------+------+------
8 6 4 |1 2 3 |5 9 7
7 2 9 |8 5 6 |1 4 3
5 3 1 |4 7 9 |6 2 8
```

（当然，这是解出来以后的数独）

利用回溯

如果仅仅使用回溯算法的话，我们可以将代码控制在几十张之内，以为其思路之简单，仅仅是在每一个格子上尝试所有可能的数，每当尝试到非法的数时，就回溯，

若合法，则继续在下一个格子上尝试所有可能的数，若下一个格子上所有数均为非法，则也回溯，就是这样简单的算法也足够解决大多数的问题。

```
1. class Solution:
2.     # @param board, a 9x9 2D array
3.     # Solve the Sudoku by modifying the input
    board in-place.
4.     # Do not return any value.
5.     def solveSudoku(self, board):
6.         if not board or len(board) != 9 or len
   (board[0]) != 9:return
7.         self.helper(board,0)
8.         return board
9.     def helper(self,board,i):
10.         if board[i / 9][i % 9] != '.':return s
   elf.helper(board,i + 1)
11.         for j in range(1,10):
12.             board[i / 9] = board[i / 9][:(i %
   9)] + [str(j)] + board[i / 9][(i % 9 + 1):]
13.             if self.isValidSudoku(board) and (
   i == 80 or self.helper(board,i + 1)):
14.                 return True
15.         board[i / 9] = board[i / 9][:(i % 9)]
   + ['.'] + board[i / 9][(i % 9 + 1):]
16.         return False
17.     # @param board, a 9x9 2D array
18.     # @return a boolean
19.     def isValidSudoku(self, board):
20.         if not board or len(board) != 9 or len
   (board[0]) != 9:return False
21.         rows = [[False for var in range(9)] fo
   r var1 in range(9)]
22.         cols = [[False for var in range(9)] fo
   r var1 in range(9)]
23.         blocks = [[False for var in range(9)]
   for var1 in range(9)]
24.         for i in range(9):
25.             for j in range(9):
26.                 if board[i][j] == '.':
27.                     continue
28.                 valueint = int(board[i][j]) -
   1
```

```
                if rows[i][valueint] or cols[j
    ][valueint] or blocks[i - i % 3 + j / 3][value
    int]:
                    return False
                else:
                    rows[i][valueint] = cols[j
    ][valueint] = blocks[i - i % 3 + j / 3][valuei
    nt] = True
        return True
```

但是这样做的后果是要解出一个数独通常要花5到10秒的时间—太久了，而且，这也不是我们一般的解决数独的方法，我们经常采用下面这些算法来解题：

唯一解法

当某行已填数字的宫格达到8个，那么该行剩余宫格能填的数字就只剩下那个还没出现过的数字了。成为行唯一解。

唯余解法

唯余解法就是某宫格可以添入的数已经排除了8个,那么这个宫格的数字就只能添入那个没有出现的数字。

我们在进行猜测之前还会用一些其他的方法进行这些“预处理”，这些技巧并不总能帮助我们解出数独，但是却很有帮助。

一般的解题步骤

我们在做数独问题时，经常采用这样的方法：将某个格子的可能出现的数字（可以有很多歌）标在格子上，通过一些手段观察这些格子，从而确定某些格子的值，继而推导出其它格子的值。

推倒重来

无疑我们需要换一种思路，仅仅使用回溯是不够的。

首先，我们需要一个这样的·phthon数组：

```
possiblenumbers = [[[i for i in range(1,10)]for j in range(9) ]for k in range(9)]
```

这个数组是一个三维数组，它用来表示在每一个格子上的可能的数字,比如

```
possiblenumbers[0][0] = [1,2,3,4,5,6,7,8,9]表示数独坐标为（0,0）的格子的可能数字为1至9
```

同轴数

我们定义与一个数同行，同列，同格的数为这个数的同轴数，为了程序运行是快速地得到一个位置数字的同轴数，我们在预处理中完成这个工作。

我们定义一个python数组来分别存储同行，同列，同格的同轴数：

```
peercross = [[[[(j,i) for i in range(9)],[(i,k) for i in range(9)],[(j - j % 3 + i / 3,k - k % 3 + i % 3)for i in range(9)]] for k in range(9)]for j in range(9)]
```

这个数组分别存储了每个grid的同行，同列，同格的同轴数，比如，peercross[3][3]：

```
[

[(3, 0), (3, 1), (3, 2), (3, 3), (3, 4), (3, 5), (3, 6), (3, 7), (3, 8)],

[(0, 3), (1, 3), (2, 3), (3, 3), (4, 3), (5, 3), (6, 3), (7, 3), (8, 3)],

[(3, 3), (3, 4), (3, 5), (4, 3), (4, 4), (4, 5), (5, 3), (5, 4), (5, 5)]

```

```
9. ]
```

但是这个数据结构有时并不是非常实用，有时我们需要遍历一个grid的所有同轴数时，peercross数组并不能提供直接的遍历方法（由于其中有重复元素），所以我们设计了另一个数据结构peers：

```
1. peers = [[set(sum(peercross[j][k],[])) - set
   ([(j,k)]) for k in range(9)] for j in range(9
   )]
```

peers 将所有同轴数一并存储，例如peers[3][3]就是这样的：

```
1. set([(7, 3), (3, 2), (1, 3), (5, 5), (3, 0), (
   5, 4), (3, 1), (8, 3), (3, 6), (6, 3), (4, 5),
   (2, 3), (4, 3), (3, 8), (3, 7), (0, 3), (3, 5
   ), (3, 4), (4, 4), (5, 3)])
```

预处理

预处理的过程中，我们模仿大多数数独爱好者的解题方法，尽可能多的在格子上填充数字。预处理分为如下几步：

1. 生成各格子的数据结构和相关存储同轴数的数组

```
1. self.grids = [(i,j) for j in range(9) for
    i in range(9)]
2. self.peercross = [[[[(j,i) for i in range(
   9)],[(i,k) for i in range(9)],[(j - j % 3
   + i / 3,k - k % 3 + i % 3)for i in range(9
   )]] for k in range(9)]for j in range(9)]
3. self.peers = [[set(sum(self.peercross[j][k
   ],[])) - set([(j,k)]) for k in range(9)] f
   or j in range(9)]
```

2. 将题面数字填入格子中

3. 依照唯一解法和唯余解法进行推理

其中，第二步和第三步其实并不应该分开，我们完全可以再讲数字填入格子的同时进行推理：

1. 设每个格子的可能数字均为1~9
2. 对于题面中的每一个数字，将其填进响应的格子，并且根据两个定理消去同轴的相应数字，当填入某一个格子的数字不在这个格子的可能数字范围中时，说明这个数独根本就没有可行解。

怎样消去一个可能的数字

如果是简单地消去一个可能的数字，那么这个过程会很简单，比如某个格子的所有可能数字是[1,2,3,4,5,6,7,8,9]，呢么假如消去一个可能的数字8，那么很显然剩余的数字[1,2,3,4,5,6,7,9]，这有并没有什么难度，可是不要忘记，我们在填入和消去数字的同时也在推理。

首选，我们应该考虑唯一解法，假如有这样一个格子，它可能的数字是[1,2]，这是我们要是消去一个1，那么这个格子唯一可能的值就是2了，这时候，由于这个格子的值已经确定，我们就需要消去它所有同轴数中的2。

其次，我们需要考录唯余解法：当我们从[1,2]中消去一个2时，我们应该在这个数的同行，同列，和同大格中寻找是否存在一个格子，在这个格子的某一个轴中，仅有这个格子能够填下2，如果有这种格子存在，我们也需要将2填入。

经过上面的描述，我们可以很方便地看懂下面这几行代码：

```
1.      #@return whether elimatesuccess or not
2.      def eliminate(self,x,y,value,possiblenumbe
   rs):
3.          if value not in possiblenumbers[x][y]:
4.              return True
5.          if len(possiblenumbers[x][y]) == 1:
6.              return False
```

```
            possiblenumbers[x][y].remove(value)
            #stg1 策略一：唯一解法
            if len(possiblenumbers[x][y]) == 0 or
     len(possiblenumbers[x][y]) == 1 and  not all
    ([self.eliminate(i,j,possiblenumbers[x][y][0],
    possiblenumbers) for (i,j) in self.peers[x][y]
    if (possiblenumbers[x][y][0]) in possiblenumbe
    rs[i][j]]):
                 return False
            #stg2 策略2：唯余解法
            for k in self.peercross[x][y]:
                tem = [(i,j) for (i,j) in k if val
    ue in possiblenumbers[i][j]]
                if len(tem) == 1 and len(possiblen
    umbers[tem[0][0]][tem[0][1]]) > 1 and  not sel
    f.fill(tem[0][0],tem[0][1],value,possiblenumbe
    rs):
                    return False
            return True
```

在预处理中究竟需要多少策略?

仅仅运用唯一解法和唯余解法已经能解出一些简单的数独题目，更多更高阶的策略能使在进行回溯遍历前确定的格子更多。但是，并没有证据显示在用回溯算法之前运用一些预处理策略能够提高多少回溯的效率，所以，在这里，我们认为这两种初等策略已经足够了。

填充一个数字

我们之所以将填充数字放在消去可能的数字之后是因为我们需要应用消去的的方法来填充数字，因为我们需要记住，我们之前假设过每一个格子一开始可能填入的数字是1到9，而设定一个格子的值无异于消去这个格子中所有其他可能的值：

```
def fill(self,x,y,value,possiblenumbers):
        if value not in range(1,10) and value
    not in "123456789":return True
        if int(value) not in possiblenumbers[x
    ][y] or x not in range(9) or y not in range(9
```

```
    ):return False
4.          if len(possiblenumbers[x][y]) == 1 or
     all([self.eliminate(x,y,i,possiblenumbers) fo
    r i in  [j for j in possiblenumbers[x][y] if j
    != int(value)]]):
5.              return True
6.          return False
```

到这里，我们终于可以写出预处理的代码了：

```
1. def preProcess(self,board,possiblenumbers):
2.          for i in range(81):
3.              if not self.fill(i / 9,i % 9,board
    [i / 9][i % 9],possiblenumbers):
4.                  return False
5.          return True
```

很简单，是不是?

回溯的起点

预处理兵不解决数独，而仅仅只是能够填充一些未知的格子而已，然而我们的目的是填充出整个数独，这样的话我们不得不回到了那个原来的话题：回溯

但是从哪里开始回溯呢？我们的第一份代码是从第一个格子往下一路回溯，这样的回溯并没有什么技巧可言，我们需要精心挑选一些格子，这些格子的可能的数字的个数最少，从而我们蒙中的概率也就会更大，比如说，一个仅有两种可能取值的格子，我们蒙中的概率就有一半，而一个有三种可能取值的格子，我们蒙中的概率就仅仅有个三分之一而已。

在回溯的同时推理

回溯是一个不断在格子上蒙数字的过程，我们并不希望这个过程仅仅依靠猜想，而是有一部分推理的成分，怎样推理呢?我们在利用回溯法向一个不确定的格子中填入一个数字的时候，应该运用与预处理相同的方法，将

填入的过程分解为排除每个其他可能数字的过程，也就是说，我们可以原封不动地照搬之前定义过的fill方法

多个解与没有解

一般来说，我们希望传入的数独至少有一个解，并且一旦我们找到一个解，其他的解就无关紧要，但是如果我们想进一步在这个程序的基础实现生成数独谜题的功能的话，判断一个数独书否有唯一解就是必须的了，所以我们的程序在搜索完全部解空间或者遇到两个不同的可行解之前并不能停止。

为了记录解，我们定义self.finalsolution 当他为False是表示这个数独没有解，为True是表示数独有多个解，否则这个值为那个唯一的解。
说了这么多，其实递归求解的代码也没有多少：

```
1. def reSlove(self,possiblenumbers,deepth):
2.         if all(len(possiblenumbers[i][j]) == 1
   for (i,j) in self.grids):
3.             if not self.finalsolution:
4.                 self.finalsolution = copy.deep
   copy(possiblenumbers)
5.             else:
6.                 self.finalsolution = True
7.             return
8.         length,x,y = min((len(possiblenumbers[
   i][j]),i,j) for (i,j) in self.grids if len(pos
   siblenumbers[i][j]) > 1)
9.         posibility = copy.copy(possiblenumbers
   [x][y])
10.        for i in posibility:
11.            copyindex = copy.deepcopy(possible
   numbers)
12.            if self.fill(x,y,i,copyindex) and
    self.finalsolution != True:
13.                self.reSlove(copyindex,deepth
   + 1)
14.        #print('deepth: %s ' %deepth)
15.        return
```

解数独所需要的所有代码：

```
1. import copy
2. class Solution:
3.     # @param board, a 9x9 2D array
4.     # Solve the Sudoku by modifying the input
    board in-place.
5.     # Do not return any value.
6.     def cross(self,A,B):
7.         "Cross product of elements in A amd el
   ements in B"
8.         return [a + b for a in A for b in B]
9.     def __init__(self):
10.         self.grids = [(i,j) for j in range(9)
   for i in range(9)]
11.         self.peercross = [[[[(j,i) for i in ra
   nge(9)],[(i,k) for i in range(9)],[(j - j % 3
   + i / 3,k - k % 3 + i % 3)for i in range(9)]]
   for k in range(9)]for j in range(9)]
12.         self.peers = [[set(sum(self.peercross[
   j][k],[])) - set([(j,k)]) for k in range(9)] f
   or j in range(9)]
13.         self.finalsolution = False
14.
15.     def solveSudoku(self, board):
16.         self.finalsolution = False
17.         possiblenumbers = [[[i for i in range(
   1,10)]for j in range(9) ]for k in range(9)]
18.         assert (len(board) == 9 and all((len(b
   oard[i]) == 9) for i in range(9))) == True
19.         if not self.preProcess(board,possiblen
   umbers):return False
20.         self.reSlove(possiblenumbers,1)
21.         return self.finalsolution
22.
23.     def preProcess(self,board,possiblenumbers
   ):
24.         for i in range(81):
25.             if not self.fill(i / 9,i % 9,board
   [i / 9][i % 9],possiblenumbers):
26.                 return False
```

```
27.         return True
28.
29.     #@return whether the value can be filled i
    n the current grid
30.     def fill(self,x,y,value,possiblenumbers):
31.         if value not in range(1,10) and value
    not in "123456789":return True
32.         if int(value) not in possiblenumbers[x
    ][y] or x not in range(9) or y not in range(9
    ):return False
33.         if len(possiblenumbers[x][y]) == 1 or
     all([self.eliminate(x,y,i,possiblenumbers) fo
    r i in  [j for j in possiblenumbers[x][y] if j
    != int(value)]]):
34.             return True
35.         return False
36.
37.     #@return whether elimatesuccess or not
38.     def eliminate(self,x,y,value,possiblenumbe
    rs):
39.         if value not in possiblenumbers[x][y]:
40.             return True
41.         if len(possiblenumbers[x][y]) == 1:
42.             return False
43.         possiblenumbers[x][y].remove(value)
44.         #stg1
45.         if len(possiblenumbers[x][y]) == 0 or
     len(possiblenumbers[x][y]) == 1 and  not all
    ([self.eliminate(i,j,possiblenumbers[x][y][0],
    possiblenumbers) for (i,j) in self.peers[x][y]
    if (possiblenumbers[x][y][0]) in possiblenumbe
    rs[i][j]]):
46.              return False
47.         #stg2
48.         for k in self.peercross[x][y]:
49.             tem = [(i,j) for (i,j) in k if val
    ue in possiblenumbers[i][j]]
50.             if len(tem) == 1 and len(possiblen
    umbers[tem[0][0]][tem[0][1]]) > 1 and  not sel
    f.fill(tem[0][0],tem[0][1],value,possiblenumbe
    rs):
51.                 return False
52.         return True
53.
54.     def reSlove(self,possiblenumbers,deepth):
```

```
55.         if all(len(possiblenumbers[i][j]) == 1
    for (i,j) in self.grids):
56.             if not self.finalsolution:
57.                 self.finalsolution = copy.deep
    copy(possiblenumbers)
58.             else:
59.                 self.finalsolution = True
60.             return
61.         length,x,y = min((len(possiblenumbers[
    i][j]),i,j) for (i,j) in self.grids if len(pos
    siblenumbers[i][j]) > 1)
62.         posibility = copy.copy(possiblenumbers
    [x][y])
63.         for i in posibility:
64.             copyindex = copy.deepcopy(possible
    numbers)
65.             if self.fill(x,y,i,copyindex) and
     self.finalsolution != True:
66.                 self.reSlove(copyindex,deepth
    + 1)
67.         #print('deepth: %s ' %deepth)
68.         return
69.
70. s = Solution()
71. print '-------answer--------'
72. print  s.solveSudoku(["..9748...","7........",
    ".2.1.9...","..7...24.",".64.1.59.",".98...
    3..","...8.3.2.","........6","...2759.."])
```

这份代码并不算太长，一共也才不到百行，但是它却能够高效的解出数独：

```
. . .|4 . .|2 . .
1 . .|6 . .|9 . 3
5 2 3|. . .|. . .
-----------------------
8 . .|. 1 .|7 . .
. . .|3 . 7|. . .
. . 6|. 9 .|. . 2
-----------------------
. . .|. . .|1 9 6
7 . 4|. . 2|. . 5
. . 1|. . 3|. . .

9 6 7|4 3 1|2 5 8
1 4 8|6 2 5|9 7 3
5 2 3|8 7 9|6 4 1
-----------------------
8 5 9|2 1 6|7 3 4
4 1 2|3 8 7|5 6 9
3 7 6|5 9 4|8 1 2
-----------------------
2 3 5|7 4 8|1 9 6
7 9 4|1 6 2|3 8 5
6 8 1|9 5 3|4 2 7
```

```
. . 6 |. . . |. . 8
. 3 . |. 5 8 |. 9 1
. 9 . |. . 6 |4 3 .
------------------------
. . . |. 7 . |3 8 .
. . . |9 . 3 |. . .
. 2 4 |. 8 . |. . .
------------------------
. 7 2 |5 . . |. 4 .
6 8 . |7 3 . |. 2 .
4 . . |. . . |7 . .

1 4 6 |3 9 7 |2 5 8
2 3 7 |4 5 8 |6 9 1
8 9 5 |2 1 6 |4 3 7
------------------------
5 6 9 |1 7 2 |3 8 4
7 1 8 |9 4 3 |5 6 2
3 2 4 |6 8 5 |1 7 9
------------------------
9 7 2 |5 6 1 |8 4 3
6 8 1 |7 3 4 |9 2 5
4 5 3 |8 2 9 |7 1 6
```

然而更加重要的是，它能够确定一个数独谜题究竟是有唯一解，有多解，或是无解，这在接下来的数独题面生成算法中至关重要。

至于这个算法的效率，由于这个算法是在Peter Norvig的研http://norvig.com/sudoku.html 基础上写的，但是并没有使用Peter Norvig的字符串来存储可能的数字，所以在复制时使用了copy.deepcopy()而不是效率更高的copy()，所以时间上会略大于Peter Norvig的算法，根据Peter的试验结果，我们可以估计生成数独解的平均时间略大于0.01s。如果可能的话将来会用试验数据来补充这一部分的臆测。

先有鸡还是先有蛋?

先有数独题面，还是先有数独答案呢？从常识上说，没有问题，怎么能有答案呢？可是数独偏偏是个例外，如果我们想要得到一个数独题面，我们就需要先得到这个题面的答案，也就是一个填充满数字的九宫格，这很奇怪，不是么？但是为什么呢？为什么我们在生成题面之前要先知道答案呢？这就要牵扯到一个数独题面的生成算法。

挖洞算法

挖洞算法是一种常见的数独生成算法，使用这个算法首先需要一个数独终盘（就是一个填满数字的9*9宫格），然后我们随机地去掉一些数字，看这时的数独是有唯一解，如果还有唯一解就继续，我们持续进行这一过程直到再去掉一个数字都会导致多解数独的产生。

对于挖洞算法如果我描述的不是很清晰大家可以百度之，相信度娘可以给出比我更好的答案。

生成母鸡

现在我们已经知道怎么生成一个数独谜题了，首先，养一只母鸡（生成终盘），然后挖一个洞（挖洞算法），接着就可以开始下蛋了（生成数独谜题）。所以，第一步一定是生成一只母鸡，也就是生成一个数独终盘，生成终盘的算法有很多，像著名的分组转轮算法，这个算法可以高效地生成终盘，但是生成的数独终盘规律性太强，并不能做到完全的随机，还有一些类似的高效算法也有相同的缺点，不能完全随机看似不是什么问题，但是仅仅这个想法就能使我很不舒服，也正是因为这一点，我选择了复杂一点的方法（我知道，我有病），也就是回溯法，但是如果仅仅使用回溯法，时间复杂度将会很大，所以我们需要在回溯的同时加上推理的部分，说到这里，我们的思路就很明确了，我们将会采用与解数独相近的方法来生成数独终盘，差别在于这一次解的是没有填上任何数字的数独，也就是说，每一个格子中初始可能的数字都是1至9，一共9个，而且我们只需要随机的找到这个数独的一个解就可以了。

所以我们可以按照这个思路，加上一些随机的方法，写出下面这一段代码：

```
1.     def generateSolvedSudo(self):
2.         self.finalsolution = False
3.         possiblenumbers = [[[i for i in range(
   1,10)]for j in range(9) ]for k in range(9)]
4.         self.reGenerate(possiblenumbers)
5.         return [[str(self.finalsolution[i][j][
   0]) for j in range(9)] for i in range(9)]
6.
```

```
    def reGenerate(self,possiblenumbers):
        if all(len(possiblenumbers[i][j]) == 1 for (i,j) in self.grids):
            self.finalsolution = copy.deepcopy(possiblenumbers)
            return True
        length,x,y = random.choice([(len(possiblenumbers[i][j]),i,j) for (i,j) in self.grids if len(possiblenumbers[i][j]) == (min(len(possiblenumbers[x][y]) for (x,y) in self.grids if len(possiblenumbers[x][y]) > 1))])
        posibility = copy.copy(possiblenumbers[x][y])
        for i in posibility:
            copyindex = copy.deepcopy(possiblenumbers)
            if self.fill(x,y,i,copyindex) and self.finalsolution != True:
                if self.reGenerate(copyindex):
                    return True
        return False
```

如果我们把reGenerate函数和reSolve函数作对比的话，我们就可以看到他们在思路上的相近之处：

```
def reSlove(self,possiblenumbers,deepth):
        if all(len(possiblenumbers[i][j]) == 1 for (i,j) in self.grids):
            if not self.finalsolution:
                self.finalsolution = copy.deepcopy(possiblenumbers)
            else:
                self.finalsolution = True
            return
        length,x,y = min((len(possiblenumbers[i][j]),i,j) for (i,j) in self.grids if len(possiblenumbers[i][j]) > 1)
        posibility = copy.copy(possiblenumbers[x][y])
        for i in posibility:
            copyindex = copy.deepcopy(possiblenumbers)
            if self.fill(x,y,i,copyindex) and self.finalsolution != True:
```

```
13.                 self.reSlove(copyindex,deepth
    + 1)
14.         return
```

reSolve函数在选择填入的格子时并没有采用随机过程，因为解题的过程不需要随机，而reGenerate函数则不同，reGenerate 函数在选择尝试的格子时随机在可能数字个数大于1且可能个数最小的格子（们）中挑选了一个（原谅我蛋疼的表述，相信代码比我说得更清楚）。

如何高效地挖洞

现在我们已经有了母鸡，挖完洞就可以开始生蛋了，之前提到的的挖洞算法并不高效，因为它也是一个回溯算法，而且并不能通过我们之前提到过的推理来简化这一过程（因为并没有这样一种规则规定哪里能挖哪里不能挖），这意味着我们需要在改进算法和忍耐长时间的回溯尝试之间选择一个，这个选择的答案很明显：我们需要改进这一算法。

有必要用回溯么?

一旦我们开始仔细思考，我们就会发现其实在这个问题上运用回溯其实并没有必要，因为如果我们去掉一个位置的数A后发现这个数独谜题有多解的话，即使我们之后去掉一个其他的数之后，还是不能去掉这个A，因为这样的情况还是会造成多解，这种情况出现的道理也很简单：

> 假如一个数独谜题有多解，那么去掉一个数字之后，这个数独谜题还是有多解

看来冥冥之中存在一种算法，似乎可以做到用线性的时间复杂度挖洞，我们要做的只是找出这种算法罢了。

线性复杂度挖洞算法，诞生

在确定有线性复杂度算法之后，我们就开始寻找这种算法，我们设想这样一种算法:

首先，将81个格子按照某种顺序（一般是某种随机顺序）遍历

去掉被遍历的格子上的数字；

如果此时的数独还有唯一解，那么继续遍历下一个格子，否则恢复这个格子上的数字，然后继续遍历下一个格子，这种算法我们也可以在一些其他文章中发现有提及，比如这篇文章（当然，我是单独发现后才知道其他人做过这些研究），根据上面说出的这些东西，我们终于可以写出挖洞函数了：

```
#The most simple way to generate a puzzle
    def generateUnsolvedPuzzle(self):
        solvedPuzzle = self.generateSolvedSudo
    ()
        ran = copy.deepcopy(self.grids)
        random.shuffle(ran)
        for (i,j) in ran:
            buffer = solvedPuzzle[i][j]
            solvedPuzzle[i][j] = '.'
            if self.solveSudoku(solvedPuzzle)
    == True:
                solvedPuzzle[i][j] = buffer
        return [''.join(i) for i in solvedPuzz
    le]
```

开始下蛋

挖好了坑，就可以开始下蛋了，我们先下几个蛋（产生几个谜题），然后看看这些蛋能不能孵出小鸡（是否有唯一解）：

```
. 7 . | . . 6 | . . .
2 . . | . . . | 3 5 .
. . . | 5 7 1 | 2 . .
-------------------------
5 1 2 | . . . | . . .
. 8 . | . 3 . | . 9 .
. . . | . 8 . | . . 6
-------------------------
6 . . | . . 2 | 5 4 .
9 5 . | . . . | . . .
. . . | 9 . . | . . .
```

```
4 7 5 |3 2 6 |9 8 1
2 6 1 |8 4 9 |3 5 7
8 9 3 |5 7 1 |2 6 4
------------------------
5 1 2 |6 9 7 |4 3 8
7 8 6 |2 3 4 |1 9 5
3 4 9 |1 8 5 |7 2 6
------------------------
6 3 8 |7 1 2 |5 4 9
9 5 7 |4 6 3 |8 1 2
1 2 4 |9 5 8 |6 7 3
```

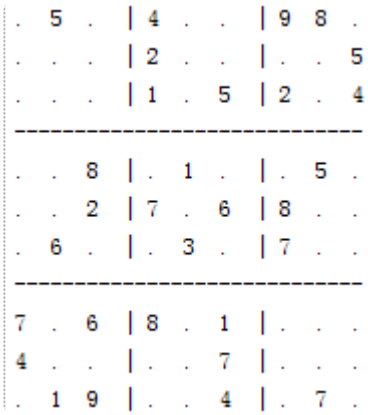
看来这个下蛋的方法是可用的，但是我们平时见到的很多数独却并不是这样的，我们平时见到的数独大都中心对称或者轴对称，有一种说不出来的数学美，但是我们却很难从上面这个数独里看出这种数学美。

生成中心对成的数独

我们生成的数独居然不好看，这简直不能忍！于是我决定重写这个生成方法，强行也要生成对称数独，我的算法很简单，成对地去掉终盘上的数字，直到不能去掉任何一对为止。算法这么简单，代码肯定也长不了：

```
1. def generateCentralSymmetryPuzzle(self):
2.         solvedPuzzle = self.generateSolvedSudo
   ()
3.         ran = copy.deepcopy([self.grids[i] for
   i in range(len(self.grids) / 2 + 1)])
4.         random.shuffle(ran)
5.         for (i,j) in ran:
6.             buffer1 = solvedPuzzle[i][j]
7.             buffer2 = solvedPuzzle[8 - i][8 -
    j]
8.             solvedPuzzle[i][j] = '.'
9.             solvedPuzzle[8 - i][8 - j] =  '.'
10.            if self.solveSudoku(solvedPuzzle)
   == True:
11.                solvedPuzzle[i][j] = buffer1
12.                solvedPuzzle[8 - i][8 - j] = b
   uffer2
13.        return [''.join(i) for i in solvedPuzz
   le]
```

代码出来了，生成一个数独玩玩？必须的：



看，这数独多么像银河，简直美爆了有没有，我们还可以利用类似的算法生成轴对称的数独，甚至私人订制自己想要的数独，至于生成速度，，，额，有点慢，2到3秒才能出一个数独题面，统计数据如果将来我统计出来的话就在这个位置补上！

附录：

1 所有的代码，包括数独生成和解数独代码

（ps：偶尔搜到这种代码可能有商用价值，但是我对我的代码还是很有信心的，这东西绝逼没有商用价值）：

```
1. import copy
2. import random
3. class Sudoku:
4.     # @param board, a 9x9 2D array
5.     # Solve the Sudoku by modifying the input
    board in-place.
6.     # Do not return any value.
7.     def __init__(self):
8.         self.grids = [(i,j) for j in range(9)
   for i in range(9)]
9.         self.peercross = [[[[(j,i) for i in ra
   nge(9)],[(i,k) for i in range(9)],[(j - j % 3
   + i / 3,k - k % 3 + i % 3)for i in range(9)]]
   for k in range(9)]for j in range(9)]
```

```
        self.peers = [[set(sum(self.peercross[j][k],[])) - set([(j,k)]) for k in range(9)] for j in range(9)]
        self.finalsolution = False

    def solveSudoku(self, board):
        self.finalsolution = False
        possiblenumbers = [[[i for i in range(1,10)]for j in range(9) ]for k in range(9)]
        assert (len(board) == 9 and all((len(board[i]) == 9) for i in range(9))) == True
        if not self.preProcess(board,possiblenumbers):return False
        self.reSlove(possiblenumbers,1)
        return self.finalsolution

    def preProcess(self,board,possiblenumbers):
        for i in range(81):
            if not self.fill(i / 9,i % 9,board[i / 9][i % 9],possiblenumbers):
                return False
        return True

    #@return whether the value can be filled in the current grid
    def fill(self,x,y,value,possiblenumbers):
        if value not in range(1,10) and value not in "123456789":return True
        if int(value) not in possiblenumbers[x][y] or x not in range(9) or y not in range(9):return False
        if len(possiblenumbers[x][y]) == 1 or all([self.eliminate(x,y,i,possiblenumbers) for i in  [j for j in possiblenumbers[x][y] if j != int(value)]]):
            return True
        return False

    #@return whether elimatesuccess or not
    def eliminate(self,x,y,value,possiblenumbers):
        if value not in possiblenumbers[x][y]:
            return True
        if len(possiblenumbers[x][y]) == 1:
```

```
40.             return False
41.         possiblenumbers[x][y].remove(value)
42.         #stg1
43.         if len(possiblenumbers[x][y]) == 0 or
     len(possiblenumbers[x][y]) == 1 and  not all
    ([self.eliminate(i,j,possiblenumbers[x][y][0],
    possiblenumbers) for (i,j) in self.peers[x][y]
    if (possiblenumbers[x][y][0]) in possiblenumbe
    rs[i][j]]):
44.              return False
45.         #stg2
46.         for k in self.peercross[x][y]:
47.             tem = [(i,j) for (i,j) in k if val
    ue in possiblenumbers[i][j]]
48.             if len(tem) == 1 and len(possiblen
    umbers[tem[0][0]][tem[0][1]]) > 1 and  not sel
    f.fill(tem[0][0],tem[0][1],value,possiblenumbe
    rs):
49.                 return False
50.         return True
51.
52.     def reSlove(self,possiblenumbers,deepth):
53.         if all(len(possiblenumbers[i][j]) == 1
    for (i,j) in self.grids):
54.             if not self.finalsolution:
55.                 self.finalsolution = copy.deep
    copy(possiblenumbers)
56.             else:
57.                 self.finalsolution = True
58.             return
59.         length,x,y = min((len(possiblenumbers[
    i][j]),i,j) for (i,j) in self.grids if len(pos
    siblenumbers[i][j]) > 1)
60.         posibility = copy.copy(possiblenumbers
    [x][y])
61.         for i in posibility:
62.             copyindex = copy.deepcopy(possible
    numbers)
63.             if self.fill(x,y,i,copyindex) and
     self.finalsolution != True:
64.                 self.reSlove(copyindex,deepth
    + 1)
65.         return
66.
67.     def generateSolvedSudo(self):
```

```
68.         self.finalsolution = False
69.         possiblenumbers = [[[i for i in range(
    1,10)]for j in range(9) ]for k in range(9)]
70.         self.reGenerate(possiblenumbers)
71.         return [[str(self.finalsolution[i][j][
    0]) for j in range(9)] for i in range(9)]
72.
73.     def reGenerate(self,possiblenumbers):
74.          if all(len(possiblenumbers[i][j]) ==
    1 for (i,j) in self.grids):
75.             self.finalsolution = copy.deepcopy
    (possiblenumbers)
76.             return True
77.         length,x,y = random.choice([(len(poss
    iblenumbers[i][j]),i,j) for (i,j) in self.grid
    s if len(possiblenumbers[i][j]) == (min(len(po
    ssiblenumbers[x][y]) for (x,y) in self.grids i
    f len(possiblenumbers[x][y]) > 1))])
78.         posibility = copy.copy(possiblenumber
    s[x][y])
79.         for i in posibility:
80.             copyindex = copy.deepcopy(possible
    numbers)
81.             if self.fill(x,y,i,copyindex) and
     self.finalsolution != True:
82.                 if self.reGenerate(copyindex):
83.                     return True
84.         return False
85.
86.     #The most simple way to generate a puzzle
87.     def generateUnsolvedPuzzle(self):
88.         solvedPuzzle = self.generateSolvedSudo
    ()
89.         ran = copy.deepcopy(self.grids)
90.         random.shuffle(ran)
91.         for (i,j) in ran:
92.             buffer = solvedPuzzle[i][j]
93.             solvedPuzzle[i][j] = '.'
94.             if self.solveSudoku(solvedPuzzle)
    == True:
95.                 solvedPuzzle[i][j] = buffer
96.         return [''.join(i) for i in solvedPuzz
    le]
97.
```

```
    #a little complicated way to generate a sy
metry sudoky puzzle
    def generateCentralSymmetryPuzzle(self):
        solvedPuzzle = self.generateSolvedSudo
()
        ran = copy.deepcopy([self.grids[i] for
i in range(len(self.grids) / 2 + 1)])
        random.shuffle(ran)
        for (i,j) in ran:
            buffer1 = solvedPuzzle[i][j]
            buffer2 = solvedPuzzle[8 - i][8 -
 j]
            solvedPuzzle[i][j] = '.'
            solvedPuzzle[8 - i][8 - j] =  '.'
            if self.solveSudoku(solvedPuzzle)
== True:
                solvedPuzzle[i][j] = buffer1
                solvedPuzzle[8 - i][8 - j] = b
uffer2
        return [''.join(i) for i in solvedPuzz
le]

    #show the puzzle
    def show(self,puzzle):
        for i in range(len(puzzle)):
            for j in  range(len(puzzle[i])):
                if puzzle[i][j] is chr:
                    print puzzle[i][j],'',
                else:
                    print puzzle[i][j][0],'',
                if j == 2 or j == 5:
                    print '|',
            print
            if i == 2 or i == 5:
                print '-' * 29
        print

s = Sudoku()
print '------------answer-----------'
#print  s.solveSudoku(['.7.8....2', '.6.9...
3.', '.......47', '....1..89', '.8.4.....',
 '.....2...', '7.93.....', '1...75.9.',
 '3.......1'])
puzzle = s.generateCentralSymmetryPuzzle()
s.show(puzzle)
```

```
133. s.show(s.solveSudoku(puzzle))
```

2 我为什么要写这篇文章

我写这篇文章的动机其实是看到了一道leetcode上的题Sudoku Solver，这道题要我们写一个程序解出一个有唯一解的数独，但是我写的python算法总是不能再规定时间内完成解题，所以我开始搜索一些良好的解题方法，就是这时我看到了 Peter Norving 的那篇 Solving Every Sudoku Puzzle 我非常喜欢peter 的算法，并在他的算法的基础上做了改进，使得这个算法不单单能够解出数独，还能够判断一个数独是否有解，和是否有多个解。当我完成这一部分的编码时，我突然想到了之前看过的一本数独杂志上提到过一种类似挖洞的算法能够用上我目前的编码，于是我google了这种算法，并根据自己的理解将挖洞算法改造成了一种线性时间复杂度的算法（后来我才知道这个算法其实已经有人提到过），终于，经过两天的努力，终于诞生了上面那个百来行的python代码，和这篇文章，我的这份代码并非完美，有很多我自己都觉得看不过去的地方，而且我在这个程序里用的数据结构并不统一，这也导致我需要进行一些蛋疼的转换，但这份代码能完成应该完成的工作，所以我也乐意将它分享出来，也希望其他人能够在我的基础上对我的代码进行重写或改进，如果这篇文章能让一些人觉得豁然开朗，那我的努力就没有白费。

3 其他说明

如果转载本文章请注明出处

您可以无偿地使用这篇文章中的代码，而无需声明作者