数独问题求解程序解题报告

陈贝宁 2100011461

1. **问题描述与需求分析**

**题目要求简述**

设计并实现一个数独问题求解程序。程序可从文本文件读取数独问题，并在适当的时间内输出该问题的解。

**问题分析**

题目要求实现一个数独问题求解程序，由于数独问题有着明确的规则以及唯一的解，所以我们只需要找到一个符合规则的解，而不需要考虑这个解的唯一性。

**设计思路**

采用最朴素的解数独的技巧，即找出每一个格子里的所有可能的数字，并采用枚举加搜索的技巧实现解的寻找。

解的读入与输出都以文本格式进行。

**二、设计（数据结构与算法）**

**整体框架的搭建**

我们将所有需要反复用到的函数进行了封装，方便整体规划与程序阅读。大致分为以下几种函数。

1. 读入函数：readin()
2. 寻找可能数字的函数：possible()
3. 获取可能数字表的函数：get\_numtable()
4. 寻找最少可能数字的函数：find\_least()
5. 打印最终答案的函数：print\_puzzle()
6. 主函数：main()

其中主函数包含了由枚举加搜索构成的主循环，函数内引用了所有其它的函数，是解决问题的关键所在。

下面来一个一个地完成这些函数的代码。

**三、具体程序实现**

**1）读入函数：**

读入函数应当从一个文本文档中读取一个数独应当包含的信息，并将这些信息存储到一个9\*9的二维列表里。

由于数独中包含空格位，这里特殊规定所有空格位都以0作为替代。为了读取方便，应该规定每个数字之间由空格隔开，每行9个数字之间有回车键隔开。因此，标准的数独文本输入应该长这样：

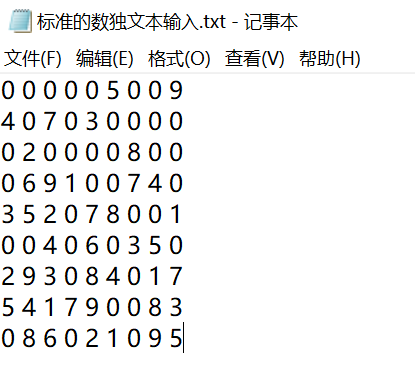


图1

此时的读入函数可以利用python自带的split方法，将一行字符串拆分为一个一个的字符。再利用python自带的int()函数将每个字符转换为数字，存入预先定义的二维列表（取名为puzzle）中并返回该列表。

代码如下：

def readin():

# 读入原始数据

# 分行读入，以空格为界，0表示无数字

# 返回9\*9列表puzzle

puzzle = []

for i in range(9):

p = [int(x) for x in input().split()]

puzzle.append(p)

return puzzle

**2）寻找可能数字的函数：**

寻找可能数字的函数应该在给定的puzzle列表中找出给定位置（由行数i、列数j确定）的所有可能数字并存在特定列表中返回。

数独的基本规则规定，1-9每个数字在每一宫、每一行和每一列中都只出现一次。因此，只需要在宫内、行内、列内依次寻找数字即可。我们可以采用排除法，先记录所有在以上范围内出现过的数字，之后再从数字0-9中排除即可。（这里的0代表空格，不用担心，我们无论如何都会把它排除的）而由于集合set()内的元素不会重复，我们可以利用这一特性来帮助记录出现过的数字。

还有重要的一点没有提到。在之后的主函数中，由于搜索的性质，势必要将之前已经走过的路径排除。这里需要再引入一个二维列表mark，用于标记已经在某个格子尝试过的数字，避免一些死循环。具体的实现方法我们会在主函数中仔细讨论，现在只需要知道它的作用即可。

代码如下：

def possible(i,j,puzzle,mark):

# 对于给定数据表，找出（i，j）可能的数字

# （i，j）位于第（m，n）个宫内

m = i//3 ; n = j//3

s = set() # 集合

# mark表的使用

for x in mark[i][j]:

s.add(x)

# 宫内

for pi in range(3\*m,3\*m+3):

for pj in range(3\*n,3\*n+3):

s.add(puzzle[pi][pj])

# 行内

for pj in range(9):

s.add(puzzle[i][pj])

# 列内

for pi in range(9):

s.add(puzzle[pi][j])

# 挨个排除

p = [x for x in range(10)] # 数字0-9

for x in s:

p.remove(x)

return p

**3）获取可能数字表的函数：**

获取可能数字表的函数的作用，单纯只是获取一个二维列表（命名为numtable），其元素为该位置（i，j）对应的puzzle元素的所有可能数字组成的列表。举个例子，如图1，第二行第二列对应的元素0（在python中，由于众所周知的原因，其坐标应该是（1,1）），只有一个可能的数字1，故numtable[1][1]为列表[1]；而第三行第一列有三个可能数字，分别为1/6/9，故numtable[2][0]为列表[1,6,9]；另外，像第二行第一列这种已经填入数字的位置，自然有numtable[1][0]为空列表[]。

这个函数看似十分复杂，其实实现方式十分简单，因为我们所需的主要解决方法已经在2）中给出，我们只需要反复调用它即可。

代码如下：

def get\_numtable(puzzle,mark):

# 获得一张记录可能数字的表

numtable = [[[]for \_ in range(9)]for \_ in range(9)]

for i in range(9):

for j in range(9):

if puzzle[i][j] == 0: # 判断是否为空格子

numtable[i][j] = possible(i, j, puzzle, mark)

return numtable

**4）寻找最少可能数字的函数：**

寻找最少可能数字的函数的作用顾名思义，需要在给定的numtable中寻找包含的可能数字最少的一格，并返回这个坐标。这里采用最朴素的搜索思想，即按行列顺序挨个寻找并记录最小长度。

由3）的讨论我们得知，在已经有数字填入的情况下，numtable会给出一个空列表，即长度为0。这当然是最小的长度，但并不是我们想要寻找的。因此我们需要增加一步判断，如果该位置没有填入任何数字，即puzzle[i][j]==0，我们才能进行下一步的比较。

至于这里为什么要引入puzzle，而不是判断numtable[i][j]是否为空列表，我们可以从全局观看待这一问题。除去特别简单的数独，大部分数独都需要一些试错过程，而如果我们误入歧途，填入了错误的数字，我们就需要一个机制来帮助我们判断正确与否，从而可以原路返回。寻找最少可能数字的函数就是这样一种函数。如果它返回的坐标在numtable中是一个空列表，那就说明在这个位置已经无法填入任何合理的数字了。而如果直接判断numtable[i][j]是否为空列表的话，我们就失去了这一保险，那么最后输出的结果就会错误。

代码如下：

def find\_least(puzzle,numtable):

# 寻找给定数据表中可能数字最少的一格

mini = 9 ; p = None # （1）

for i in range(9):

for j in range(9):

if puzzle[i][j] == 0 and len(numtable[i][j]) < mini:

mini = len(numtable[i][j])

p = [i,j]

return p

1. 注意这里我们把p的初始值设定为None，是为了方便判断数独有没有被填完。如果被填完了，那么接下来的双重循环中，所有位置都会被if中第一个判断给排除，从而函数返回了一个None。我们只需判断其返回值是不是None即可判断数独有没有被填完。

**5）打印最终答案的函数：**

利用python自带的join函数即可，代码如下：

def print\_puzzle(puzzle):

# 打印

for i in range(9):

print(' '.join([str(x) for x in puzzle[i]]))

**6）主函数：**

主函数的主体是一个循环，用于模拟搜索过程。这是一个典型的深度优先搜索算法，但是由于一些历史原因和个人因素，我们采用循环来模拟搜索过程。而栈作为一种“先进后出”的数据类型，极其适合用来模拟深搜算法的路线进退。

每次循环开始时，都要对numtable进行一次更新，然后利用find\_least()寻找这一步应该到达的坐标，在numtable中寻找对应的可能数字，随便选一个（这里选的是第一个）然后重复循环，直到find\_least()返回None为止，就是说全部填完了，打印即可。

以下是循环部分的伪代码：

while True:

numtable = get\_numtable(puzzle, mark)

p = find\_least(puzzle, numtable)

if p:

# 还没填完

i,j = p

if numtable[i][j]:

# 存在可用数字，填入，进栈

else:

# 走入歧途，退栈

else:

# 填完了，打印

print\_puzzle(puzzle)

break

对于存在可用数字的情况，我们需要填入一个选好的数字，然后把它放进mark[i][j]中，表示这个数字已经被用过了。最后把这个操作放到栈中。

这里有两个问题，一个是mark的实际用途，另一个是栈的元素的规定。先解决后者：将栈的元素规定为（i，j，k），其中i，j分别为操作所在的行列坐标，k为填入的数字。

现在考虑mark的用途。典型的深搜算法会把已经走过的路线做标记，而这里并没有“路线”一说，因为每次循环都会重新寻找一个新的坐标，与之前的坐标其实没有直接的联系。在这种情况下，mark的记录并没有影响，和填数字一起即可；而mark的回退时，则需要把mark[i][j]的值初始化，否则就会丢失一些信息。而对于mark的初始化问题，其实0作为人为引入的标记，应该尽力避免出现在搜索过程中，因此我们让mark[i][j]初始值为[0]，这样我们就可以保证在2）中的函数不会返回包含0的列表了。

再者就是关于计时问题。Python中可以import一个time模块，而time.time()可以返回当前时间的浮点值。我们可以在主循环外增加两个time.time()作为始末的时间，两者相减就是整个函数运行的时间。

最终主函数代码如下：

def main():

# 主程序

# 用来记录操作的栈，元素（i，j，填入的数字k）

stack = []

# 用来记录被排除的数字

mark = [[[0]for \_ in range(9)]for \_ in range(9)]

# 读入数独

puzzle = readin()

# 计时模组

s = time.time()

while True:

numtable = get\_numtable(puzzle, mark)

p = find\_least(puzzle, numtable)

if p:

# 还没填完

i,j = p

if numtable[i][j]:

# 存在可用数字，填入，进栈

k = numtable[i][j][0]

mark[i][j].append(k) # 标记填过的数字

puzzle[i][j] = k

stack.append((i,j,k))

else:

# 走入歧途，退栈

mark[i][j] = [0] # 回退mark

i,j,k = stack.pop()

puzzle[i][j] = 0

else:

# 填完了，打印

print\_puzzle(puzzle)

break

e = time.time()

print('用时：' + str(e-s) + '秒')

if \_\_name\_\_ == '\_\_main\_\_':

main()

最后一句是对于主函数的调用。

**四、算法与数据结构的分析与评价**

为了进一步认识评估自己的程序效率，我们对其中几个主要部分进行了细致的测试分析与评价。

**1）读入与输出函数**

并没有什么特殊的地方，对于一个普通的二维列表，读入与输出函数的时间复杂度为o(n^2)。（这里及以下情况下n=9）

**2）寻找可能数字的函数**

对于某一个特定的位置坐标，我们需要遍历它的宫、行、列中（包括自己）共(n-2\*(n-n^0.5) = 21)个，时间复杂度为o(n)

**3）获得可能数字表的函数**

这个函数调用了n^2次上个函数，时间复杂度为o(n^3)

**4）寻找最少可能数字的函数**

这个函数遍历了整个二维列表，时间复杂度为o(n^2)

**5）主函数**

循环前的数据预处理包括：建立栈（o(1)），建立mark表（o(n^2)），读入数独（o(n^2)），这三者存在加法关系。

每次循环中包括：更新numtable表（o(n^3)），寻找坐标（o(n^2)），进行判断（o(1)），这三者存在加法关系，故单次循环的时间复杂度为o(n^3)

至于循环的次数，最好情况为o(n^2-m)，m表示原数独的已填充格数，表示每次选取的数字恰好都是正确的数字；至于最坏情况和平均情况，我们并没有足够的能力从理论上解决，姑且定义为o(n^k)，其中k>=2,。

所以解决一个数独的总时间复杂度为o(n^k)\*o(n^3) + o(n^2) = o(n^(k+3)) >= o(n^5)。考虑到n=9，解决的时间应不会太长。

而对于空间复杂度，我们所用到的数据结构有：二维列表o(n^2)，栈（最多o(n^2)）。

**具体测试分析与算法优化**

我们从网站（https://cn.sudokupuzzle.org/）中选取了5种不同难度共25例数独（3.11-3.15日每日一题），分别运行得到答案并记录了所有的用时。经对比，所有答案均与网站中给出的答案一致。下表给出了所有测试用例的解答用时：



表1 数独测试用例用时记录表（单位：秒）

从表中可以看出，解决一个数独的平均用时在10秒左右，且实际用时与（网站标注的）难度有较大出入。

我们注意到，大部分例子的解决用时都在1秒内，而有小部分的例子用时在10秒以上，甚至达到了100秒。造成这种现象的原因是深度优先搜索的特性所致。例如3.12日的中级题：

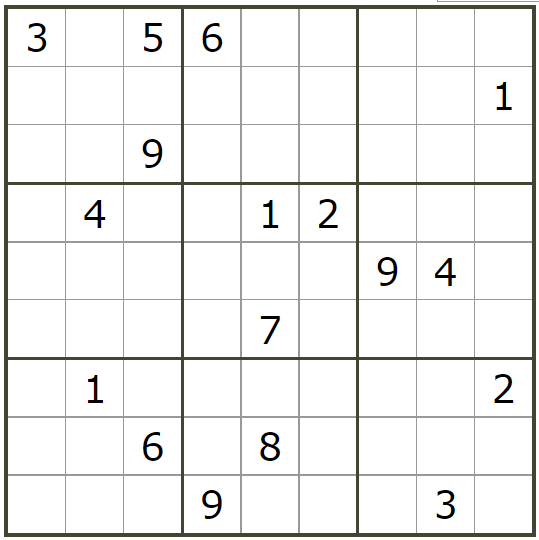


图2 3.12中级题

我们可以写出它的可能数字表：

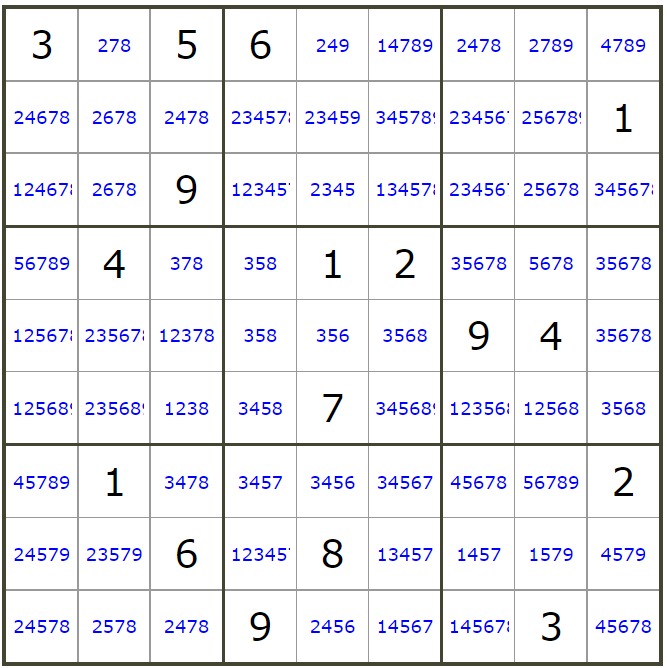


图3 3.12中级题的可能数字表

跟着程序的想法，第一次操作的位置坐标是(0,1)，即3和5中间的空格，并选取2作为预选数字填入（其实正确数字是8）。之后会在其它的格子反复试验（但是都是错的），而且由于mark表会自动回退，所以总共要试验约4^30次（经测试栈深最大可达30层）。故此题的求解时间会很长。

这种情况一般出现在初级到中级之间，因为入门级并不会出现填入错误数字的情况；而高级和骨灰级则会在较浅的栈深时回退。

对于这种情况，我们可以采取适当的数据预处理工作。

我们注意到，如果一个宫内的numtable只有一格有唯一的数，比如说图3中的第一宫，只有左下格有1，那么左下格就必须填1。这样我们就能预先排除一些容易判断的格子。

如下定义一个找必须填的数字的函数：

def find\_must(numtable):

# 寻找必须填的数字

l = [None] ; pk = None

for k in range(1,10):

# 宫内

for m in range(3):

for n in range(3):

# 第（m，n）宫

f = True

for a in range(9):

i = a//3 + 3\*m ; j = a%3 + 3\*n

if k in numtable[i][j]:

# 找到了

if f:

# 只有一个

pk = (i,j,k) ; f = False

else:

# 不止一个

pk = None

if pk not in l:

l.append(pk)

# 行内

for i in range(9):

# 第i行

f = True

for j in range(9):

if k in numtable[i][j]:

# 找到了

if f:

# 只有一个

pk = (i,j,k) ; f = False

else:

# 不止一个

pk = None

if pk not in l:

l.append(pk)

# 列内类似，由于空间原因不再赘述

return l[1:]

并在主函数的主循环前插入如下循环：

# 先找必须填的

numtable = get\_numtable(puzzle, mark)

l = find\_must(numtable)

# print(l)

while l:

p = l.pop()

# 存在，填入，进栈（其实无所谓）

i,j,k = p

puzzle[i][j] = k

stack.append(p)

如此即可实现程序的优化。下表给出了优化后同样测试用例的用时：



表2 优化后用时记录表（单位：秒）

可见优化后能节省近一半的时间。

**五、上机实习总结**