

# AIを利用した ナンプレの問題自動生成

進化計算（山登り法）による

ゼロからの進化計算

Python版

株式会社タイムインターメディア

知識工学センター

UECアライアンスセンター

はじめに

第 1 部 問題を解く

第 2 部 問題を作る

第 3 部 発展課題編

# はじめに

ナンプレ（ナンバープレース）というパズルが世界的に流行している。海外では、SUDOKUと呼ばれることが多い。

このパズルは、右図の様に、9x9のマスが、太さの違う罫線で描かれていて、マスの一部に数字が入っています。

この空白マスに、以下のルールにしたがって各マスに1～9の数字を入れます。

**タテ9マス、よこ9マス、太線で囲まれた9マス（ブロックと呼ぶ）の中には、1から9までの数字がそれぞれ1つずつ入ります。**

この問題を、ルールに従って解くと右図のように全マスが埋まります。どのタテ列、どの横列、どのブロックを見ても、1から9までの数字が1つずつになっています。

書店では、この問題集が多数販売されていて、ネット上でも無料で遊べるサイトが多数あります。さらに、問題が難しくて解けないときには、問題を入力すると、解を教えてくれる親切なサイトもあります。

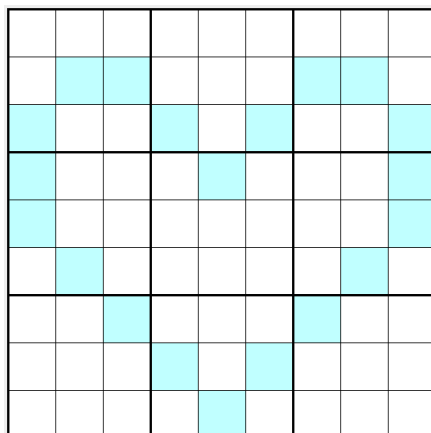
	7	1				9	3	
3			8		2			5
1				7				3
4								8
	2						4	
		6				1		
			3		4			
				1				

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

さて、この解説では、問題を解くプログラムの作成ではなく、さらにその先の問題を作

る（作問）プログラムの説明をします。問題を何とか作れる程度ではなく、パズル作家が手および頭を使っただけでは作成が困難か、非常に時間がかかってしまうレベルの問題を、普通のパソコンでラクラクと作れる方法を説明します。それも、非常に簡単なアルゴリズムで。

右図のように最初から見える数字（ヒント）の配置を与えて、これらのマスの数字を適当に決めることで、解が1つだけ（ユニーク解）の問題（最初の図）を自動生成する方法を説明します。



ただ、適当に数字を並べて入れ替える程度では、延々と入れ替えを繰り返してもユニーク解にはなりません。

人工知能技術の1つに進化計算という方法があり、これは生物の進化にならって、少しずつ良くする（進化する）ことで最終的な目的を達成しようという方法です。ナンプレの自動生成の場合には、解がただ1つ（ユニーク解）のちゃんとした問題を完成させます。

人工知能技術というと高度で無理と思うかもしれませんが、とても簡単な方法しか使わないので、プログラミングを始めたばかりの方でも十分理解できると思います。プログラムもそんなに長くなく、全体を把握しやすくなっています。

人工知能の中の進化計算の中で最もやさしい手法である山登り法を用いて説明します。山登り法は簡単な問題にしか適用できない方法ですが、ここではそれをさらに簡単化した方法を紹介しています。9x9の標準的なナンプレ問題の自動作成なら、これでも楽勝です。

プログラミング言語はPythonを用いています。バージョンは3です。

Python以外の言語でも大丈夫ですが、現在Pythonが広く普及していて、Pythonなら分かるという人が多いため、オリジナルのJava版をPython版に書き換えたものを用意しました。

オリジナルがJavaになっているのは、2019年2月に、タイのタマサート大学シリントーン国際工学部の学生を日本に招待してインターンシップ実習をしたとき、タマサート大学での標準言語がJavaになっていたため、Javaのプログラムを用意し、そのプログラムを少し変更したのが、公開しているJava版のプログラムです。

Javaプログラムは、インターンシップ実施時には、Raspberry Pi3で動かしていたものです。Javaの開発環境であるEclipseも使っていませんでした。

実は、Python版よりもJava版の方が非常に高速です。Javaでなくても、コンパイル言語と言われている、C, C++, C#, Rustなどで書けば、Python版よりも遥かに高速になります。しかし、コンパイル言語は、Pythonと比較すると習得が困難なため、まずはプログラムの内容、アルゴリズムを理解してもらうためにPython版を用意しました。

Pythonのプログラムは自由に改変してご利用ください。実際に問題を便利に作ったり、できた問題を画面上で遊ぶためにはGUIも必要でしょう。でも、本説明はあくまで自動生成はこうすれば簡単にできるという一例を示すのが目的ですので、本質的以外の部分は全部省略しています。

Pythonのコーディングも、模範的でない箇所も多々あります。また、ファイル入力の扱いなど、柔軟性を欠いているところもありますが、それらも読者への課題として残したままにしています。

本格的な進化計算を行う場合は、山登り法ではなく遺伝的アルゴリズムを使うことが多くなります。また、計算量が増えるため、そして高性能になったCPUを有効利用するために、高速化にはマルチプロセス・マルチスレッドによる並列処理を使うのが一般的です。さらに、現実の最適化問題は、複数の相反する項目の調整も必要になることが多く、多目的遺伝的アルゴリズムが使われるようになります。

本説明書で、進化計算の第一歩を経験していただき、そしてさらに先に進んでいかれることを期待しています。

そして、近々、高速なPythonと言われているCython版も公開します。Pythonのプログラムに若干の修正、C言語の型宣言などを加えるだけで100倍ほど高速になるのを実感していただきたいと思います。

# 第 1 部 問題を解く

ナンプレの問題を作るのですが、その前に、ナンプレの問題を解くプログラムを完成させる必要があります。問題を解くプログラムのことをソルバーと呼び、問題を作るプログラムをジェネレータと呼びます。

問題の自動生成をするとき、ジェネレータは頻繁にソルバーを呼び出します。普通は、市販の問題集とか、ネット上に公開されている問題などを解くことが多いので、解が一意に決まる問題を解くソルバーを作ることが多いと思います。

しかし、問題を作る場合には、未完成の色々トラブルのある問題を解く必要があります、どのくらいトラブルがある問題かを調べます。そのため、不完全な問題、矛盾している問題を解けるところまで解くプログラムが必要になります。

## 実行環境

LinuxのUbuntu 20で実行しており、Intelのi5-7500 CPU @ 3.40GHzで確認しています。

では、実際にプログラムを見ていきましょう。  
プログラムは、自動生成も含め全体で720行になっています。

```
237 NP.py
168 generator.py
   9 parameter.py
   62 solution.py
244 solver.py
720 合計
```

NP.pyがメインプログラムで、問題を解くときは問題ファイル、問題を作成するときはパターンファイルを読み込みます。その後、実際に問題を1問解くのがsolver.pyで、1つのパターンに対して問題を自動生成するのがgenerator.pyです。

Parameter.py は非常に短く、問題のサイズ(SIZE)とブロックのサイズ (SUBSIZE) をそれぞれ9と3に指定しているだけです。

各ソースファイルの先頭には簡単なヘッダがあり、MITライセンスのオープンソースとして提供していることが書かれています。これらのコメント部分を除くと、全体で700行に満たない短いプログラムです。

solverの方が、generatorより行数が多くなっています。このプログラムでは、解くための様々なテクニックを組み込んでいないので、solverはかなり簡単になっていますが、それでもgeneratorよりも長くなっています。様々な高度なナンプレを解くテクニックを組み込もうとすると、solverがどんどん長くなっていきます。

問題の自動生成generatorは人工知能(AI)を使っていて、そのアルゴリズムの理解は非常に難しいという印象が世間にはあるようですが、実際にはsolverの方が面倒です。さらに、generatorは、別のパズルの自動生成をする場合もほとんどそのまま使えますが、solverはパズルが異なれば、パズルに合わせて作り直す必要があります。

まず、問題ファイルを用意します。

問題ファイルは、問題のタイトル行と、それにつづく問題を図のように用意します。

1～9のマスは、そのまま数字が入り、数字のないマスには-(マイナス)が入っています。また、カラムとカラムの間、横方向には間にスペース1文字が入っています。

複数の問題が1つのファイルに入っている場合、問題間に行は開けず、タイトル行、問題、タイトル行、問題を繰り返します。

```
Heart      H 20
- - - - -
- 6 2 - - - 5 4 -
7 - - 8 - 9 - - 1
1 - - - 9 - - - 3
9 - - - - - - - 7
- 7 - - - - - 5 -
- - 5 - - - 2 - -
- - - 7 - 1 - - -
- - - - 6 - - - -
```

タイトル行の内容は任意ですが、適当な名前（問題番号）とヒント数を示すようにしています。

ナンプレの問題を解くのと作るのは同じプログラム NP.py にまとめられていて、引数のオプションで切り替えています。

ファイル構成は、Pythonというフォルダに全てのPythonのプログラム \*.py が存在し、その下のdataの中に、問題あるいはヒントパターンのファイルが存在する場合について説明します。

とりあえず、NP.py を実行すると、以下のようになります。

```
Python$ python3 NP.py
===== arguments input error =====
python3 NP.py -s problem_file [answer_file]
python3 NP.py -g pattern_file [problem_file]
```

## 問題を解く

dataフォルダ内のHeartQ.txtを使って、問題を解いてみましょう。



```

Python$ cat data/HeartQ.txt
Heart    H 20
- - - - -
- 1 6 - - - 5 9 -
4 - - 3 - 7 - - 2
7 - - - 5 - - - 1
8 - - - - - - 9
- 6 - - - - 7 -
- - 5 - - - 6 - -
- - - 2 - 3 - - -
- - - - 7 - - - -

Python$ python3 NP.py -s data/HeartQ.txt
Heart    H 20
- - - - -
- 1 6 - - - 5 9 -
4 - - 3 - 7 - - 2
7 - - - 5 - - - 1
8 - - - - - - 9
- 6 - - - - 7 -
- - 5 - - - 6 - -
- - - 2 - 3 - - -
- - - - 7 - - - -

0
2 8 7 5 1 9 4 3 6
3 1 6 4 2 8 5 9 7
4 5 9 3 6 7 1 8 2
7 9 4 8 5 2 3 6 1
8 3 1 7 4 6 2 5 9
5 6 2 9 3 1 8 7 4
9 7 5 1 8 4 6 2 3
6 4 8 2 9 3 7 1 5
1 2 3 6 7 5 9 4 8

Total 1      Success 1
Time      0.021847 sec

```

NP.py を実行すると、まず、読み取った問題をそのまま表示しています。

その直後に0だけの行があります。

この行は、問題の不都合度を示し、0は、ユニーク解だったことを示しています。従って、上記のハートは、正しい問題だったことが分かります。

その次に、解を示しています。

問題、解のいずれも、数字、空きマスだけを示し、罫線は省略しています。

ファイル中の全問題を解いた後に、トータルの問題数と成功数が示されます。

最後に、全体の時間を秒単位で示しています。

添付のサンプル問題500題を解くと以下ようになります。

```
Python$ python3 NP.py -s data/Problem500.txt
```

```
--- 中略 ---
```

```
Total 500    Success 500
```

```
Time      11.032532 sec
```

500問を11秒で解いているので、1問あたり、0.022秒（22ミリ秒で解けていることが分かります。

## NP.py

では、プログラムで、問題を読み込み、実行する部分を追いかけていきましょう。

NP.pyの最後に、`def main(args)` があり、ここから始まるようになっています。

コマンドオプションで `-s` を付けた時に、引数のファイルを問題ファイルとみなして解きます。

```
225     solveNP(args[2])
```

これが、解くプログラムを呼び出しています。

## class Problem

`solveNP()`の説明の前に、ソルバー、ジェネレー共通でよく利用する問題データクラス (`class Problem`)を説明します。

```
19 class Problem:
20     def __init__(self):      # , prob, idstr, blk, ans, pat ):
21         self.problem = None
22         self.id      = None
23         self.blanks  = None
24         self.answer  = None
25         self.pattern = None
```

クラスのメンバーの`problem`, `answer`, `pattern`はいずれも $9 \times 9$ のnumpy配列ですが、ここではサイズを指定していません。

`Problem`は問題を、`answer`は解を、`pattern`はヒントパターンを入れておく配列です。

`id`は、問題のタイトルを文字列で記憶しておくためのものです。

`blanks`は、問題を解いた時に決められなかった空白マスの数で、問題を自動生成するときの参考にします。

## **solveNP(filename)**

問題を連続して解く関数がsolveNP()です。

```

97  # 問題を解く
98  def solveNP(filename):
99      problems = []
100
101      with open(filename) as f:
102          lines = [l.rstrip('\n') for l in f.readlines()]
103
104      while len(lines) > SIZE:
105          pr = Problem()
106          line = lines.pop(0)
107          pr.id = readProblemTitle(line)
108          linebody = lines[:SIZE]
109          lines = lines[SIZE:]
110
111          pr.problem = readProblemBody(linebody)
112          ##          print(pr.problem)
113          problems.append(pr)
114          ##          printBoard(sys.stdout, pr.problem)
115
116      # 全問を解くループ
117      start_time = time.perf_counter()
118
119      for pb in problems:
120          pb.blanks = solver.solve(pb.problem)
121          if pb.blanks >= 0:
122              pb.answer = solver.getAnswer()
123
124      exe_time = time.perf_counter() - start_time
125
126      # 全問を解いた結果表示のループ
127      success = 0;
128      for pb in problems:
129          if dataoutput!=None:
130              dataoutput.write(pb.id, '\n')
131              sys.stderr.write("{}\n".format(pb.id))
132
133      if pb.blanks < 0:
```

```

134         if dataoutput!=None:
135             dataoutput.write("ERROR\n")
136         sys.stderr.write("ERROR\n")
137     else:
138         printBoard(sys.stderr,pb.problem)
139         sys.stderr.write('{}\n'.format(pb.blanks))
140         printBoard(sys.stderr,pb.answer)
141         sys.stderr.write('\n')
142
143     if pb.blanks==0:
144         success += 1
145
146     probSize = len(problems)
147     sys.stderr.write( "Total {}    Success {}\n"
148                      .format(probSize, success))
148     sys.stderr.write( "Time\t{:06f} sec\n".format(exe_time) )

```

最初のwhileループ(101行~)で全問題を、行よりなるテキストリストlinesに読み込みます。

2番めのwhileループ(104行~)で、linesを10行毎に分け、Problemオブジェクトprにタイトル行と問題データをセットします。そして、問題リストproblemsに追加します。

107行の readProblemTitle(line) がタイトル行の読み取り、  
 111行の readProblemBody(linebody) が問題データの読み取りで、  
 読み取った内容はprの中のそれぞれのところに入ります。

1問読んで解き、次の問題を読んで解くという方法もありますが、これでは解く時の時間計測が面倒になるので、最初に全問を読み込んでリストに溜め込んでおき、次のforループ(119行~)で1問ずつ取り出して解きます。つまり、実行時間を正確に計測するために、入出力にかかる時間を含めないようにしています。

```

119     for pb in problems:
120         pb.blanks = solver.solve(pb.problem)
121         if pb.blanks >= 0:
122             pb.answer = solver.getAnswer()

```

どんなに多い問題数でも、このforループの中で全部解かれます。

各問題はループ毎にpbに与えられます。

1つの問題を解くのは、`solver.solve(pb.problem)`で行われます。

戻り値が負だと、何らかのエラー（不都合）が発生しています。

戻り値が0または正の場合、ソルバーから答えを取り出し、pbオブジェクトの`answer`に入れます。

この2番めのforループの前後で時間計測をして、その差を実行時間として最後に示しています。

最後の長めのforループで、解いた結果を表示したりファイルに書き込んでいます。

盤面のプリントとして2つの`printBoard`メソッドがあり、引数により問題盤面のプリントか、解のプリントかを区別しています。

```
138             printBoard(sys.stderr,pb.problem)
```

```
140             printBoard(sys.stderr,pb.answer)
```

これら以外に、ヘッダ行のプリント、空白マス数のプリントもあります。

あまり本質的な部分ではないので、他の説明は省略しますので、各自でご確認ください。

## `solver.py`

では、`solver`モジュール(`solver.py`)を調べましょう。  
与えられた1つの問題を解いています。

先頭に`import`があります。

```
9  import  parameter
10 import  numpy as np
11 import  sys
12 import  NP
```

次に、`solver`モジュールのグローバル変数があります。

```
14  SIZE = parameter.SIZE
15  SUBSIZE = parameter.SUBSIZE
16
17  board   = None
18  candidate = None
```

`SIZE`は9, `SUBSIZE`は3になります。

`board`は現在の盤面を示すので、実際には `SIZE×SIZE` の`numpy`の2次元配列で、`0`(空白)または決まった数字の1から`SIZE`が入ります。  
`Board`は問題を解き進めるための作業用の配列で、解き終わったときには問題の解が入っています。

candidateは本来は3次元の真偽値配列ですが、ここではint型にして、0/1だけを使っています。

第1次元は行（y座標）、第2次元は列（x座標）を示すのは、boardと同じです。値の範囲は0～8で、ゼロベースになっています。これは、1ベースよりも計算上都合が良いので、そのようにしています。

candidate[r][c][n]

r : 行 (row) (0～8)

c : 列 (column) (0～8)

n : 候補数字 (1～9)

第3次元が候補数字を示します。

問題を解く時、まだ数字の入っていないマスに

は、1～9までの数字が入る可能性があります。

しかし、問題の初期ヒントの数字の影響

や、解き進んだときに決めたマスの数字の影響で、各マスに入れられる数字はどんどん減っていきます。

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

右図の白マスに小さく書かれている数字が候補数字です。ルールに従って、同じタテ、同じヨコ、同じ3×3のブロックにある数字を候補から削除すると、図のように減ってきます。

候補数字の部分だけ、添字として1～9を使っています。これは、0～8を使うより便利なので、配列が少し無駄になるけれども、そのようにしています。

このcandidateを監視し、決まる箇所を次々と決めていくことがナンプレを解くことになります。同じマスの9つの候補の内、1つだけが1になったとき、そのマスはその数字に決定します。

では、実際にプログラムに従って、動きを見ていきましょう。



## def solve( bd )

問題bd(numpyの2次元配列)を引数で与えると解く関数です。

戻り値は解けるところまで進んだ時に残った空白マス数になります。もし矛盾が発生した場合には -1 を返します。

```
20 def solve( bd ):  
21     initialize()  
22     blanks = setProblem(bd)  
23  
24     if blanks < 0:  
25         return -1  
26  
27     try:  
28         checkLoop()  
29     except:  
30         return -1  
31  
32     return blankCount()
```

最初にinitialize()で初期化します。

initialize()の定義は34行からですが、solverの2つのグローバル変数 board, candidate を初期化しているだけです。

setProblem(bd)で、引数で渡された問題を、オブジェクトのメンバー配列にセットします。そして、その間にも問題のチェックを行い、異常があれば負の値を返してきます。

異常がないことが分かれば、そこでcheckLoop()を呼び出し、延々と解きます。これ以上解けなくなったらcheckLoop()は終了します。

ループ中で矛盾を見つけると例外が発生し、その場合にはsolve(db)は -1 を返します。

矛盾が起きなかった場合は、blankCount()を呼んで、残りの空白マス数を計算し、そのまま戻り値とします。

## def initialize()

問題を解くための準備、初期化を行います。

```
34 def initialize():
35     global candidate, board
36
37     board = np.zeros((SIZE,SIZE)).astype(int)
38     candidate = np.ones((SIZE,SIZE,SIZE+1)).astype(int)
```

盤面配列のboardは、全要素に0をセットします。これで、数字が盤面のどこにも入っていない状態にしています。

候補配列のcandidateは、全要素に1をセットします。つまり、全てのマスに対して、1から9の全ての数字は候補である、とセットします。

## def setProblem( bd )

setProblemは、引数で与えられた盤面bdを、ソルバーの作業盤面boardにセットし、決まるマスは決めてしまい、最後に残りの空白マス数を戻り値とします。また、セットの途中で矛盾が発生した場合には、戻り値を-1とします。

```
41 def setProblem( bd ):
42     try:
43         for r in range(SIZE):
44             for c in range(SIZE):
45                 if bd[r][c] != 0:
46                     setValue( r, c, bd[r][c] )
47     except:
48         return -1
49
50     return blankCount()
```

setProblem(bd)は、引数で与えられた問題の数字(1からSIZE)を設定しながら、作業用ボード配列boardと、候補配列candidateを変更していきます。

try-exceptで括っていますが、これは矛盾することがあればsetValue()以下で発生する例外を捉えるためです。

最初の2重のforループで、与えられた問題の実際に数字が入っているマスについてだけ setValue( c, r, bd[c][r] ) を呼び出している。

これは、与えられた数字を、指定の場所に設定するためのメソッドを呼び出しているのですが、この呼び出したメソッドの中で、マスに数字を入れる場合のチェックが行われています。

## def setValue( r, c, v )

第1、第2引数でマスの位置、第3引数でマスにセットすべき値を指定し、盤面に数字を書き込みます。書き込みの前に、矛盾のチェックも行います。

```
86 def setValue( r, c, v ):
87     if candidate[r][c][v]==0:
88         raise Exception
89
90     if board[r][c]!=0:
91         if board[r][c] != v:
92             raise Exception
93     return
94
95     board[r][c] = v
96     for n in range(1,SIZE+1):
97         candidate[r][c][n] = 0
98
99     r0 = (r//SUBSIZE)*SUBSIZE
100    c0 = (c//SUBSIZE)*SUBSIZE
101
102    for i in range(SIZE):
103        candidate[r][i][v] = 0
104        candidate[i][c][v] = 0
105        candidate[r0+(i//SUBSIZE)][c0+i%SUBSIZE][v] = 0
```

まず、candidate[r][c][v]が0でない、つまりそのマスにその数字をセットして良いか確認します。

次に、数字を入れるマスに既に0以外の数字が入っていて、その数字がこれから入れようとする数字と異なる場合、例外が発生します。問題として与えられた数字が、他のマスとの関係から既に決定していることもあるので行っている確認です。

これで問題なければ、`board[r][c] = v` で数字をセットします。

次に、数字を決めたことで、候補配列`candidate`の、入れた数字の影響を受ける部分を全部0(使用済)にします。

## **def solve()**

```
20 def solve( bd ):
21     initialize()
22     blanks = setProblem(bd)
23
24     if blanks < 0:
25         return -1
26
27     try:
28         checkLoop()
29     except:
30         return -1
31
32     return blankCount()
```

最初に初期化して、次に引数で与えられた問題をセットします。

現状の`board`で、どこまで解けるか延々と試します。

実際に試すのは、`checkLoop()`の中です。

問題なく解き進められたら、最後に空白マスを数えて戻り値とします。

## def checkLoop()

ルールに従って、これ以上決まるところが無くなるまで、延々と調べます。

```
53 def checkLoop():
54     changed = True
55     while changed:
56         changed = False
57
58         for r in range(0,SIZE,SUBSIZE):      # 3x3 Blocok Check
59             for c in range(0,SIZE,SUBSIZE):
60                 if checkBlock(c,r):
61                     changed = True
62             if changed:
63                 continue
64
65         for r in range(SIZE):                # HLine check
66             if checkHline(r):
67                 changed = True
68
69         for c in range(SIZE):                # VLine check
70             if checkVline(c):
71                 changed = True
72
73         if changed:
74             continue
75
76         for r in range(SIZE):                # Cell check
77             for c in range(SIZE):
78                 if checkCell(c,r):
79                     changed = True
80
81         if blankCount() == 0:
82             break
```

全体は、何らかの変化がある限り、forループを永久に回ります。そのうち変更することが無くなるか、矛盾が発生してループが終了する筈です。

forループの中が4つの処理に分かれています。

- ①3x3のブロックで、決まるところを探す。行rと列cを、3ずつ増加。
- ②ヨコ列について決まるところをチェックする。
- ③タテ列について決まるところをチェックする。
- ④マス（cellと呼ぶ）についてチェックする。

それぞれについて、専用のチェック関数が用意されています。チェック関数がTrueを返してきた時、新たに数字が決まった、つまり変化があったので、変更フラグchangedをTrueにします。

4つの処理は、①がブロックを調べるもので、ここで変化が発生したら、②以降へは進まず、ループを先頭から再開します。

②または③で変化が発生したら、④には進まず、ループの先頭から再開します。

これは、人間が調べる時と同様に、簡単に見つけられるものから順番に並べて、変化が起きたのが分かったら、また易しい調べ方からやり直しています。

forループを一周して何か変化が起きた場合には、forループをまた最初から続けていますが、一周して何も変化が起きなかった場合は、これ以上ループを回り続けても無駄なので終了します。

また、空白マスが0になった場合には、問題を正常に解き終わっているので、終了します。

## **def checkBlock( c0, r0 )**

3x3のブロックをチェックするのがcheckBlock関数です。  
引数c0, r0は、ブロックを左上隅で指定します。

```
146 def checkBlock( c0, r0 ):  
147     changed = False  
148     for n in range(1,SIZE+1):  
149         exist = False  
150         cnt = 0  
151         col = 0
```

```

152     row = 0
153     for r in range(r0,r0+SUBSIZE):
154         for c in range(c0,c0+SUBSIZE):
155             can = candidate[r][c][n]
156             if board[r][c] == n:
157                 exist = True
158                 if can == 1:
159                     cnt += 1
160                     col = c
161                     row = r
162
163     if not exist:
164         if cnt == 1:
165             setValue( row, col, n )
166             changed = True
167         elif cnt == 0:
168             raise Exception
169
170     return changed

```

一番外側のforループのnは、指定した3x3のブロック内で決められる数字を1から順に調べるための変数です。

既にブロック内に数字nが存在する場合があります、その場合はそれ以上調べる必要がありません。

そうでない場合には、ブロック内のどこかのマスが数字nに決まる可能性があります。そのため、ブロック内の数字nの候補の数(candidate[r][c][n]が1(True)の数)を数えます(cnt)。

そして、そのマスの位置をcolとrowに記憶しておきます。

前半のforループを終えたら、ブロック内の数字nの候補数と、数字nが既にブロック内で決まっているかが分かっています。

後半で、数字nが決まっていなかった時、そして候補が1つだけだった場合には、そのマスに数字nを入れます。数字をセットするのは、既に説明したsetValueメソッドを用います。このメソッドは、単にマスに数字をセットするだけではなく、その数字をセットしたことで影響を受ける候補の削除も行ってくれます。

まだ数字nが決まっていなくて、数字nの候補数が0、つまりどこにも入れられるところがない場合には矛盾しているので、Exceptionを発生します。

このメソッドでは、外側のnのループでは、1つの数字が決まっても、さらに次の数字について調べるようになっていきますので、複数の数字を一度の呼び出しだけで決めることもあります。

最後に、changedを返すことで、本関数で変更があったかどうか呼び出し元に返されます。

```
def checkHline(r)  
def checkVline( c )
```

この2つの関数は、ヨコ列またはタテ列について、決まる数字があるかどうかチェックするもので、checkBlock( int c0, int r0 ) と同じ構成になっています。ヨコ列またはタテ列の9マスについて調べるので、checkBlockよりも簡単になっていますので、ご自身で内容を確認してください。説明は省略します。

```
def checkCell( int c, int r )
```

checkCellは、1つのマスの候補をだけを調べて数字を決められるかどうかをチェックします。つまり、マスに候補を小さい数字で書き込んだ時、マスに小さい数字（候補）が1つだけだと、その数字がそのマスの数字になります。これを行うのが、checkCellです。

candidate[r][c]は、rとcで指定されたマスに入れられる数字の候補の配列です。もし、この配列で1の個数(入れても良い数字の個数)を数え上げたとき、1が1個だけだと、その1に対応する数字に決定します。

```
216 def checkCell(c,r):  
217     if board[r][c] != 0:  
218         return False
```



```

219
220     cnt = 0
221     v = 0
222     for n in range(1,SIZE+1):
223         if candidate[r][c][n]!=0:
224             cnt += 1
225             v = n
226
227     if cnt == 1:
228         setValue( r, c, v )
229     elif cnt == 0:
230         raise Exception
231
232     return cnt==1

```

## def blankCount()

boardの中の空白(0)、つまり未決定マス数を数えます。

```

236 def blankCount():
237     cnt = 0
238     for r in range(SIZE):
239         for c in range(SIZE):
240             if board[r][c] == 0:
241                 cnt += 1
242
243     return cnt

```

空白マス数は非常に重要なので、例を使って説明します。  
 ソルバーで問題を解いた時(-sを指定) の出力で確認できます。

同じハートの問題ですが、一番下の2を3に変更しただけで、結果が下図のように変わります。上側が問題、次の1行が残り空白マス数（赤色）、最後が問題を解けるところまで解いた結果です。

左は解き切ることができたので、残りの空白マス数は0になっています。

右は解き切ることができず、空白マスを示す-(ハイフン)が29個あることを示しています。

解けるところまで延々と解き進んで、その時に盤面を示す配列boardの要素が0のマス（空白マス）を数え上げ、戻り値として返します。

Heart	H	20
-	-	-
-	6	2
7	-	8
1	-	-
9	-	-
-	7	-
-	-	5
-	5	-
-	-	7
-	-	2
0		
5	8	7
9	6	3
1	2	4
2	1	8
4	7	5
6	3	9
7	5	6
3	4	2
8	9	1

Heart	H	20
-	-	-
-	6	2
7	-	8
1	-	-
9	-	-
-	7	-
-	-	5
-	5	-
-	-	7
-	-	3
29		
5	8	7
9	6	-
1	2	-
-	1	8
-	7	5
-	3	9
7	5	6
3	4	2
8	9	1

空白マス数は、問題の自動生成で重要な働きをします。この数字が0に近づくように問題の数字をどんどん入れ替えていき、0になったところでユニーク解になったことが分かるので、そこで問題の調整を完了とし、問題を取り出して表示するのが自動生成です。

## その他

以上が、問題を解くための主要なメソッドです。  
それ以外にもいくつかのメソッドを用意しています。

プリント関係では、

<code>printCandidate()</code>	候補(candidate)をプリントする
<code>printBoard()</code>	盤面(board)をプリントする

プリントの実体はNP.pyの中のprintBoard関数  
を利用しているだけです。

がありますが、説明は省略します。

ところで、盤面のプリントなど、さまざまな情報を出力するのに、普通なら  
`print` (標準出力) を使うのですが、このプログラムでは、ほとんどの場合  
`sys.stderr.write`(エラー出力)を使っています。

わざわざ`sys.stderr.write`を使っているのは、以下の理由からです。何らかのトラブル  
に対してシステムから出力されるエラーメッセージはエラー出力に出力されるのです  
が、エラー出力は標準出力よりも優先して出力されます。こうなると、エラーが発生  
するよりも前にプリントしたはずの出力が、エラー出力よりも後に表示され、出力の順  
番とプログラムの流れに不一致が発生し、デバッグに不都合になります。そのため、ほ  
とんどの出力で`sys.stderr.write`を使っています。

値の取得関係では、

```
def getAnswer()  
    解をコピーして戻り値として返します。  
def getValue(r,c)  
    指定したマスの値を返します。  
def getCandidate(r,c)  
    指定したマスの候補の配列を返します。
```

がありますが、いずれも簡単ですので、説明は省略します。

以上だけから、500問の問題が入ったProblem500.txtを解かせると、次のように表示  
します。解くのにかった時間は、10.9秒で、1問あたり0.0218秒(21.8ミリ秒)に  
なっています。

Python\$ python3 NP.py -s data/Problem500.txt

No.1 H 18

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

0

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

No.2 H 18

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

0

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

---- 中略 ----

No.499 H 24

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

0

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

No.500 H 24

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

0

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

Total 500 Success 500

Time 10.892606 sec

## 第2部 問題を作る

問題を解くソルバーができたので、第2部では、このソルバーを利用して問題を自動生成してみましょう。

問題を作る部分は`generator.py`(`generator`モジュール)にまとめられています。

ソルバー`solver.py`よりも`generator.py`の方がプログラムが短くなっています。

問題を作るのは、解くよりも遥かに困難だと思いませんか。でも、プログラムの長さを見ると、`generator.py`の方が短くなっています。

これは、問題を解くより作るほうが簡単ということかも知れません。

では、実際に、`generator`はどのようなになっているか調べてみましょう。

問題を作るときには、コマンドパラメータに `-g` で問題生成であることを伝え、その後に問題パターンファイルを指定します。  
すると、以下のように実行されます。

```
Python$ python3 NP.py -g data/HeartP.txt
```

```
No.1    H 20
```

```
- - - - -  
- X X - - - X X -  
X - - X - X - - X  
X - - - X - - - X  
X - - - - - - - X  
- X - - - - - X -  
- - X - - - X - -  
- - - X - X - - -  
- - - - X - - - -
```

```
*****SUCCESS TRY 16
```

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

```
total 1 failure 0
```

```
Time    193.364106 sec
```

16回再トライして完成しました。時間は193秒でした。

では、500問のパターンファイルで、一気に500問を自動生成してみよう。

```
Python$ python3 NP.py -g data/Pattern500.txt
```

```
No.1    H 18
```

```
- - - - X - - - -
```

```

- - - X - X - - -
X X - - - - X - -
- - X - - X - - -
- X - - - - - X -
- - - X - - X - -
- - X - - - - X X
- - - X - X - - -
- - - - X - - - -

```

\*\*\*\*\*SUCCESS TRY 13

```

- - - - 3 - - - -
- - - 6 - 2 - - -
9 8 - - - - 1 - -
- - 2 - - 9 - - -
- 7 - - - - - 8 -
- - - 1 - - 5 - -
- - 6 - - - - 2 9
- - - 5 - 7 - - -
- - - - 8 - - - -

```

No.2 H 18

```

- X - - - - X - -
- X - X - - - X -
- - - X - - - - -
- - - - - X X - -
- - X - - - X - -
- - X X - - - - -
- - - - - X - - -
- X - - - X - X -
- - X - - - - X -

```

\*\*\*\*\*SUCCESS TRY 24

```

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

```

◇ ◇ 以下略 ◇ ◇

とても時間がかかるので、中断しました。1日くらいかかりそうです。

実行すると、問題が表示され、\*が次々に表示され、SUCCESSと表示されると問題作成に成功したことを示します。その後のTRYは成功までに何度やり直したかの回数を示します。問題作成に成功した場合には、その後に出来上がった問題を表示します。

何度やり直しても成功しない場合には、FAILUREと表示され、問題は表示されません。最後に、全体の作成時間を表示します。

以上が自動生成をしたときの動きですが、これをプログラムでどのように実現しているか見ていきましょう。

## NP.py

自動生成(generator)とソルバー(solver)はかなり似た構成部分もあり、共通するような部分は、自動生成関連のプログラムもNP.pyの中に書かれています。

### def generateNP(filename)

```
150 def generateNP(filename):
151     problems = []
152
153     with open(filename) as f:
154         lines = [l.rstrip('\n') for l in f.readlines()]
155
156     while len(lines) > 0:
157         pr = Problem()
158         line = lines.pop(0)
159         pr.id = readProblemTitle(line)
160         linebody = lines[:SIZE]
161         lines = lines[SIZE:]
162         if len(linebody) == SIZE:
163             pr.pattern = readPatternBody(linebody)
164             problems.append(pr)
```



```

165
166     # 全問作るループ
167     start_time = time.perf_counter()
168
169     failureCount=0
170     successCount=0
171     n=0
172
173     for pb in problems:
174         pattern = pb.pattern
175         n += 1
176         sys.stderr.write("No.{}    H {} \n"
                           .format(n, countHint(pattern)))
177     if dataoutput != None:
178         dataoutput.write("No.{}    H {} \n"
                           .format(n, countHint(pattern)))
179     printHintBoard(sys.stderr, pattern)
180
181     if generator.generate(pattern):
182         pb.problem = generator.getProblem()
183         if dataoutput != None:
184             printBoard(dataoutput, pb.problem)
185             printBoard(sys.stderr, pb.problem)
186             successCount += 1
187     else:
188         if dataoutput != None:
189             dataoutput.write("FAILURE \n")
190             sys.stderr.write("FAILURE \n")
191             failureCount += 1
192         sys.stderr.write(' \n')
193
194     exe_time = time.perf_counter() - start_time
195
196
197     probSize = len(problems)
198     sys.stderr.write( "total {}  failure {} \n"
                       .format((successCount+failureCount), failureCount))
199     sys.stderr.write( "Time\t{:06f} sec \n".format(exe_time) )

```

与えられたパターンファイルを一気に読み込んで、延々と問題を作る関数です。構成は、ソルバーのsolveNP()に非常に似ています。

まず、出来上がった問題を格納するための問題リスト`problems`を初期化します。

最初の`while`ループ(156行~)で、問題パターンを読み込んで、`problems`にパターンを入れます。パターンを0/1の2次元配列に読み込むのが、`readPatternBody()`関数です。

延々と問題を作り続けるのが、真ん中あたりの大きな`for`ループです。

`for`ループの前半で、ヒントの数を数えて、ヒント数とヒントの配置をプリントします。

1つのパターンに対して問題を実際に作るのが、`generator.generate(pattern)`で、問題作成に成功すると`True`を、失敗すると`False`を返してきます。成功のときは出来上がった問題を表示し、失敗したときには`FAILURE`の表示します。そして、成功、失敗の数も数えます。

最後に、現在時刻を求め、問題作成直前の時刻との差より経過時間を求めて表示します。

ということで、実際に問題を作っている部分は`generator.generate(pattern)`であることが分かりました。

## Solution.py

問題を生成するときに、問題生成の種になる、全マスに数字が詰まった解を用います。  
この解は、呼ばれる毎に異なる解でなければいけません。

グローバル変数 `board` に、全マスに値が入った初期盤面が入っています。これを、適当に変更しては返すだけで、延々と異なる解を呼び出し毎に返しています。

簡単にテストできるように作られているので、まず動作確認してみます。

```
Python$ python3 solution.py
```

```
No. 0
```

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

```
No. 1
```

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

◇◇◇ 以下略 ◇◇◇

10個の解を表示して止まります。

それぞれの解が全然違っていることが分かるでしょう。

## def getANewSolution()

```
32 def getANewSolution():
33     for i in range(REPLACE):
34         line1 = random.randrange(SIZE)
35         line2 = line1+1
36         if line2 % SUBSIZE == 0:
37             line2 -= SUBSIZE
38         if i % 2 == 0:
39             exchangeVline( line1, line2 )
40         else:
41             exchangeHline( line1, line2 )
42     return board
```

forループをREPLACE(10)回周り、カウンタの奇偶により、縦2列、あるいは横2列を入替えています。

何番目の列にするかをline1で決め、次にその隣の列を選びます。ただし、SUBSUZEの切れ目（太線枠）を跨いで2列を入替えられないので、line2は同じ太線枠の中に収まるように調整しています。

実際の入替えは、縦と横で別々の関数を用意しています。

内容は簡単なので省略します。

## generator.py

では、自動生成の本体であるgeneratorモジュール(generator.py)を調べましょう。

ここが、問題を実際に作っている部分であり、肝心な部分です。

プログラムに沿って詳しい説明をする前に、このモジュールでは何をしているのかの概要を説明をします。

最初、指定されたヒント位置の数字を適当に決めます。でも、完全にランダムに決めると、同じ列、同じ行、同じ3×3のマスの中に同じ数字が入ってしまう可能性が非常に高いです。また、解けるかどうか確認する時に、解くことが出来ない矛盾のある問題になっている可能性も高いです。

これを避けるため、適当な全マス数字が詰まっていて、ルールを満たす任意の解を用意します。この任意の解はどんな方法で作っても構いませんが、できるだけランダムになるように作ります。

解サンプル	マスクで問題作成	解く、空白マス数は？
8 2 4 1 5 7 3 9 6 9 1 5 4 6 3 7 2 8 3 7 6 9 2 8 1 4 5 2 8 9 3 1 6 5 7 4 7 5 1 2 8 4 6 3 9 6 4 3 5 7 9 8 1 2 4 6 8 7 3 2 9 5 1 5 3 2 6 9 1 4 8 7 1 9 7 8 4 5 2 6 3	- - - - - - - - - - 1 5 - - - 7 2 - 3 - - 9 - 8 - - 5 2 - - - 1 - - - 4 7 - - - - - - - 9 - 4 - - - - - 1 - - - 8 - - - 9 - - - - - 6 - 1 - - - - - - - 4 - - - -	55 - - - 1 - - - 9 - 9 1 5 - - - 7 2 8 3 - - 9 - 8 1 - 5 2 - - - 1 - - - 4 7 - 1 - - - - - 9 - 4 - - - - - 1 - - - 8 - - - 9 - - - - - 6 - 1 - - - - - - - 4 - - - -

次に、全部詰まった盤面から与えられた問題パターンで指定されたマスの数字だけ残り、それ以外は全部0にクリアします。すると、問題の形をした問題らしきものができるがります。

問題らしきものを解いて、ユニーク解であることが分かれば、これで問題作成が完成です。

普通は、空白マスがいくつも残ります。

上の例では、解いても6マスしか決まらず、残りの空白マス数は55になりました。

この、空白マス数をどんどん減らす方法を考えます。

ここでは、ヒントのうちから2箇所（マス）をランダムに選び、その2つのマスの数字を色々入替えた問題をつくり、それを解いて空白マスがどれだけ残るかを調べます。

### 2つのヒントを消去

```

- - - - -
- 1 5 - - - 7 2 -
# - - 9 - 8 - - 5
2 - - - 1 - - - 3
7 - - - - - - 9
- 4 - - - - 1 -
- - 8 - - - # - -
- - - 6 - 1 - - -
- - - - 4 - - - -

```

### 2つのヒントの入替

```

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

```

### 解く、空白マス数は？

52

```

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

```

入れ替える数字によっては、ルール上不可能な場合（最初からルールと矛盾したり、解き進めていくと途中で矛盾する）には、その入替えはマズイということで、無視することになります。

このヒント2箇所の入替えを行って解くと、残り空白マスの数が、入替え前よりも減ることがあります。これは、問題がより完成に近づいたと考えられます。それで、次からより残り空白マスの数が少なくなった問題を基準に、同様の変更を繰り返して、残り空白マスがより少ない問題を探していきます。そして、残り空白マスが0になったら、ユニーク解になったので、問題の完成です。

## 2つのヒントの入替

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

## 解けた。ユニーク解。

0
6 7 9 1 5 4 8 3 2
8 2 5 3 7 6 1 4 9
3 1 4 9 2 8 6 7 5
9 8 3 7 1 5 2 6 4
7 5 1 4 6 2 3 9 8
2 4 6 8 9 3 7 5 1
4 6 8 5 3 1 9 2 7
5 9 2 6 8 7 4 1 3
1 3 7 2 4 9 5 8 6

## 完成した問題

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

2箇所の全入替えを調べても残り空白マス数が少なくなる場合があります。そのときは、問題盤面を入替え前の状態に戻し、異なる2箇所の組み合わせを選択し、同じことを繰り返します。

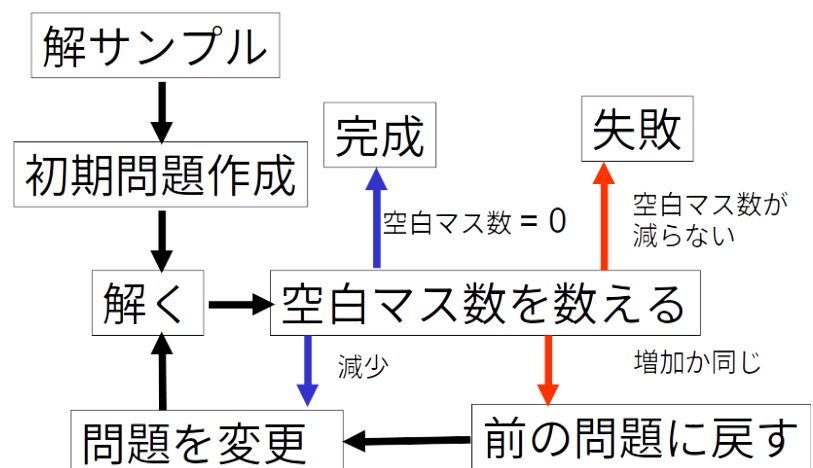
これを延々と続けても、全然空白マス数が減らないこともあります。このような状態が延々と続いてしまうことがあるので、減らない状態が何回続くか数えて、一定の回数続いたら諦めることにします。

以上は、最初に解サンプルから問題のヒントパターンに従って問題を型抜きした状態から徐々に改良（進化）を繰り返して完成するまでの流れです。

しかし、この方法では、最初の解サンプルによっては上手く行かないこともよくあります。それで、これで一度駄目だったら、再度解サンプルを作り直してから同じことを繰り返すと、できるかもしれません。

解サンプルから作り直すのを何度も何度も繰り返せば、そのうち完成するのではないかと甘い考えで本プログラムは作られています。

実際にどのくらい上手く行くかは、ご自身で動かして確認してください。



では、プログラムに沿って説明します。

## getHintArray()

ヒントパターンが与えられた時、このパターンのXの位置を配列で持っておくと都合がよいです。  
そのヒントの位置配列を作って返すのがこのメソッドです。

```
- - - - - - - -  
- X X - - - X X -  
X - - X - X - - X  
X - - - X - - - X  
X - - - - - - - X  
- X - - - - - X -  
- - X - - - X - -  
- - - X - X - - -  
- - - - X - - - -
```

```
91 def getHintArray():  
92     global hintpos, pattern, hintcount  
93     hintpos = np.zeros((hintcount,2)).astype(int)  
94  
95     idx = 0  
96     for r in range(SIZE):  
97         for c in range(SIZE):  
98             if pattern[r][c] != 0:  
99                 hintpos[idx][0] = r  
100                 hintpos[idx][1] = c  
101                 idx += 1  
102     return hintpos
```

ヒント数（問題の数字の個数）`hintcount` は既に決まっているものとします。  
このとき、ヒントパターン `pattern` から、ヒントに指定されたマスの位置を`hintpos`にセットします。

`hintpos`は、`numpy`の2次元配列で、第1添字がヒントの順番、第2添字により行と列を記憶します。

ヒントの9x9のboolean配列である`pattern`に既にヒントが入っています。

最初にヒントの位置配列`hintpos`を、`numpy`の2次元配列で、全要素0として作成しています。



次に、`pattern`配列の全要素を調べ、もし0でなかったらその位置を記録します。全位置を調べる時に、水平方向を`r(row)`、垂直方向を`c(column)`という変数を2重のforループで調べています。

上のハートの場合には、次のように`hintpos`に記録され、これを戻します。

```
[[1 1] [1 2] [1 6] [1 7] [2 0] [2 3] [2 5] [2 8] [3 0] [3 4] [3 8]
[4 0] [4 8] [5 1] [5 7] [6 2] [6 6] [7 3] [7 5] [8 4]]
```

では、最初から説明していきます。最初にいくつかの変数が宣言されていますが、これらは利用する都度説明していきます。

## **def generate( pat )**

引数で与えられたヒントパターンから問題を作る関数です。

中央付近の、`generateOnce( pat )` が頑張って「一度」だけ、解サンプルから問題を作る関数です。一度だけではなかなか成功しないので、問題作成に成功するまで400回まで繰り返し試します。

400回のループの途中で完成すれば成功の表示と再試行回数をプリントし、ループを中断し`true`を返します。

400回ループしても完成しなければ、`False`を返し、あきらめたこと(`False`)を伝えます。400は実行してみて、適当に決めた値です。

```
29 def generate( pat ):
30     for i in range(400):
31         print('*', end='', file=sys.stderr, flush=True)
32         if generateOnce( pat ):
33             sys.stderr.write( "SUCCESS TRY {}\n".format(i))
34             return True
35     return False
```

これ以外に、1回ループする毎に\*をプリントしています。そのため、なかなか完成しないと\*が多数並びます。

## def generateOnce( pat )

この関数が、1つの解サンプルを出発点にして、延々と最大200回までループを回りながら問題の完成を試みるもので、本プログラムで最も肝心なメソッドです。  
ここの200も適当に決めた値です。

```
37 def generateOnce( pat ):  
38     global pattern, xcells, blankcount, problem  
39     global hintarray  
40  
41     pattern = pat  
42     xcells = np.zeros((XCOUNT,2)).astype(int)  
43     initialSetting()  
44  
45     i = 0  
46     while i < 200:  
47         if blankcount == 0:                                # SUCCESS!!  
48             break  
49  
50         backup = problem.copy()  
51  
52         ## select XCOUNT cells and changed them  
53         selectXCells()  
54         clearXCells()  
55  
56         ## change some cells value on problem  
57         blk = changeXCells()  
58  
59         ## if new problem is better, update current blankcount,  
                                                and continue  
60         ## else restore problem  
61         if blk >= 0 and blk < blankcount:  
62             blankcount = blk                ## update blankcount  
63             i = 0  
64         else:                                ## restore from backup  
65             problem = backup  
66             i += 1  
67  
68     return blankcount==0
```

関数の最初で、引数で渡されたパターンをgeneratorモジュールのグローバル変数として用意した配列patternに代入しています。

backupは、問題盤面を元に戻すときのためのバックアップです。

xcellsは、問題の中の数字を変更するセルの位置を記憶するための配列です。XCOUNTは何箇所変更するかで、現在は2になっています。

次に、initialSetting()を呼んで、必要な初期化を行っています。

initialSetting()は以下のようになっています。

```
70 def initialSetting():
71     global pattern, solution, problem, blankcount, hintcount, hintarray
72
73     solution = getANewSolution()
74
75     problem = makeInitialProblem( pattern, solution )
76
77     blankcount = solver.solve(problem)
78
79     hintcount = countTrue(pattern)
80     hintarray = getHintArray()
```

最初にgetANewSolution()で、解サンプルを入手し、solutionに入れます。

次に、solutionと問題パターンから、最初の問題を作るのがmakeInitialProblem関数です。makeInitialProblemはgenerator.pyの最後の方にありますが、簡単なので説明は省略します。

次に、

空白マス数を数えて、blankcountに入れます。

問題パターンからヒント数を数えて、hintcountに入れます。

ヒント位置の配列を作り、hintarrayに入れます。

ここまでの初期化で、その後forループを200回繰り返します。

ループの最初で、完成したか(残り空白マス数が0)を確認し、0だと問題が完成しているのでループを抜けます。

次に、問題の配列problemをバックアップ配列backupにコピーします。

単なる代入ではだめで、深いコピーを実行します。

`selectXCells()` で、変更するマス（セル）をランダムに選択します。

`clearXCells()` で、`problem`の変更するマスを0クリアします。

`changeXCells()` で、変更するマスをランダムに適当な値に変更し、残り空きマス数を`blk`(変更後の空きマス数)に入れます。詳細は後述。

`blk`が負(不可能な問題)だったときに、`backup`から`problem`にコピーすることで以前の状態に戻しforループを続けます。

`blk`が今までの(最小の)空白マス数以上の場合も、空白マス数が減らなかったので進化は無かったとして、同様に`backup`から`problem`にコピーして元に戻してループを続けます。

`blk`が今までの(最小の)空白マス数より少ない場合は、進化が進んだということで、変更後の`problem`はそのままにし、空白マス数`blankcount`を`blk`に更新してからforループを続けます。

forループを抜けてくるのは、問題が完成したとき(`blankcount==0`)と、200回ループを周り終えたときです。

この関数は、問題作成に成功したか否かを戻すので、`blankcount==0`により、成否を調べて戻しています。

## def changeXCells()

```
124 def changeXCells():
125     global problem, xcells
126
127     blk = solver.solve(problem)
128
129     for i in range(XCOUNT):
130         r = xcells[i][0]
131         c = xcells[i][1]
132         val = solver.getValue(r,c)
133         if val > 0:
134             problem[r][c] = val
```

```

135             continue
136
137     cans = solver.getCandidate(r,c)
138     val = selectCandidate(cans)
139     if val < 0:
140         return -1
141     problem[r][c] = val
142
143     blk = solver.solve(problem)
144     if blk < 0:
145         return -1
146
147     return blk

```

入替える予定のマスに実際に適当に数字を入れて、解いてみて、空白マス数を戻す関数です。

まず最初に、今の問題`problem`を解いて、空白マス数を`blk`に入れます。解くのは、空白マス数を求めるだけではなく、入替えマスに入れられる数字がどうなるか調べるためでもあります。

次に、入替えマスの個数だけ`for`ループを回ります。一回周るごとに、入替えマスの数字を決めていきます。

`for`ループの最初で、入替えマスの位置(`r`行`c`列)を求めます。

入替えマスは空白(`0`)にしたのですが、問題を解くと勝手に数字が決まるマスかも知れないので、それを調べます。

空白マスのまま解いていますが、それでもその入替えマスの値`val`が決まっている場合は、その入替えマスの値をその値にするしかないので、問題`problem[r][c]`にその値を代入し、次のループに進みます。

入替えマスの数字を決定する前に、入れられる数字の候補を`getCandidate(r,c)`で取り出し、`cans`で参照できるようにします。

`problem[r][c]`のマスに入れられる値は、`selectCandidate()`で決められます。これは、候補の中から可能な数字をランダムに1つ選びます。詳しくは後述。

```

138     val = selectCandidate(cans)

```

戻ってきた値が-1だとエラーなので、中断します。  
大丈夫だったら、`problem[r][c]`にその値を代入します。

これで、1つの交換マスに入れる数字が設定されました。  
その状態で問題を解いて、空白マス数を求めます。  
もし負の値だったらエラーなので中断します。

大丈夫だと、forループの次の周回により、次の交換マスに対して同じ処理を繰り返します。  
これを繰り返すことで、forループが終了した時、全ての交換マスについてランダムに値を変更し、解いて空白マス数がカウントされた状態になっています。

最後に、空白マス数を戻り値としてリターンします。

## **def selectCandidate( cans )**

`changeXCells()`の中で呼ばれます。  
引数で渡された可能な候補の中から、ランダムに1つの数字を選択し戻します。

```
149 def selectCandidate( cans ):  
150     r = random.randrange(SIZE)  
151     for i in range(SIZE):  
152         v = ((r+i) % SIZE) + 1  
153         if cans[v] != 0:  
154             return v  
155     return -1
```

まず、`r`に0~8までの範囲の乱数を求めます。  
この`r`から初めて、1つずつ増やしながら、引数で渡された候補が0でない候補数字を調べ、0でない場合はその候補数字を返します。

1から9を順に調べるだけだと、もっと簡単なプログラムになりますが、1から順に調べて最初に0でない数字を返すと、小さい数字になる確率が高くなり、問題の数字が小さい数字になりやすくなります。

ここでは、候補が0でない数字から、片寄ること無くランダムに数字を選ぶために、少し工夫をしています。

そのため、最初の乱数  $r$  から順に増やしていき、数字が一巡したら終わりにしています。

```
152      v = ((r+i) % SIZE) + 1
```

この式が少し難しいかもしれません。

最後の+1は、候補配列のサイズは10で、添字の0は使わず、実際にマスに入れる数字（1～9）を添字として渡された候補配列を参照(`cans[v]`)することで、まだ候補であるかどうか判断しています。

数字 $v$ が候補として可能だったら、その値を返します。

そして、この $v$ のループの中で、その数字 $v$ が入れられるかを`cans[v]`で調べています。可能なら、`problem[r][c]`に書き込むことで、入替えマスの1つを決めています。決められたら、それ以上調べる必要がないので、ループを抜けます。

forループを終えてしまった場合は、可能な数字は無かったということで、-1を返します。

数字の入替えは、このプログラムではできるだけ単純なアルゴリズムながら、ちゃんとした問題を作れるものを採用しています。

より効率の良い方法とかありますが、それらは読者への課題として残しています。良ければ、改良にチャレンジしてみてください。

次ページの実行例は、ヒント数18個のきれいな配置の場合の自動生成実行例です。

ヒント数が少なくなると、問題作成が可能なパターンでも、なかなか成功しなくて何度も作り直しを繰り返すことが多くなります。また、パターンにより、作りやすいパターンや作りにくい（成功しにくい）パターンがあります。

```
Python$ python3 NP.py -g data/18P.txt
```

```
No.1    H 18
```

```
- X - - - X - - -  
- X - - - X - - -  
- X - - - X - - -  
- - X - - - X - -  
- - X - - - X - -  
- - X - - - X - -  
- - - X - - - X -  
- - - X - - - X -  
- - - X - - - X -
```

```
*****SUCCESS TRY 31
```

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

```
No.2    H 18
```

```
X - - - - - X -  
- - - X - - - X  
- - X - X - - -  
- X - - - X - -  
- - X - - - X -  
- - - X - - - X -  
- - - - X - X - -  
X - - - - X - - -  
- X - - - - - X
```

```
*****SUCCESS TRY 11
```

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



No.3 H 18

- - - - X - - - -  
- - - X - X - - -  
X X - - - - X - -  
- - X - - X - - -  
- X - - - - X -  
- - - X - - X - -  
- - X - - - - X X  
- - - X - X - - -  
- - - - X - - - -

\*\*\*\*\*SUCCESS TRY 8

- - - - 5 - - - -  
- - - 4 - 6 - - -  
7 2 - - - - 8 - -  
- - 4 - - 5 - - -  
- 9 - - - - - 2 -  
- - - 3 - - 7 - -  
- - 6 - - - - 4 5  
- - - 7 - 9 - - -  
- - - - 2 - - - -

total 3 failure 0

Time 534.544169 sec

## 第3部 発展課題編

ナンプレの問題を自動生成する簡単だけれど強力な方法を説明しました。ヒント数が少なくなるに従って問題は作りにくくなります。ナンプレの問題は人が作ることもできますが、ヒントの少ない問題は手作業だけでは非常に困難になりますが、人工知能を使うことで、ヒント数の少ないスッキリした問題を作ることができることがわかりいただけたでしょうか。

ここでは、紹介したプログラムの抱えている問題点や改良点を紹介しますので、ぜひチャレンジしてみてください。

内容的には、プログラミングに関する事柄、パズルに関する事柄、人工知能の進化計算に関する事柄などがあります。

## 入力ファイル

現在は問題やパターンの前に1行のヘッダ行が入っていますが、途中で空白行をいれてより見やすくしたファイルが使いません。

空白行を無視するように変更してみましょう。

入力ファイルの形式がもっと自由になるように、改善してみてください。

## GUI

現在のプログラムは、CUIといって文字によるユーザインターフェイスになっています。マウスなどを使ってグラフィカルな画面で操作できるようにGUIになると便利でしょう。

1. 問題パターンを画面で入力できる
2. 出来た問題を解いて遊べる
3. ルール違反の部分のマスを明示して警告する
4. 解く時に各マスに候補を表示できるようにする

他にも色々考えられるでしょう。

GUIになると、急に実用的になったと感ずることでしょう。

## 作成中の問題の数字の入れ替え回数

作成中の問題の数字のうち、2マスを選んで別の数字に入れ替えることを繰り返すことで徐々にユニーク解に近づけています。

入替え回数の指定は、`generator.py`の最初の以下で指定されています。

```
20  XCOUNT = 2
```

この値を、1にした場合、3にした場合、性能はどのように変化するでしょうか。ぜひ試してください。

## 山登り法の改良

今の自動生成は、調整中の問題の2マスを選んで数字を適当に入れ替えて、もし残りの空白マス数が減ったら、入れ替え後の問題を調整中の問題するようになっています。

これでは、せっかく選んだ2マスの入れ替えですが、たった1種類の入れ替えを行っただけで、また別の2マスを選ぶということをしています。

選んだ2マスに入れられる数字の組み合わせは複数あります。それらの全組み合わせの中で最も残り空白マス数が少なくなったのを次の調整中の問題とすれば、もっと効率よく進化が進むかも知れません。

ぜひプログラムを変更して、調べてみましょう。

## 問題の難易度

本自動作成プログラムでは、問題の難易度の計算は省略しています。

問題を作れても、生成された問題がどのくらい難しいか易しいか、どのくらいの時間がかかりそうかが分からないと、人が解いて確認することになり、とても面倒です。問題を多数提供しようとする、難易度の自動判定が必要になります。

難易度はどうすれば判定できるか、チャレンジしてみてください。

## 再帰で解いてはいけない理由

ナンプレの問題を解くだけでしたら、再帰(リカーシブコール)プログラムにより、簡単にソルバーを作れます。再帰回数はせいぜい数万回程度ですので、今のコンピュータなら瞬時で終わります。

しかし、再帰を使った場合には、色々困ったことになります。

まず、複数解がある問題を普通は判定できません。自動生成に対応するためには、論理的に決まらない部分は残したまま残さなければいけません。

ナンプレの問題を人が解いて説明する時、再帰で解けましたではパズルとして成立しません。ちゃんと論理的に解ける、きちんとした理由でマスを埋めることができることが

重要です。理由自体はとても難しくても構いません。そのときには、とても上級の理由（手筋）で決まるということで、上級の問題ということになります。

## ナンプレの上級手筋への対応

本プログラムは、基本手筋と言われる、

- ①タテ列、ヨコ列、3x3ブロックという9マスの集合の中である数字の候補が1つのマスだけになった場合は、そのマスはその数字に決まる
- ②1つのマスの中に入れられる候補が1つになったら、そのマスはその数字がに決まる

だけを利用しています。solver.pyのcheckLoop()以下の関数で、これらの基本手筋に従って調べています。

ナンプレを解くための解法テクニック、解き方、定石、などがネット上に多数公開されています。checkLoop()に他の上級テクニックを入れると、入れた上級テクニックも使って数字を決めていくので、より難度の高い問題が作れるようになります。

知っているより高度なテクニック、あるいはネット上などで調べてより高度なテクニックを加えてみましょう。

## 調整中の問題の数を複数個に

このプログラムでは、1つの調整中の問題を、延々と変更してユニーク解に近づけようとしています。しかし、これではいつまでたってもユニーク解にたどり着けないことが多いので、適当な回数（現在は200回）試みて駄目だったら、最初からやり直しという方法になっています。

ちゃんとした問題を生成できるパターン配置でも、最初の解サンプルや変更する2マスの場所により、残り空白マスがゼロに達しないことがあります。このようになってしまった調整中の問題を局所最適解といいます。もうどんな2マスを入替えても、これ以上よくならない状況に陥ってしまったわけです。

これを避ける方法がいろいろ考えられるでしょう。

上手く行かなくなったら、一度に2マスではなく3マス変更してみるのはいかがでしょうか。

今のプログラムは、調整中の問題はただ1つでした。調整中の問題を多数にするとどうなるでしょうか。

調整中の問題を多数用意します。ここで、進化計算の用語にしたがって、2つ用語を定義しておきます。

個体：調整中の問題のこと

プール：多数の個体（調整中の問題）を入れておく容器（個体集合）

多数の個体を同時に進化させたいのですが、どうすれば可能でしょうか。

新しいプールを用意しておき、プールから個体を取り出し、変更しては新しいプールに変更後の個体を入れます。

そして、元のプールに入っていた良い個体(空白マス数の少ない個体)も全部新しい個体に移してしまいましょう。すると、新しいプールの個体数は増えてしまいます。個体を残り空白マス数でソートして、空白マス数が少ない個体から順に、前のプールと同じ個数だけ残して、あとの個体は捨てましょう。

すると、プール内の残り空白マス数の平均値は徐々に少なくなるでしょう。

この手順を延々と繰り返すと、残り空白マス数はどんどん減っていき、残り空白マス数が0の個体ができないでしょうか。

このやり方、ちょっと生物の進化に似ていると思いませんか。

こんな方法を試してみましょう。

# バラエティーナンプレ

ナンプレも9x9の標準的な問題だけではなく、対角線上のマスにも1から9の数字を1つつ入れる「対角線ナンプレ」という亜種があります。

対角線ナンプレに対応したSolverとGeneratorはどうすればできるでしょうか。

2つの対角線上に対する制約条件だけが追加されているので、その部分だけプログラムを追加すれば解くのも作るのもできるようになるでしょう。

			7	4	6			
		8				2		
4							6	
6							5	
3							1	
		6				7		
			2	3	4			

対角線ナンプレ以外にも様々なバラエティーナンプレがあります。また、通常のナンプレが重なり合った合体ナンプレというものもあります。

いずれも基本は同じです。興味があれば、プログラムを拡張してみましょう。

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

9x9の標準的なナンプレの問題が重なった「合体ナンプレ(GATTAI)」というものもあります。

重なっている部分には、同じ数字が入ります。

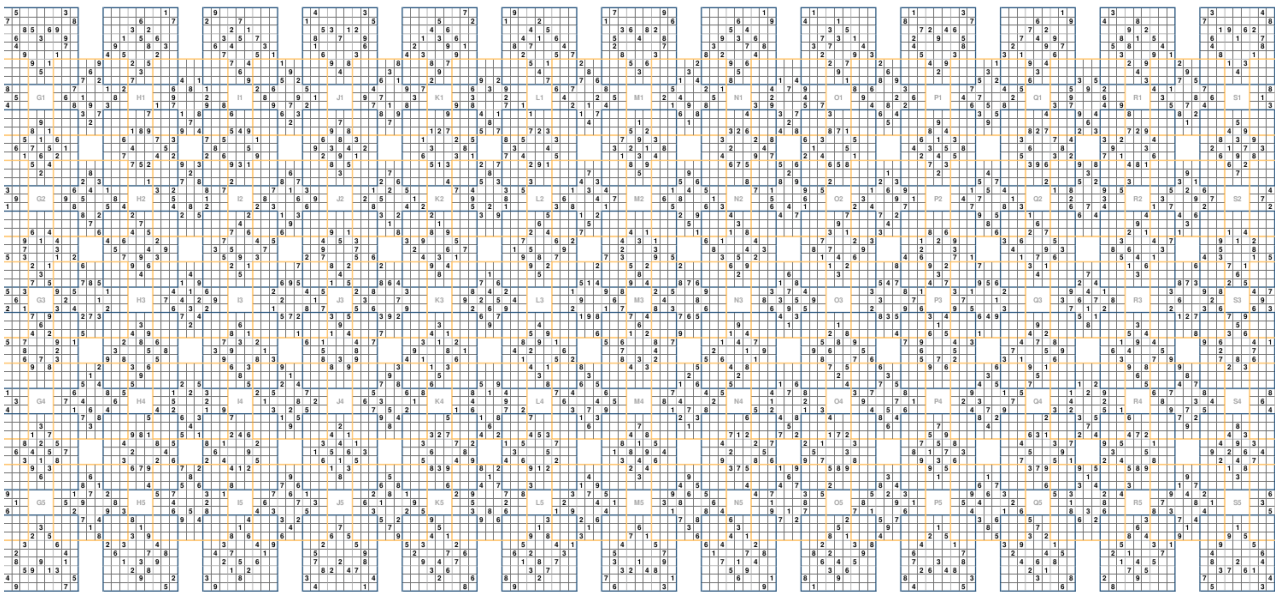
これも基本的には同じ考え方で解けるし、作ることもできます。

[illegible]

2018年10月には、MATH POWERという数学の祭典が、六本木のドワンゴのニコファーレというイベント会場で開催され、280合体の問題を提供しました。本イベントは2日間に渡る31時間連続イベントで、多数の来場者により解られました。

本問題は、合体ナンプレとして世界最大であるとギネスワールドレコード(TM)に認定されました。

史上最大 280合体ナンプレ
数学の祭典 MATH POWER 2018
10月6日(土)12:00~7日(日)19:00 31時間連続イベント
公式ニコニコ生放送



インターメディア 知能工学センター@電気通信大学100周年記念キャンパス    最適化による問題自動生成    合体数: 280    サイズ: 69x309    総マス数: 18180    重なり: 500ブロック=4500マス    ヒント数: 4604    ヒント数/合体数: 16.4429    空白マス: 13571





## 他のパズル

ナンプレ以外にも、多数の論理的に解くパズルが存在します。書店でパズル雑誌を見るとき、ネット検索すると簡単に確認できます。

どのパズルもルールがあり、ルールに従って解き進めます。

これらのパズルの多くも、コンピュータで解くことができます。

さらに、ナンプレの問題自動生成と同様に、問題を自動生成できるパズルも多数存在します。

パズルの解き方は、パズルのルールに沿ったプログラムを書くことで解くことができます。つまり、ソルバーの作成は、個々のパズルに依存します。

しかし、自動生成に関しては、ナンプレと同様の方法で、あるいはほぼそのままナンプレの自動生成を流用することができます。

興味のあるパズルがあれば、ぜひチャレンジしてみましょう。

アルファ碁がプロ棋士よりも強くなって、人工知能が急に脚光を浴びました。人間が行うもっとも高度な知的ゲームである囲碁は当分人工知能が人間に追いつけないだろうと言われていましたが、今では囲碁も将棋も、プロ棋士が人工知能を利用して腕を磨くのが一般的になりました。

そして、これをきっかけに、一気に様々な分野への応用が急激に進みました。

ゲームあるいはパズルは、昔から人工知能あるいはアルゴリズムの研究に非常によく利用されました。また、社会のさまざまな活動は、ゲーム（経済ゲーム）やパズルとして捉えることができると言われています。したがって、ゲームやパズルに強い人工知能が生まれたら、当然社会の課題の解決にも役立つだろうことは予想でき、実際コンピュータの歴史を見ても、ゲームやパズルで生まれた技術が応用されることは少なくありません。

## さまざまな社会課題への応用

問題（データ）を多数用意し、その問題を元に色々な方法で新しい問題を作り、良い問題だけを残しては同じ方法を繰り返すのは、生物の進化と同じですので、進化計算といいます。

とくに、遺伝の仕組みを模倣し、DNAの交叉と突然変異と同様にデータを変更し、良いものを残していくことを続けるのが遺伝的アルゴリズムです。

詳しい説明はしませんが、とても強力なアルゴリズムで、人工知能における重要なアルゴリズムの1つになっています。単独で利用することもあるし、他の人工知能手法と組み合わせて利用することもあります。

進化計算、あるいは遺伝的アルゴリズムをキーワードしてネット検索してみてください。いろいろな解説があります。また、書籍も出ていますので、参考にしてください。

進化計算・遺伝的アルゴリズムは、さまざまな社会の課題を解決するために使われています。面倒な組み合わせを最適化しないといけない場合はよくあります。多数のパラメータを同時に変更することは人間には大変面倒な、あるいは不可能なことですが、進化計算により実現することができます。

有名な応用例の1つが新幹線のN700系の先頭車両のフロントノーズの形状です。この形状は人が設計したのではなく、さまざまな条件下での最適化を遺伝的アルゴリズムで行って得られたもので、とてもエコでありながらスピードも出せるようになりました。様々な機器の自動運転、配置や分割の最適化、人員調整など多方面で利用されている技術です。

ぜひ人工知能の重要な分野の1つである進化計算/遺伝的アルゴリズムをマスターして、様々な社会課題の解決に取り組んでみましょう。

#### 著者紹介

TK-80など8ビットコンピュータ向けコンパイラ、画像処理、3Dソリッドモデラー、ポケットコンピュータ、日本語情報処理、X Window、Unix、Linux、科学技術計算処理、パズルプログラミングなどの開発を経て、最近は人工知能・進化計算を中心にプログラミング活動。

著作：『Cプログラミング診断室』、『Cプログラミング専門課程』、『(コ)の業界のオキテ』、『実践遺伝的アルゴリズム』など。

合体ナンプレの問題作成で280合体ナンプレのギネス世界記録(TM)を持つ。

## ナンプレの自動生成説明書 Python版

進化計算 天啓 | TENKEI

株式会社タイムインターメディア  
知識工学センター

東京都調布市小島町1-1-1  
電気通信大学 UECアライアンスセンター217

[puzzle@timedia.co.jp](mailto:puzzle@timedia.co.jp)

藤原博文

2022年1月10日