# SMT ソルバを用いた Slitherlink パズルの解法

宮内 哲夫[1,a)]　田中 清史[1,b)]

**概要**：Slitherlink はペンシルパズルといわれるパズルに属し，その解の存在問題は NP-完全問題に属する [10]．これまでにいくつかの解法が提案されており，文献 [1] では整数計画法を用いてこれまでの解法より高速に解けることを示している．また，文献 [5] では制約ソルバーによる解法が示されている．本稿では，高速に解を得られる新たな定式化の方法を提案し，SMT ソルバ Z3 を用いることで，サイズの大きい問題に対して既存の解法よりさらに高速に解けることを示す．

# A Solution to the Slitherlink Puzzle Using SMT Solver

Tetsuo Miyauchi[1,a)]　Kiyofumi Tanaka[1,b)]

***Abstract:*** Slitherlink is one of pencil puzzles and an NP-complete problem[10]. Several solutions for it have been proposed. Literature [1] provides a solution by IP (integer programming) which is faster than the existing ones. Literature [5] shows a solution with using a constraint solver. This paper proposes a way of formulating the problem and shows it can obtain solutions for large-sized problems faster than the technique in [1], by utilizing the SMT solver, Z3.

## 1. はじめに

Slitherlink は，一般に良く知られたペンシルパズルのひとつである．問題に書かれた升の数字に従ってひとつの輪を作ることを目的とする．ひとつの輪は平面を内側の領域と外側の領域に分ける．輪を作ることと内側の領域を求めることは同値であるため，問題に書かれた升の数字に従って Slitherlink の解となる条件を満たす，ひとつの連結で穴の空いていない領域を求めればよい．文献 [1] では，このような連結な領域を整数計画法を用いて求めている．しかしながら，解として複数の連結成分から成る領域が得られる場合がある．得られた解が複数の連結成分から成る場合は，この解が満たさない条件を切除平面法により追加し，複数の連結成分を持たなくなるまで繰り返す必要がある．

本稿で提案する方式では，同様に平面を内側の領域と外側の領域に分けることに着目して定式化を行うが，Slitherlink の解となる条件を満たすこと，ひとつの連結な領域であること，穴が空いた図形でないことを，制約条件として与え，SMT ソルバ Z3 により解を求める．既存の解法と異なり，連結でない解を排除するために繰り返し実行することなく，高速に解を求めることができることを示す．

本稿は次の内容から構成される．まず，Slitherlink パズルについて説明し，次に，Slitherlink の問題と Slitherlink の解となる条件から，SMT ソルバに対して与える制約条件について述べる．いくつかの問題に対して，本解法で解いた結果による計算時間を示し，既存解法と比較する．最後にまとめを述べる．

## 2. Slitherlink とは

Slitherlink とは，以下の条件に従って格子点を線でつないでひとつの輪を作ることを目的とするペンシルパズルである [2]．

(1) 点と点をタテヨコに線でつなげ，全体でひとつの輪をつくる．
(2) 四つの点で作られた正方形 (升と呼ぶ) の中の数字は，その正方形の辺に引く線の数を表す．0 のまわりには

[1] 北陸先端科学技術大学院大学
Japan Advanced Institute of Science and Technology (JAIST)
[a)] t-miyauc@jaist.ac.jp
[b)] kiyofumi@jaist.ac.jp

線は引かれない．数字のない正方形の辺には，何本の線をひいてもよい．

(3) 線は交差したり，枝分かれはしない．

Slitherlink の問題の例を図 1 に示す．

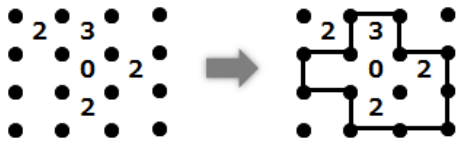



**図 1** Slitherlink の問題例

## 3. 解法

SMT ソルバに対して制約条件を与え，制約条件を充足する解が Sitherlink の解となるように，制約条件を与える．次節以降に制約条件について述べる．本解法では SMT ソルバとして Z3 [3] を用いた．Z3 は Microsoft Research で開発された SMT ソルバである．

### 3.1 解となる変数

N 行 × M 列の問題に対し，問題の外側の領域も含めて $0 \leq i < N+2$, $0 \leq j < M+2$ として，変数の配列 $ans[i][j]$ に対して，$ans[i][j]$ が解となるような制約条件を考える．

Slitherlink の解となる線はひとつの輪となるが，ひとつの輪は，平面を内側と外側の領域に分けるため，i 行, j 列に対応する升が内側の領域となる場合は $ans[i][j]$ の値が正の数，外側の領域のときは，$ans[i][j]$ は 0 となるような制約条件を与え，その制約条件を充足する解を SMT ソルバで求めることを考える．

制約条件を充足する解が得られたとき，i 行，j 列の升を示す変数 $ans[i][j]$ に正の数が入っている場合，この升を内側の領域とした図形を考えると，この領域の境界が求める Slitherlink の解となる．

### 3.2 問題の入力

変数 $problem[i][j]$ が与えられた問題に対する升の数字となるようにする．すなわち，升の数字が 0 から 3 の場合，$problem[i][j]$ の値がそれぞれ 0 から 3 となるようにし，数字が書かれていない升および問題の領域の外側の場合は負となるように制約条件を与える．

### 3.3 各升の数字

問題に書かれた各升の数字 0 から 3 に対して制約条件を与える．

0 が書かれた升に対しては，図 2 に示されるように，Slitherlink の解となる条件から，その升が内側の場合，上下左右の升も内側になる．または，その升が外側の場合，上下左右の升も外側になるため，解としては次の条件が成り立たなければならない．

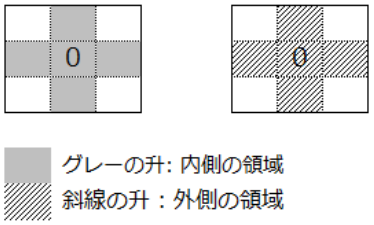



**図 2** 0 が書かれた升

problem[i][j] == 0 ならば，
((ans[i-1][j] == 0) かつ (ans[i][j-1] == 0)
かつ (ans[i][j] == 0) かつ (ans[i][j+1] == 0)
かつ (ans[i+1][j] == 0))
または，
((ans[i-1][j] > 0) かつ (ans[i][j-1] > 0)
かつ (ans[i][j] > 0) かつ (ans[i][j+1] > 0)
かつ (ans[i+1][j] > 0))

この条件を，Z3 の Python API を用いた記法で表すと次のようになる．ここで SIZEROW, SIZECOLUMN はそれぞれ問題の行，列のサイズの値に問題の外側の領域も考えて 2 を加えた値とする．

0 が書かれた升の場合

```
cellnumber0 = [
Or(
 Not(problem[i][j] == 0),
 And(
  (problem[i][j] == 0),
  (ans[i-1][j] == 0), (ans[i][j-1] == 0),
  (ans[i][j] == 0), (ans[i][j+1] == 0),
  (ans[i+1][j] == 0)
 ),
 And(
  (problem[i][j] == 0),
  (ans[i-1][j] > 0), (ans[i][j-1] > 0),
  (ans[i][j] > 0), (ans[i][j+1] > 0),
  (ans[i+1][j] > 0)
 )
)
for i in range(1, SIZEROW-1)
for j in range(1, SIZECOLUMN-1)
]
```

数字 1 が書かれた升の場合図 3 に示されるように，数字 1 が書かれた升が内側か外側か，辺に引かれる線の位置により，8 通りのパタンがある．例えば，図 3 の上段の最も左側のパタンの場合，1 と書かれた升の左側に線があるため，1 と書かれた升が内側の場合，線で接する左側の升は外側となる．1 と書かれた升の周囲には他に線がないため上下，右側の升は，1 とかかれた升と同様に内側の升となる．それ以外の升は，線の引かれ方により，内側にも外側にもなることができる．他のパタンについても同様に考えて，制約条件を与える．Z3 の Python API では．下記のように記述できる．

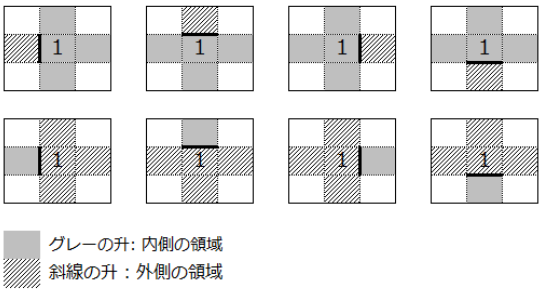



**図 3** 1 が書かれた升

1 が書かれた升の場合

```
cellnumber1 = [
Or(
 Not(problem[i][j] == 1),
 And(
  (ans[i-1][j] > 0), (ans[i][j-1] == 0), (ans[i][j]
   == 0), (ans[i][j+1] == 0), (ans[i+1][j] == 0)
 ),
 And(
  (ans[i-1][j] == 0), (ans[i][j-1] > 0), (ans[i][j]
   == 0), (ans[i][j+1] == 0), (ans[i+1][j] == 0)
  ),
 And(
  (ans[i-1][j] == 0), (ans[i][j-1] == 0), (ans[i][j]
   == 0), (ans[i][j+1] > 0), (ans[i+1][j] == 0)
 ),
 And(
  (ans[i-1][j] == 0), (ans[i][j-1] == 0), (ans[i][j]
   == 0), (ans[i][j+1] == 0), (ans[i+1][j] > 0)
 ),
 And(
  (ans[i-1][j] == 0), (ans[i][j-1] > 0), (ans[i][j]
   > 0), (ans[i][j+1] > 0), (ans[i+1][j] > 0)
 ),
 And(
  (ans[i-1][j] > 0), (ans[i][j-1] == 0), (ans[i][j]
   > 0), (ans[i][j+1] > 0), (ans[i+1][j] > 0)
 ),
 And(
  (ans[i-1][j] > 0), (ans[i][j-1] > 0), (ans[i][j]
   > 0), (ans[i][j+1] == 0), (ans[i+1][j] > 0)
 ),
 And(
  (ans[i-1][j] > 0), (ans[i][j-1] > 0), (ans[i][j]
   > 0), (ans[i][j+1] > 0), (ans[i+1][j] == 0)
 )
)
for i in range(1, SIZEROW-1)
for j in range(1, SIZECOLUMN-1)
]
```

数字 2 が書かれた升についてのパタンは，図 4 のようになり，同様の制約条件を与える．

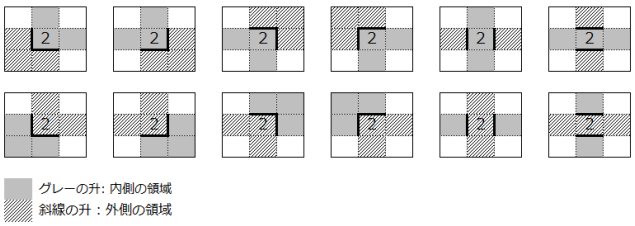



**図 4** 2 が書かれた升

以下に，図 4 の $ans[i][j]$ が内側になるパタンの一部の場合の条件を示す．他のパタンも同様の条件を記述する．

2 が書かれた升の場合

```
cellnumber2 = [
Or(
 Not(problem[i][j] == 2),
 And(
  (ans[i-1][j] > 0),
  (ans[i][j-1] == 0), (ans[i][j] > 0),
  (ans[i][j+1] > 0),
  (ans[i+1][j] == 0), (ans[i+1][j-1] == 0)
 ),
 And(
  (ans[i-1][j] > 0),
  (ans[i][j-1] > 0), (ans[i][j] > 0),
  (ans[i][j+1] == 0), (ans[i+1][j] == 0),
  (ans[i+1][j+1] == 0)
 ),
 And(
  (ans[i-1][j] == 0),
  (ans[i][j-1] > 0), (ans[i][j] > 0),
  (ans[i][j+1] == 0),
  (ans[i+1][j] > 0), (ans[i-1][j+1] == 0)
 ),
 And(
  (ans[i-1][j] == 0),
```

```
  (ans[i][j-1] == 0), (ans[i][j] > 0),
  (ans[i][j+1] > 0),
  (ans[i+1][j] > 0), (ans[i-1][j-1] == 0)
  ),
 And(

 #省略

 )
)
for i in range(1, SIZEROW-1)
for j in range(1, SIZECOLUMN-1)
]
```

数字 3 が書かれた升についてのパタンについても同様に考え，図 5 のようになる．

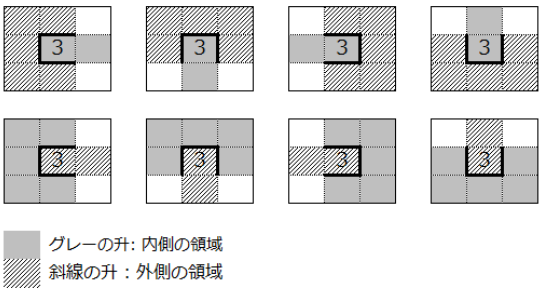



**図 5** 3 が書かれた升

以下に，図 5 の $ans[i][j]$ が内側になるパタンの一部の場合の制約条件を示す．他のパタンも同様の条件を記述する．

3 が書かれた升の場合

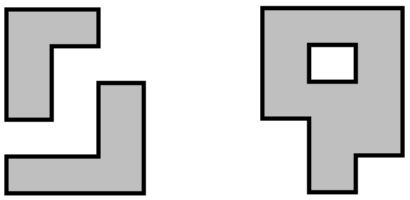
```
cellnumber3 = [
Or(
 Not(problem[i][j] == 3),
 And(
  (ans[i-1][j] == 0),
  (ans[i][j-1] == 0), (ans[i][j] > 0),
  (ans[i][j+1] > 0), (ans[i+1][j] == 0),
  (ans[i-1][j-1] == 0),
  (ans[i+1][j-1] == 0)
 ),
 And(
  (ans[i-1][j] == 0),
  (ans[i][j-1] == 0), (ans[i][j] > 0),
  (ans[i][j+1] == 0),
  (ans[i+1][j] > 0),
  ans[i-1][j-1] == 0), (ans[i-1][j+1] == 0)
 ),
 And(
  (ans[i-1][j] == 0),
  (ans[i][j-1] > 0), (ans[i][j] > 0),
  (ans[i][j+1] == 0),
  (ans[i+1][j] == 0), (ans[i-1][j+1] == 0),
  (ans[i+1][j+1] == 0)
 ),
 And(
  (ans[i-1][j] > 0),
  (ans[i][j-1] == 0), (ans[i][j] > 0),
  (ans[i][j+1] == 0),
  (ans[i+1][j] == 0), (ans[i+1][j-1] == 0),
  (ans[i+1][j+1] == 0)
 ),
 And(

 #省略

 )
for i in range(1, SIZEROW-1)
for j in range(1, SIZECOLUMN-1)
]
```

### 3.4 連結性

上記の制約条件を満たす解が得られても，図 6 のようにその解の示す領域が連結でない, または，連結であっても穴が空いている場合は，その領域を囲む輪が複数になってしまうため，Slitherlink の解にはならない．



(a) 連結でない領域　　(b) 穴が空いている領域

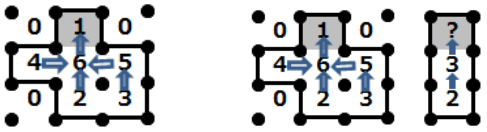
**図 6** 解とならない場合

そのような場合を排除するため，まず，解の示す領域が連結であることを表す制約条件を次のように与える．

まず，問題の作り方として，1 行目に必ずひとつは内側の領域があると仮定してよい．(もし，1 行目にひとつも内側の領域がなければ，N 行からなる問題の 1 行目は不要になってしまい，N-1 行の問題としてよいため，そのような問題の作り方はしないことが暗黙に仮定される.) 従って，解が得られたときに，1 行目の変数 $ans[1][j]$ のうちひとつだけ値が 1 であるとしてよい．また，解を示す領域が連結の場合，内側の升は必ずこの 1 となる升と連結しているため，任意の内側の升をひとつとったときに，隣接する升の変数の値に，この升の変数の値より大きい値か 1 が必ず存在するようにすることができる．(連結でない場合は，1 と

連結していない方の領域においては，その領域内で最大の値をとる変数があり，その升に対しては，上記の条件を満たすことができない．)

図 7 に ans[i][j] の例を示す．(b) の場合 ? の升に上記制約条件を充足する数字を入れることができない．




(a) 連結な領域　　(b) 非連結な領域

**図 7**　ans[i][j] に入る値の例

従って，連結性の判定を行う制約条件は次のように与えられる．

- 1 行目の升に 1 がひとつだけ存在 (すなわち, ans[1][j] に 1 がひとつだけある)
- 2 行目以降には 1 がない
- 外側の領域の升の数字は 0 である
- ans[i][j] で表される升が内側の升，すなわち，ans[i][j] が 1 より大きいとき，隣接する升，ans[i-1][j]，ans[i][j-1]，ans[i][j+1]，ans[i+1][j] の中に ans[i][j] の数字より大きい数，または 1 が存在する

Z3 の Python API の記法で表すと次のようになる．

連結性 (1 行目の升に 1 がひとつだけ存在)

```
exist1 = [
Or(
And((ans[1][1] == 1), Not(ans[1][2] == 1),
Not(ans[1][3] == 1), ...),
And(Not(ans[1][1] == 1), (ans[1][2] == 1),
Not(ans[1][3] == 1), ...),
    ...

    #省略

]
```

連結性 (2 行目以降には 1 がない)

```
notone = [
Not(ans[i][j] == 1)
for i in range(2, SIZEROW-1)
for j in range(1, SIZECOLUMN-1)
]
```

連結性 (外側の領域)

```
brim1 = [
(ans[0][j] == 0) for j in range(SIZECOLUMN)
]

brim2 = [
(ans[i][0] == 0) for i in range(SIZEROW)
]

brim3 = [
(ans[i][SIZECOLUMN-1] == 0) for i in range(SIZEROW)
]

brim4 = [
(ans[SIZEROW-1][j] == 0) for j in range(SIZECOLUMN)
]
```

連結性 (隣接する升の条件)

```
connectchk = [
Implies(
 (ans[i][j] > 1),
 Or(
 (ans[i-1][j] > ans[i][j]),(ans[i][j-1] > ans[i][j]),
 (ans[i][j+1] > ans[i][j]),(ans[i+1][j] > ans[i][j]),
 (ans[i-1][j] == 1),(ans[i][j-1] == 1),
 (ans[i][j+1] == 1),(ans[i+1][j] == 1)
 )
)
for i in range(1, SIZEROW-1)
for j in range(1, SIZECOLUMN-1)
]
```

### 3.5　穴が空いていないこと

連結な領域に穴が空いていないことは，Pick の定理 [4] を満たしているかどうかを調べることでわかる．Pick の定理によれば，格子点を結んで得られる領域が連結であり穴が空いていなければ，その面積は，次の式で求められる．

$S = i + b/2 - 1$

ここで，$S$ はその領域の面積，$b$ はその領域の境界にある格子点の数，$i$ はその領域の内側にある格子点の数を示す．

すなわち，格子点を結んで得られる連結な領域に対して，Pick の定理が成り立たないならば，その領域は穴が空いていることになる．

Slitherlink の解を示す領域は，格子点を結んで得られる領域であり，$S, i, b$ は Slitherlink の解の場合，それぞれ次のようになる．

$S$: 内側となる升の数

$i$: 内側の領域の中にある頂点の数 (その頂点を含む 4 つの升がすべて内側の領域となるような頂点の数を数えればよい．)

$b$: 境界にある辺の数 (Slitherlink の解を示す領域の場合，頂点を垂直方向, または水平方向に結んで得られる領域であるため，境界にある頂点の数は，境界にある辺の数 (垂直方向の辺の数, 水平方向の辺の数の和) と等しくなる．従って，垂直方向または水平方向に，内側の領域となる升と外側の領域となる升が隣り合っている数を数えればよい．)

例えば，図 8 に示すグレーの領域が内側の領域とすると，$S = 6, i = 1, b = 12$ となり，Pick の定理が成り立っていることがわかる．

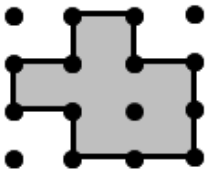


**図 8** 解の領域

### 3.6 線の交差の排除

さらに，図 9 のように線が交差する場合は，Slitherlink の解となる条件に適合しないため，排除するような制約条件を加える．線が交差する場合は，内側の升 (図 9 のグレーの升) と外側の升 (図 9 の斜線の升) が交互に隣り合う市松模様のようなパタンとなるため，そのようなパタンが存在しないことを制約条件として与える．

線の交差の排除

```
checkerpattern1 = [
Not(
 And((ans[i][j] == 0), (ans[i][j+1] > 0),
 (ans[i+1][j] > 0), (ans[i+1][j+1] == 0)
 )
)
for i in range(1, SIZEROW-2)
for j in range(1, SIZECOLUMN-2)
]

checkerpattern2 = [
Not(
 And((ans[i][j] > 0), (ans[i][j+1] == 0),
 (ans[i+1][j] == 0), (ans[i+1][j+1] > 0)
 )
)
for i in range(1, SIZEROW-2)
for j in range(1, SIZECOLUMN-2)
]
```

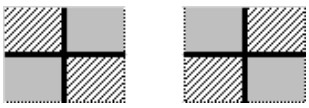


**図 9** 線の交差の排除

### 3.7 SMT ソルバ Z3 による解の探索

前節までで述べた制約条件を SMT ソルバ Z3 に与え，解の探索を行う．解が得られた場合 `ans[i][j]` の値が正の領域が解となり，この変数を表示することにより解が得られる．解を求めるプログラムの全体の構成は以下のようになる．

解の探索

```
#プログラムの開始
starttime0 = time.time() #プログラム開始時の時刻

#制約条件の設定
cellnumber0 = [ ... ]
cellnumber1 = [ ... ]
cellnumber2 = [ ... ]
cellnumber3 = [ ... ]

...

#ソルバの生成
s = Solver()

#ソルバに制約条件を追加
s.add(cellnumber0)
s.add(cellnumber1)
s.add(cellnumber2)
s.add(cellnumber3)

...

#制約条件を充足する解を求める
if s.check() == sat:
 #充足する解があった場合
 endtime = time.time() #解の探索終了時の時刻
 m = s.model()
 #解を表示
 ...

else:
 #充足する解がない場合
 print "Unsat"
 exit()
```

## 4. 評価と考察

表 1 にいくつかの問題に対して本解法を適用した実行時間を示す．実行時間は，本解法によるプログラムの起動から解が出力されるまでの時間 (起動時 `starttime0` から `s.check()` の直後, `endtime` までの時間を Python の time API で取得したもの) を秒単位で示している．

また，比較のため同じサイズの問題に対して文献 [1] および文献 [5] に示される解法での実行時間を既存解法の実行時間として示す．但し，文献 [1] では既存解法で解いた問題が特定できないため，本提案解法で解いた問題と単純な比較はできないが問題のサイズによる計算時間の傾向の比較としては有効と考える．文献 [5] で示される問題に対しては，同じ問題を入力して得られた結果を示している．

実行環境は，Core i5-4210M @ 2.60GHz であり，Z3 の Python API を使用して，Windows7 上の Pyscripter [6] で実行し，それぞれの問題に対して 3 回実行した平均時間である．文献 [1] では，Core i7 3.33GHz の PC，文献 [5] では Let's Note CF-W5 を用いている．

これらの問題は，10×10 サイズの問題については，[2] より，20×30 サイズの問題については，[7] より，14×24, 20×36 サイズの問題については，[8] より取得した (実際の問題は，24×14, 36×20 の問題であり行と列のサイズが逆となっている)．No.17 の問題は，文献 [5] に示される問題である．

**表 1** 評価結果

| No. | 問題サイズ | 提案方式 (sec) | 既存解法 (sec) |
|---|---|---|---|
| 1 | 10×10 | 8.228 | – |
| 2 | 10×10 | 6.366 | – |
| 3 | 10×10 | 7.462 | – |
| 4 | 10×10 | 7.377 | – |
| 5 | 10×10 | 8.327 | – |
| 6 | 10×10 | 8.253 | – |
| 7 | 10×10 | 6.557 | – |
| 8 | 10×10 | 7.164 | – |
| 9 | 10×10 | 6.923 | – |
| | 10×10 の平均 | 0.104 | – |
| 10 | 14×24 | 34.524 | – |
| 11 | 14×24 | 92.612 | – |
| | 14×24 の平均 | 63.568 | 34.937 |
| 12 | 20×30 | 76.386 | – |
| 13 | 20×30 | 40.767 | – |
| 14 | 20×30 | 35.256 | – |
| | 20×30 の平均 | 50.803 | 2372.7 |
| 15 | 20×36 | 50.848 | – |
| 16 | 20×36 | 1272.876 | – |
| | 20×36 の平均 | 661.862 | – |
| 17 | 20×36 | 70.216 | 約 450 |

提案方式は，Z3 のインタフェースとして Python の API を利用している．Python はインタプリタ環境で実行されるためインタプリタ実行によるオーバヘッドがありサイズが小さい問題に対しては，文献 [1] の解法より結果が出力されるまでの時間が大きいが，問題のサイズに対する結果が出力されるまでの時間の増え方が文献 [1] の解法より少なく，20×30 の問題の実行時間 2372.7(sec) と比較して，本提案の解法では，50.803(sec) という十分速い時間で結果を得ることができている．

図 10 に Pyscripter 環境での実行例を示す．`ans[i][j]` が正の領域を `#` で表示し，その領域の境界が求める解となる．

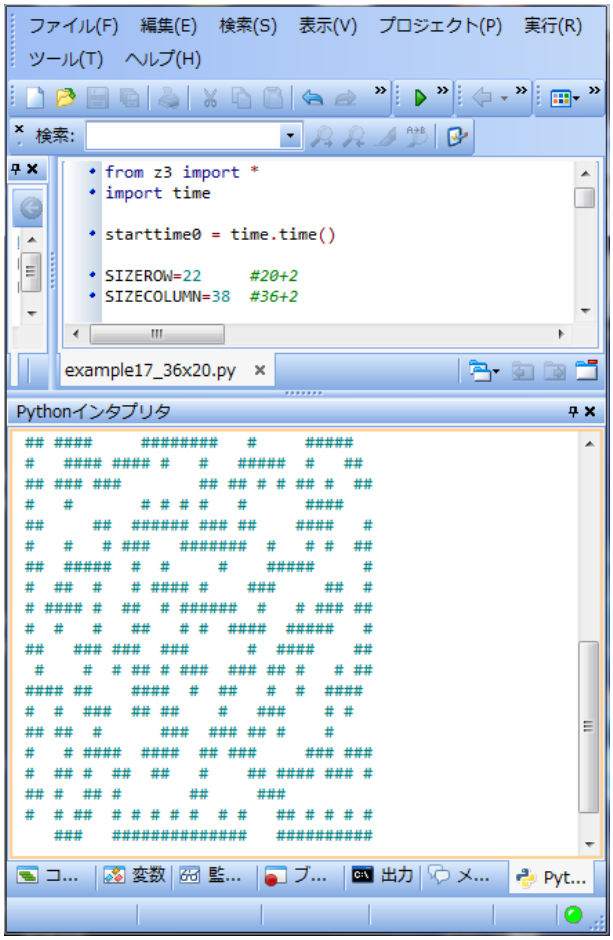



**図 10** 実行例

文献 [5] に示される解法は，Sugar 制約ソルバーに本提案解法同様に制約条件を与えて制約ソルバーで解を求める解法であるが，制約条件として，Slitherlink を構成する線に対する条件を与えるものであり本提案解法とは制約条件の与え方が異なる．評価結果 No.17 に文献 [5] に示された 20×36 のサイズの問題と同じ問題を入力して得られた結果を示す．文献 [5] の解法では 20×36 のサイズの問題に対して約 7 分半かかっているが，本提案解法では同じ問題に対して約 70 秒で解けており，CPU の性能差を考慮しても十

分早く解けていることがわかる．

## 5. 関連研究

Slitherlink はよく知られたパズルであり，アルゴリズム研究の対象としてとりあげられ，上記評価で引用した研究以外にもいくつかの研究結果が報告されている．

文献 [9] では，その解の有無判定問題は NP 完全問題であることが示されている．また，ひとつの解が与えられたときに別解が存在するかを判定する問題も NP 完全問題となることが文献 [10] により示されている．

前述した文献 [1] では整数計画法による解法により既存解法より短い時間で解けることが示されている．文献 [11] では ZDD といわれる二分決定図による解法，文献 [5] では Sugar 制約ソルバーでの解法，文献 [12] では升単位の探索により解く方法と問題作成の手法が示されている．

文献 [13] では Slitherlink の線に対する制約を定式化して解く解法を示しているがこの解法では制約条件の数が非常に大きくなり，Satzoo SAT-solver でエンコードし 10×10 の問題に対して約 5 分かかっている．

## 6. おわりに

本稿では，ペンシルパズル Slitherlink に対して，解が得られるような制約条件を与えて，SMT ソルバ Z3 を使用した解法を示した．特に，制約条件として，得られる領域が連結であること，穴が空いていないことを条件として入力することで，既存解法で，繰り返し計算を行っていた処理を除くことができている．実行環境として，Python インタプリタを用いているため，インタプリタのオーバヘッドがあるが，サイズの大きい問題に対しては既存解法に比べて解が得られる時間が大幅に短縮されており，本解法が有効であることが示された．本解法の他のパズル等への応用を今後は考えてゆきたい．

**謝辞**　本研究の一部は JSPS 科研費 JP 15K00073 の助成を受けて行われた．

### 参考文献

[1] 石濱 友裕, 久野 誉人, ”整数計画法を用いた高速な Slitherlink パズルの解法”, 情報処理学会論文誌, Vol.54, No.8, pp. 2103 - 2108, 2013.
[2] http://www.nikoli.com/ja/puzzles/slitherlink/
[3] https://github.com/z3prover
[4] B. Grunbaum and G. C. Shephard, Pick’s Theorem, The American Mathematical Monthly, Vol. 100, No. 2 (Feb., 1993), pp. 150-161, 1993
[5] 田村直之, “パズルを Sugar 制約ソルバーで解く (オンライン)”, http://bach.istc.kobe-u.ac.jp/sugar/puzzles/slitherlink.html
[6] https://sourceforge.net/projects/pyscripter/
[7] http://www.pro.or.jp/~fuji
[8] フレッシュ スリザーリンク 1, ニコリ, ISBN978-4-89072-303-4
[9] 八登 崇之, “スリザーリンクの NP 完全性について”, 情報処理学会研究会報告, *IPSJ SIG Technical Reports*, Vol.2000, No.84(2000-AL-074), pp.25–31, 2000.
[10] T. Yato and T. Seta, “Complexity and Completeness of Finding Another Solution and Its Application to Puzzles”, *IEICE Trans. Fundamentals of Electronics, Communications and Computer Sciences*, Vol.86-A, pp.1052-1060, 2003
[11] R.Yoshinaka, T.Saitoh, J. Kawahara, K. Tsuruma, H. Iwashita, and S.Minato, “Finding All Solutions and Instances of Numberlink and Slitherlink by ZDDs,” *Algorithms*, Vol.5, No.2, pp.176–213, 2012.
[12] 白井 裕己, 五十嵐 力, 但馬 康宏, 小谷 善行, “スリザーリンク解答システムと問題作成システム”, ゲームプログラミングワークショップ 2006 論文集, pp.32-39, 2006.
[13] S. Herting, “A rule-based approach to the puzzle of Slither Link”, *Univ. Kent, UK, Tech. Rep.*, 2004.