Electronic Preprint for Journal of Information Processing Vol.28

Regular Paper

Solving Slitherlink with FPGA and SMT Solver

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

Received: December 30, 2019, Accepted: September 10, 2020

Abstract: "Slitherlink" is one of popular pencil puzzles. The purpose of the puzzle is to make a link according to the digits written in cells. While determining the existence of a solution to a given puzzle is proved to be an NP-complete class of problems, which means it is difficult to find an effective algorithm to solve the puzzle, solving the puzzle has been studied and there are several previous researches for the puzzle. In this paper, we show two new methods to solve the puzzle. One is with hardware acceleration on an FPGA and the other is based on an SMT solver. We focus on determining inside or outside for each cell instead of making a link and propose a new formulation. With hardware acceleration, it takes 0.1578 seconds on average for solving 10×10 puzzles, and with an SMT solver, our solution is faster than previous researches in most cases.

Keywords: Slitherlink, FPGA, SMT solver, Z3

1. Introduction

"Slitherlink" is one of popular pencil puzzles [16], [17], [19]. The goal of the puzzle is to make one link according to digits written in cells surrounded by lattice points. Deciding whether solutions exist or not is proved to be in the NP-complete class in the literature [12]. There are several previous studies to solve the puzzle, such as the literature [3]. They showed that their method, which applies integer programming, can solve faster than the previous researches.

In solving the puzzle, there is possibility that two or more links are obtained with satisfying the condition of the puzzle. However, from the rule of the puzzle, two or more links cannot be a solution. In the previous research, the solving process is iterated until a solution with a sole link is found, whereas we show our idea to eliminate such kind of candidates for a solution without repeating the solving process with using Pick's theorem [1].

In this paper, we propose two solutions for solving the puzzle. One is with hardware acceleration on an FPGA (Field Programmable Gate Array) and the other is with using an SMT solver. While the goal of Slitherlink is to make a link, as one link divides a plane into two regions, inside and outside, so we focus on determining inside or outside for each cell according to the digits in the puzzle. In the hardware acceleration, checking conditions for a solution of Slitherlink is accelerated. In the solution with an SMT solver, we use Z3 [8] for the SMT solver. We explain our new way for the formulation.

While a main idea for hardware acceleration is from our research in the literature [6] and the formulation for an SMT solver is our proposal from [7], we changed our FPGA platform and evaluated again and the evaluation environment was changed for the SMT solver and we measured the data for this time. We illustrate hardware resources and the execution time with hardware acceleration and execution time comparing with the previous studies.

[1] Japan Advanced Institute of Science and Technology, Nomi, Ishikawa 923–1292, Japan
[a)] t-miyauc@jaist.ac.jp
[b)] kiyofumi@jaist.ac.jp

2. About Slitherlink

2.1 Definition

Slitherlink's goal is to make a link (or a round/circular chain) by drawing lines to connect lattice points, according to the following conditions.

(1) The number of lines drawn around a cell is equal to the digit in the cell (When the digit is 0, there are no lines around the cell. When there is no digit in a cell, any number of lines can be drawn around the cell).
(2) Any line is a part of a link. No intersections or branches are allowed.
(3) A solution includes only one link.

Here, we call a box surrounded by four lattice points "cell". In this paper, when all the above conditions (1) to (3) are satisfied, it is regarded as "satisfying Slitherlink condition". On the other hand, the situation in which the conditions (1) and (2) are satisfied is called "satisfying partial condition" or "candidate". If a situation is not satisfying partial condition, it is called unsatisfying partial condition. The term "attribute" means whether a cell is inside or outside of a link.

Figure 1 shows an example of Slitherlink puzzles. The left figure is a given puzzle, and the right one is a solution.

Examples of wrong solutions are shown in **Fig. 2** and in **Fig. 3**. Figure 2 cannot be a solution since it consists of two disconnected

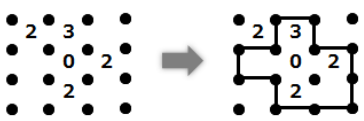


Fig. 1 Example of Slitherlink.

links, which is a violation of the condition (3). The right one is also rejected since it has nested links, which are regarded as two links.

2.2 Strategy for Solving

While the goal of Slitherlink is to make a link, one link divides a plane into two regions, inside and outside. That means to make a link and to determine an inside region and an outside region are equivalent. Therefore, our idea to solve the puzzle is to find one connected region without a hole according to digits in cells of a puzzle instead of making a link.

Whether a cell is in the inside or outside of a link is judged as follows. For example, in the case of **Fig. 4**, there is a cell with no digit between the cell with “3” and the cell with “2”. If the both cells with “3” and “2” have been already decided to be in the inside region, and the lines are to be drawn as in the figure, the resulting situation necessarily leaves the middle cell inside so that it does not contradict the partial condition ((1) and (2) in Section 2.1).

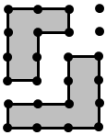


Fig. 2 Two links.

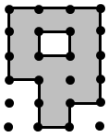


Fig. 3 With a hole.

Fig. 4 Satisfying partial condition.

Fig. 5 Unsatisfying partial condition.

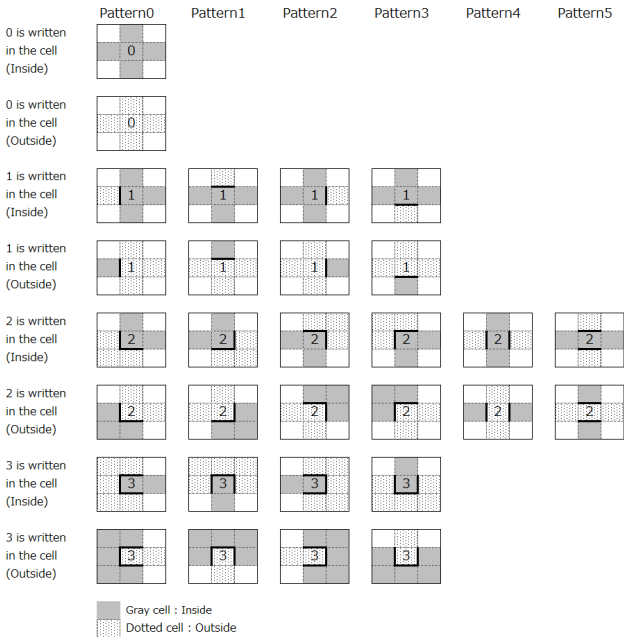


Fig. 6 Patterns corresponding to each digit and attribute.

On the other hand, if the lines are drawn as in **Fig. 5**, from the state of the left cell, the middle cell has to be inside. However, from the state of the right cell, it has to be outside. Therefore, drawing these lines is not allowed to be a solution (This situation eventually leads to a violation of the condition (2)).

Focusing on an inside or outside cell with a digit, the valid combinations of the lines corresponding to its digit with the attributes of the surrounding eight cells are categorized into “patterns”. **Figure 6** lists all of the combinations of attributes and patterns for each possible digit in a cell.

We search for a solution of a puzzle with the steps as follows. First, we select a cell with a digit, then we select a combination of an attribute and a pattern, and check if Slitherlink partial condition is satisfied. If the partial condition is satisfied, we go to the next cell with a digit and we select a combination of an attribute and a pattern, and check if the partial condition is satisfied. We keep on these steps as long as the partial condition is satisfied. If the partial condition is not satisfied, another combination of an attribute and a pattern is checked. If all combinations for the cell violate the partial condition, we go back to the previous cell and check another combination of an attribute and a pattern.

When we finish determining attributes and patterns for all cells with digits, that is a candidate of a solution. While the partial condition is satisfied, we need to reject the possibility of multiple links (See Fig. 2) or a region with a hole (Fig. 3). Confirmation of partial condition, connectivity, and a region without a hole is performed with the hardware we implemented. In the following sections, we explain our method in detail.

3. Hardware Solver

3.1 Overview

We have been studying a soft processor of which instruction-set architecture is MIPS [15] and a system development environment for integrating the processors and hardware accelerators [5]. With using this environment, we designed a hardware solver dedicated to Slitherlink and implemented it on an FPGA. The solver is mapped to a part of memory space for the processor, so that the processor can communicate with the solver by reading from or writing to the specific addresses. **Figure 7** shows the structure of the processor (CPU core) and the solver (Slitherlink Solver). The detailed structure of the Hardware Solver is to be described in Section 3.2.

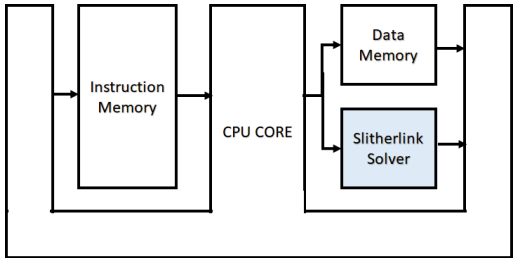


Fig. 7 Organization of the Solver.

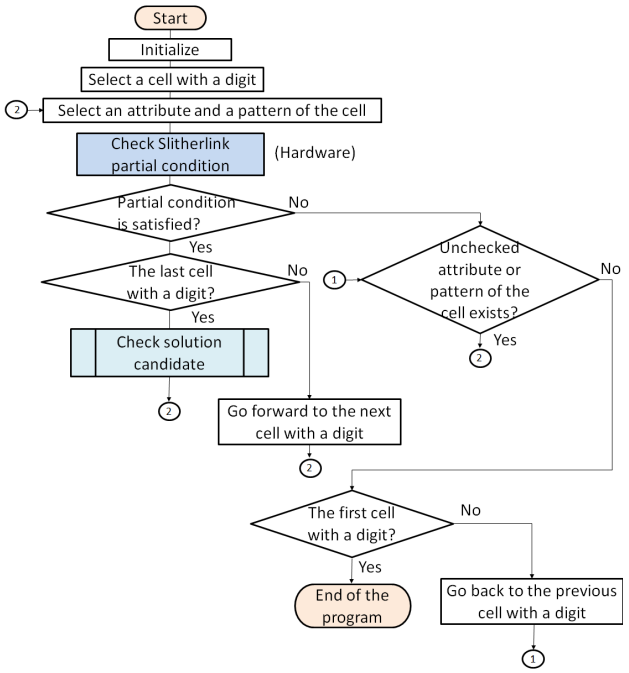


Fig. 8 Algorithm (overall).

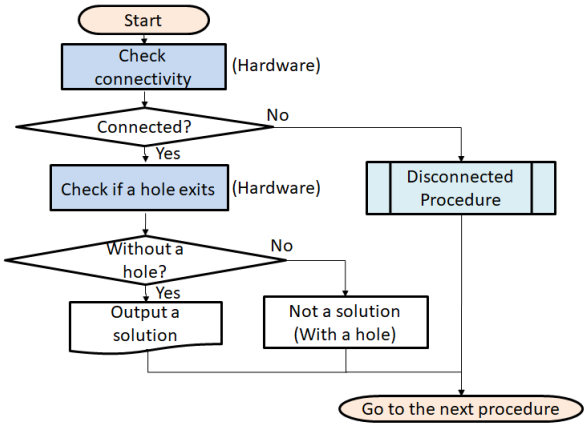


Fig. 9 Checking a solution candidate.

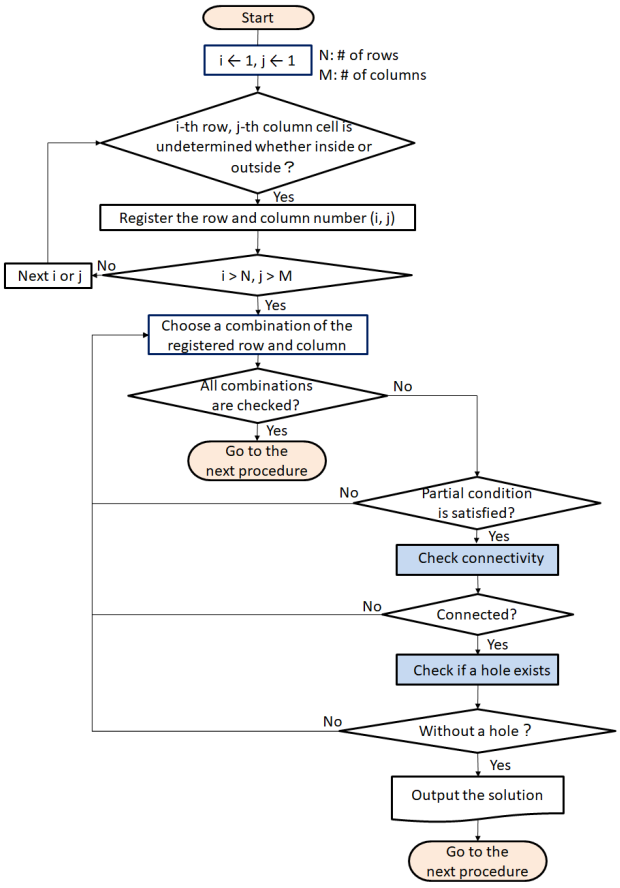


Fig. 10 Disconnected procedure.

3.1.1 Algorithm

Figure 8 shows the overall flow of solving the puzzle with the hardware solver. First, all cells are initialized as "undecided on inside or outside" and one of cells with a digit is picked up. A combination of attribute and pattern is selected and temporarily assigned to the cell. Then, it is checked whether Slitherlink partial condition is satisfied or not. This checking is performed by the solver, which is explained in Section 3.3. If it is satisfied, then it is checked whether all the cells with digits have been traced or not. When there is no cell left, a set of all the attributes and patterns assigned form a candidate for a solution, and then it goes to the process for checking the candidate for a solution, which is described in Section 3.1.3. When there remain cells with digits, it goes to another cell and, after assigning an attribute and pattern, the partial condition checking is repeated.

On the other hand, while scanning cells, once Slitherlink partial condition is not satisfied, it goes to the process for the case of partial condition unsatisfied, where another pattern is selected or it steps back to the previous cell. The details of this process are explained in Section 3.1.2.

3.1.2 Procedure When the Partial Condition is Unsatisfied

As Fig. 8 shows, when Slitherlink partial condition is unsatisfied, it is checked whether there is another unchecked attribute and pattern combination left for the cell. If an unchecked one is found, it returns to the checking loop with it. Otherwise, the current cell is given up and it goes back to the previous cell in order to apply and try another combination of attribute and pattern. When the current cell is the first one in the scanning and it has no unchecked attribute and pattern left, the whole process finishes without a valid solution.

3.1.3 Checking a Candidate for a Solution

Figure 9 shows how to check a candidate for a solution. First, the procedure checks whether the obtained region is connected or not. This connectivity checking is performed by the solver, which is described in Section 3.3.1. After confirming the region is connected, whether the region includes a hole or not is examined based on Pick's theorem [1], which is performed by the solver described in Section 3.3.2. If a hole is found, this candidate turns out not to satisfy Slitherlink condition and therefore, with changing an attribute and pattern combination or returning to the previous cell, it goes to the next iteration. Otherwise, this candidate turns to be a solution. When the region is found not to be connected, it goes to the following "disconnected procedure."

3.1.4 Procedure When the Region is Disconnected

Figure 10 shows the procedure when the region is disconnected. So far, all the cells with digits have been assigned an attribute and a pattern. On the other hand, it is possible that no-

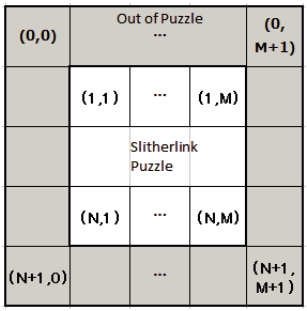


Fig. 11 Framework for Slitherlink.

digit cells which have not been decided whether inside or outside still remain. This process assigns attributes to such cells one by one while checking the connectivity and the hole-less condition. This attempt is repeated until a valid solution is found or all the combinations of attributes for the cells are tried. In Fig. 10, the connectivity checking and the hole checking are done by the solver.

3.2 Structure of Hardware Solver

3.2.1 Hardware Structure for a Given Puzzle

We describe the hardware structure of a solver implemented on an FPGA. For a given Slitherlink puzzle whose size is N × M, (N+2) × (M+2) matrix (**Fig. 11**) is prepared by a combination of register arrays (For simplicity of hardware, M is equal to or less than 32. This is enough since M≤32 is the case in almost all Slitherlink puzzles).

The states of cells are kept in the registers x, $a0$, $a1$, $s0$, and $s1$. x, $a0$ and $a1$ are used for input to the solver and $s0$, and $s1$ are used for storing information generated by the solver. In advance, the processor writes the information about a way of drawing lines (i.e., assigned pattern) for each cell in x. The solver generates suitable attributes in $s0$ and $s1$ for the neighboring cells that do not have digits. The generated attributes are used for checking the consistency of the neighbors' attributes. Each register is prepared as follows.

x: The size of array x is M × 4 bits × (N+2). Every four bits keep a pattern for the corresponding cell. (From Fig. 6, the maximum number of patterns for a digit is six. While 3 bits are enough to express the pattern, 4 bits are reserved to make the implementation simpler.) The initial value is zero. We use the notation x_i_j (4 bits) for a cell in i-th row and j-th column.

$a0$: 32 bits × (N+2) register array. The notation $a0\_i\_j$ (1 bit) corresponds to the information for a cell in i-th row and j-th column. In order to assign an attribute "outside" to a cell, the corresponding bit is set to 1.

$a1$: 32 bits × (N+2) register array. The notation $a1\_i\_j$ (1 bit) corresponds to the information for a cell in i-th row and j-th column. In order to assign an attribute "inside" to a cell, the corresponding bit is set to 1.

$s0$: 32 bits × (N+2) register array. The notation $s0\_i\_j$ (1 bit) corresponds to the information for a cell in i-th row and j-th column. This bit represents that the cell in i-th row and j-th column should be outside. The way to set the bit is described in Section 3.3 in detail.

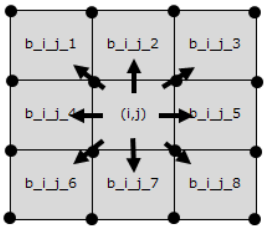


Fig. 12 Around a Cell.

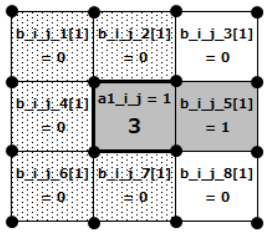


Fig. 13 Suitable Attribute Value b_i_j_*[1].

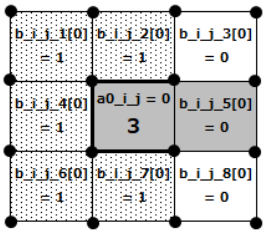


Fig. 14 Suitable attribute value b_i_j_*[0].

$s1$: 32 bits × (N+2) register array. The notation $s1\_i\_j$ (1 bit) corresponds to the information for a cell in i-th row and j-th column. This bit represents that the cell in i-th row and j-th column should be inside. The way to set the bit is described in Section 3.3 in detail.

p: 32 bits × N register array. p_i_j represents j-th bit of $p[i]$. The notation p_i_j corresponds to the i-th row, j-th column cell. This register array is used for checking connectivity in Section 3.3.1.

3.3 Checking Partial Condition

As mentioned above, for each cell, the solver generates suitable attributes for the neighboring cells that do not have digits. As **Fig. 12** shows, there are eight neighboring cells for each cell, from the top left one with the subscript "1" to the bottom right with "8". According to the digit, attribute, and pattern of the central cell, suitable attributes of the neighbors are naturally decided. For example, as in **Fig. 13**, if a cell with the digit of "3" is assigned an attribute of "inside" and a pattern of "Pattern0" (found in Fig. 6), the neighbor cells of 1, 2, 4, 6, and 7 have to be outside while the cell of 5 has to be inside. The cells of 3 and 8 can be either inside or outside.

The suitable attributes are notated as $b\_k[1:0]$, or $b\_i\_j\_k[1:0]$ which explicitly indicates the indexes of the central cell. For example, when the suitable attribute for the left cell is decided as outside, the solver generates $b\_4[1:0] = 01$ as shown in Fig. 13 and **Fig. 14**. Similarly, $b\_5[1:0] = 10$ is generated so that the right cell is judged to be inside. $b\_3[1:0] = 00$ means the attribute for the top right cell is not decided from the state of the

Table 1 *b_k* for out-of-bound cells.

x	[a1:a0]	b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8
X	X	00	00	00	00	00	00	00	00

X: Don't care.

Table 2 *b_k* for cells with no digit.

x	[a1:a0]	b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8
X	X	00	00	00	00	00	00	00	00

X: Don't care.

Table 3 *b_k* for cells with 0.

x	[a1:a0]	b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8
X	00	00	00	00	00	00	00	00	00
X	01	00	01	00	01	01	00	01	00
X	10	00	10	00	10	10	00	10	00

X: Don't care.

Table 4 *b_k* for cells with 1.

x	[a1:a0]	b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8
0	00	00	00	00	00	00	00	00	00
0	01	00	01	00	10	01	00	01	00
0	10	00	10	00	10	10	00	10	00
1	00	00	00	00	00	00	00	00	00
1	01	00	10	00	01	01	00	01	00
1	10	00	01	00	10	10	00	10	00
2	00	00	00	00	00	00	00	00	00
2	01	00	01	00	01	00	10	01	00
2	10	00	10	00	10	00	01	10	00
3	00	00	00	00	00	00	00	00	00
3	01	00	01	00	01	01	00	10	00
3	10	00	10	00	10	10	00	01	00

Table 5 *b_k* for cells with 2.

x	[a1:a0]	b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8
0	00	00	00	00	00	00	00	00	00
0	01	00	01	00	10	01	10	10	00
0	10	00	10	00	01	10	01	01	00
1	00	00	00	00	00	00	00	00	00
1	01	00	01	00	01	10	00	10	10
1	10	00	10	00	10	01	00	01	01
2	00	00	00	00	00	00	00	00	00
2	01	00	10	10	01	10	00	01	00
2	10	00	10	10	10	10	00	10	00
3	00	00	00	00	00	00	00	00	00
3	01	10	10	00	10	01	00	01	00
3	10	01	01	00	01	10	00	10	00
4	00	00	00	00	00	00	00	00	00
4	01	00	01	00	10	10	00	01	00
4	10	00	10	00	01	01	00	10	00
5	00	00	00	00	00	00	00	00	00
5	01	00	10	00	01	01	00	10	00
5	10	00	01	00	10	10	00	01	00

central cell.

For each digit in a cell, the combination of pattern (x) and attribute ($a1$, $a0$) decides the value of $b\_k[1:0]$. **Table 1**, **Table 2**, **Table 3**, **Table 4**, **Table 5** and **Table 6** show all possible values for $b\_k[1:0]$. The solver includes the logic circuit to generate $b\_k[1:0]$ for all cells.

Whether each cell satisfies Slitherlink partial condition is judged by referring to *b_k* generated by the neighboring cells.

For a cell in i-th row, j-th column, $s0\_i\_j$ is set by:

$$
\begin{aligned}
& b\_(i-1)\_(j-1)\_8[0] \text{ or } b\_(i-1)\_j\_7[0] \\
\text{or } & b\_(i-1)\_(j+1)\_6[0] \\
\text{or } & b\_i\_(j-1)\_5[0] \text{ or } b\_i\_(j+1)\_4[0] \\
\text{or } & b\_(i+1)\_(j-1)\_3[0] \text{ or } b\_(i+1)\_j\_2[0] \\
\text{or } & b\_(i+1)\_(j+1)\_1[0].
\end{aligned}
$$

Table 6 *b_k* for cells with 3.

x	[a1:a0]	b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8
0	00	00	00	00	00	00	00	00	00
0	01	10	10	00	10	01	10	10	00
0	10	01	01	00	01	10	01	01	00
1	00	00	00	00	00	00	00	00	00
1	01	10	10	10	10	10	00	01	00
1	10	01	01	01	01	01	00	10	00
2	00	00	00	00	00	00	00	00	00
2	01	00	10	10	01	10	00	10	10
2	10	00	01	01	10	01	00	01	01
3	00	00	00	00	00	00	00	00	00
3	01	00	01	00	10	10	10	10	10
3	10	00	10	00	01	01	01	01	01

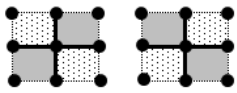


Fig. 15 Checker pattern.

If $s0\_i\_j = 1$, this cell is required to be outside by the neighbor cells. Similarly, $s1\_i\_j$ is set by:

$$
\begin{aligned}
& b\_(i-1)\_(j-1)\_8[1] \text{ or } b\_(i-1)\_j\_7[1] \\
\text{or } & b\_(i-1)\_(j+1)\_6[1] \\
\text{or } & b\_i\_(j-1)\_5[1] \text{ or } b\_i\_(j+1)\_4[1] \\
\text{or } & b\_(i+1)\_(j-1)\_3[1] \text{ or } b\_(i+1)\_j\_2[1] \\
\text{or } & b\_(i+1)\_(j+1)\_1[1].
\end{aligned}
$$

If $s1\_i\_j = 1$, this cell is required to be inside by the neighbor cells. If both $s0\_i\_j$ and $s1\_i\_j$ are 1, this cell is required to be inside and outside at the same time, which means this cell does not satisfy Slitherlink partial condition. When this kind of cell exists, the current assignments of attributes and patterns cannot lead to a solution of the puzzle. On the other hand, if both are 0, the current assignments may be valid as a candidate of a solution.

Additionally, checker patterns as in **Fig. 15** are not allowed for being a solution due to intersected lines. (The gray cells and the dotted cells represent inside and outside, respectively.) That is, if there exists i and j such that:

$$
\begin{aligned}
& (\; s1\_i\_j \;\; \text{xor} \;\; s1\_(i+1)\_(j+1) \;) \text{ or} \\
& (\; s1\_i\_(j+1) \;\; \text{xor} \;\; s1\_(i+1)\_j \;) \text{ or} \\
\text{not } & (\; s1\_i\_j \;\; \text{xor} \;\; s1\_i\_(j+1) \;) \quad \text{or} \\
& (\; s0\_i\_j \;\; \text{xor} \;\; s0\_(i+1)\_(j+1) \;) \text{ or} \\
& (\; s0\_i\_(j+1) \;\; \text{xor} \;\; s0\_(i+1)\_j \;) \text{ or} \\
\text{not } & (\; s0\_i\_j \;\; \text{xor} \;\; s0\_i\_(j+1) \;) \quad \text{or} \\
\text{not } & (\; s1\_i\_j \;\; \text{xor} \;\; s0\_i\_j \;)
\end{aligned}
$$

is 0, the current assignments are rejected.

3.3.1 Checking Connectivity

Connectivity of a region for given assignments is judged as follows. In the checking process, if $p\_i\_j = 1$, this cell is regarded as a "connected cell". The following procedure is performed.

1. At the first step, an "inside" cell in the 1st (i=1) row is selected. (If there is no inside cell in the 1st row, the assignment should be N-1 rows instead of N rows, so we can assume implicitly there is at least one inside cell in the 1st row. If there are two or more inside cells in the row, the right most

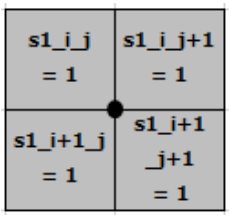


Fig. 16 Surrounded by inside cells.

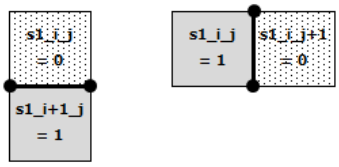


Fig. 17 Lines between inside and outside cells.

one is selected.) p_1_j of the selected cell is set to 1.

2. For a cell of i-th row and j-th column, if the cell is inside, it is examined whether there is an adjacent cell which is connected. If it is found, p_i_j of the cell is set to 1 in the following way: p_i_j or (($p\_(i-1)\_j$ or $p\_i\_(j-1)$ or $p\_i\_(j+1)$ or $p\_(i+1)\_j$) and $s1\_i\_j$), for all i and j. It goes to the third step.
3. The value of p_i_j is summed up to obtain the number of 1s in p. If the number of 1s is greater than that in the previous clock cycle, it returns to the second step in the next clock cycle. Otherwise, it goes to the final step.
4. The number of 1s in p is compared to the number of 1s in $s1\_i\_j$. If they are equal, it turns out that the region is connected. Otherwise, there are two or more disconnected regions.

The number of 1s in 32-bit register $p[i]$ can be acquired within one clock cycle by using the following formula [11]. In the following, p5 is the result.

```
p1 = (p[i]  & 0x55555555) + ((p[i]  >> 1) & 0x55555555);
p2 = (p1 & 0x33333333) + ((p1 >> 2) & 0x33333333);
p3 = (p2 & 0x0f0f0f0f) + ((p2 >> 4) & 0x0f0f0f0f);
p4 = (p3 & 0x00ff00ff) + ((p3 >> 8) & 0x00ff00ff);
p5 = (p4 & 0x0000ffff) + ((p4 >>16) & 0x0000ffff);
```

The sum of $p[1]$ to $p[N]$ gives the size of the connected region in the procedure above.

3.3.2 Checking a Hole

Using Pick's theorem [1], the size "S" of a connected polygon without a hole, where every vertex is at a lattice point, is given as follows.

$S = i + b/2 - 1$

- i : The number of vertices which are inside the polygon
- b : The number of vertices which are on the boundary of the polygon

In the case of Slitherlink, as **Fig. 16** shows, i is equal to the number of vertices surrounded by inside cells. b is equal to the number of lines which are drawn between an inside cell and an outside cell as **Fig. 17**.

To count i, we assign a wire $wi\_i\_(j+1)$ to ($s1\_i\_j$ and $s1\_i\_(j+1)$ and $s1\_(i+1)\_j$ and $s1\_(i+1)\_(j+1)$), then count 1s of wi_i_j with using the formula in the previous subsection. Similarly, to count b, we assign a wire $wbhorizontal\_i\_j$ to ($s1\_i\_j$ xor $s1\_i\_(j+1)$), and $wbvertical\_i\_j$ to ($s1\_i\_j$ xor $s1\_(i+1)\_j$), then count 1s of $wbhorizontal\_i\_j$ and $wbvertical\_i\_j$ with using the formula in the previous subsection. As the size S is obtained in the previous subsection with counting the number of 1s of $s1\_i\_j$, S is compared with the right hand of the Pick's theorem, so that when they are equal, the region turns out not to include a hole.

4. Solution with SMT Solver

In the following subsections, we show another method for solving Slitherlink with using an SMT Solver. To solve the puzzle, we give constraints to the SMT solver with formulation so that the solution which satisfies the constraints becomes the desired one. We explain how to formulate the constraints in detail. For SMT solver, we used Z3 [8], which was developed by Microsoft Research.

4.1 Variables

For an instance of the puzzle of N × M matrix, we consider (N+2) × (M+2) matrix including the outside of the puzzle. We consider an integer variable array $ans[i][j]$ which shows a solution of the puzzle, while we assign constraints to the array. As we illustrated in Section 2.2, a link divides an area into two regions. We give constraints for the SMT solver that $ans[i][j]$ shows "inside" if $ans[i][j] > 0$ for a cell in i-th row and j-th column. When a solution satisfying the constraints is given, the boundary of the inside region should be the desired solution, or the resulting link, for the given puzzle. In addition, we set a variable array element $problem[i][j]$ for a cell in i-th row and j-th column as follows.

$$problem[i][j] = \begin{cases} n & \text{(if there is a digit in the cell)} \\ -1 & \text{(no digit in the cell)} \\ -2 & \text{(out of the cell)} \end{cases} \quad (1)$$

n : the digit in i-th row, j-th column

4.2 Constraints for the Partial Condition

In this section, we describe how the constraints are given to satisfy the Slitherlink partial condition.

For a cell with a digit "0", as shown in Fig. 6, if the cell is inside, the upper, left, right and lower cells have to be also inside, and if the cell is outside, the upper, left, right and lower cells have to be also outside, that is, if $problem[i][j] == 0$, then

```
(   (ans[i-1][j] == 0) and (ans[i][j-1] == 0)
and (ans[i][j] == 0) and (ans[i][j+1] == 0)
and (ans[i+1][j] == 0))
or  (   (ans[i-1][j] > 0) and (ans[i][j-1] > 0)
and (ans[i][j] > 0) and (ans[i][j+1] > 0)
and (ans[i+1][j] > 0))
```

This condition is described in Z3 with Python API as follows. Here, `SIZEROW` and `SIZECOLUMN` are the sizes of the row and the column including the outside of the puzzle , respectively.

0 is written in a cell

```
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)
]
```

In the case of a cell in which 1 is written, as shown in Fig. 6, there are eight combinations of patterns and attributes, of which constraints are given as follows.

1 is written in a cell

```
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)
]
```

Similarly, in the case of a cell in which 2 is written, as shown in Fig. 6, there are twelve combinations of patterns and attributes, of which constraints are given as follows (For the limitation of pages, we show a part of the code, but the omitted ones would be easily constructed).

2 is written in a cell

```
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(

 #Omit other cases

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

Similarly, the case of a cell in which 3 is written is shown as follows.

3 is written in a cell

```
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(

 #Omit other cases

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

4.3 Rejecting Checker Pattern

We add constraints so that checker patterns such as in Fig. 15 are rejected. These constraints are described in Z3 with Python API as follows.

Reject Checker patterns

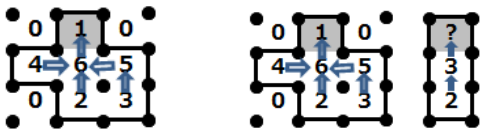
```
checkerpattern1 = [
Not(
 And((ans[i][j] == 0), (ans[i][j+1] > 0),
 (ans[i+1][j] > 0), (ans[i+1][j+1] == 0)
 )
)
for i in range(1, SIZEROW-2)
for j in range(1, SIZECOLUMN-2)
]

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)
]
```

4.4 Checking Connectivity

Constraints for checking connectivity are provided as follows. We assume there is at least one inside cell in the 1st row as described in Section 3.3.1. In addition, we assign a constraint that only one element among *ans*[1][*j*] for j = 1 to M is "1". When the region is connected, all inside cells are connected to the cell of which digit is 1. Therefore, we can assign to all inside cells, if connected, *ans* values as follows. Any inside cell, *c*, has at least one adjacent cell which has an *ans* value greater than *c*'s *ans* or equal to 1.

Figure 18 shows an example of the values of *ans*[*i*][*j*]. As shown in Fig. 18 (a), any inside cell has an adjacent cell of which value of *ans* is greater than the cell's *ans* value or equal to 1. Whereas, Fig. 18 (b) illustrates the cell of "?" cannot be assigned a value which satisfies the constraint above.



(a) Connected (b) Not Connected

Fig. 18 Example values of ans[i][j].

In summary, we give constraints for connectivity as follows. Here, the value means the cell's *ans* value.

(1) Only one cell in the 1st row has the value of 1.
(2) Any cell from the 2nd row to N-th row does not have the value of 1.
(3) Outside of the puzzle has the value of 0.
(4) When a cell is inside, there is an adjacent cell of which value is greater or 1.

These constraints are described in Z3 with Python API as follows.

Constraint No.1

```
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), ...),
    ...

    #Omit other cases

]
```

Constraint No.2

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

Constraint No.3

```
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)
]
```

Constraint No.4

```
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)
]
```

4.5 Checking a Hole

As we explained in Section 3.3.2, satisfying Pick's theorem implies the region does not have a hole. "S", the size of the region, is the number of cells of which the value of *ans* is positive. "i", the number of vertices which are inside the region, is obtained by counting the number of vertices surrounded by inside cells. "b", the number of vertices which are on the boundary of the region, is the number of horizontal edges and vertical edges. The number of vertical edges are equal to the number of pairs of which the left cell is inside and the right cell is outside, or the left cell is outside and the right cell is inside. Similarly, the number of horizontal edges are equal to the number of pairs of which the upper cell is inside and the lower cell is outside, or the upper cell is outside and the lower cell is inside. We add the constraint that S, i, and b satisfy the formula of Pick's theorem. This constraint is described in Z3 with Python API as follows.

Checking a Hole

```
#Count the number of inside cells
for i in range(0, SIZEROW):
  for j in range(0, SIZECOLUMN):
    s.add(If(ans[i][j] > 0, incell[i*SIZECOLUMN+j] == 1,
incell[i*SIZECOLUMN+j] == 0))
s.add(n_inside == Sum(incell))

#Count the number of boundary edges
#vertical
for i in range(1, SIZEROW-1):
  for j in range(0, SIZECOLUMN-1):
    s.add(If(Or(And((ans[i][j] <= 0), (ans[i][j+1] > 0)),
And((ans[i][j] > 0), (ans[i][j+1] <= 0))),
  edgev[i][j] == 1, edgev[i][j] == 0))
s.add(countedgev[0] == 0)
for i in range(1, SIZEROW-1):
  s.add(countedgev[i] == Sum(edgev[i]))
s.add(n_edgev == Sum(countedgev))

#horizontal
for j in range(1, SIZECOLUMN-1):
  for i in range(0, SIZEROW-1):
    s.add(If(Or(And((ans[i][j] <= 0), (ans[i+1][j] > 0)),
And((ans[i][j] > 0), (ans[i+1][j] <= 0))),
      edgeh[i][j] == 1, edgeh[i][j] == 0))
for i in range(0, SIZEROW-1):
  s.add(edgeh[i][0] == 0)
for i in range(0, SIZEROW-1):
  s.add(countedgeh[i] == Sum(edgeh[i]))
s.add(n_edgeh == Sum(countedgeh))

#n_edge
s.add(n_edge == (n_edgev + n_edgeh))

#Count internal vertices
for i in range(1, SIZEROW-2):
  for j in range(1, SIZECOLUMN-2):
    s.add(If(And((ans[i][j] > 0), (ans[i][j+1] > 0),
      (ans[i+1][j] > 0), (ans[i+1][j+1] > 0)),
      countinternal[(i-1)*(SIZECOLUMN-3)+(j-1)] == 1,
      countinternal[(i-1)*(SIZECOLUMN-3)+(j-1)] == 0))
s.add(n_internal == Sum(countinternal))

#Confirm Pick's theorem
#n_iside = n_internal + n_edge/2 -1
s.add((n_inside + n_inside)
  == n_internal + n_internal + n_edge - 2)
```

4.6 Solving the puzzle

With adding the constraints explained in the previous sections to the SMT solver, Z3, we solve the puzzle. The procedure to solve the puzzle is as follows.

(1) Formulate the constraints described in the previous sections.
(2) Create the solver by invoking the API.
(3) Add the constraints to the solver.
(4) If a solution is found, the solution is printed out, otherwise, the fact that it is unsatisfied is printed out.

5. Evaluation

We show the evaluation result for solving Slitherlink with the hardware solver and SMT solver. We implemented the hardware solver on Xilinx Artix-7 FPGA(XC7A35T) with Xilinx Vivado 2018.3. The solver runs with a soft-processor of which architecture is MIPS instruction set. The running clock frequency is 50MHz. GCC 4.3.3 is used for the compiler. **Table 7** shows the resource usage in the FPGA device (with utilization rate in the device).

The puzzles from No.1 to No.3 are from Ref. [17]. These three puzzles are the same ones as those in the literature [14]. The puzzles from No.4 to No.9 are chosen from Ref. [19] randomly. "Puzzle Size" is the size of the puzzle (the number of rows × the number of columns).

Table 8 shows execution times for the same puzzles as in Table 7 by the hardware solver, the SMT solver, and a previous research [3], [14] for comparison. In the literature [3], the authors formulate the conditions for the puzzles and use integer programming. The table includes, for the previous research, the average execution time only since they did not designate the puzzles for each size. The SMT solver, Z3, runs on a PC with Core i7-6500U CPU @ 2.50 GHz and RAM of 8 GBytes. For the hardware solver, the time is from the reset of the processor to the acquisition of the solution. For the SMT solver, the time is from the beginning of the program to the acquisition of the solution.

Table 7 FPGA Resource Usage.

No	Puzzle Size	# of LUTs	# of FFs
1 [17]	10x10	3941(19.0%)	1542(3.7%)
2 [17]	10x10	4040(19.4%)	1542(3.7%)
3 [17]	10x10	4057(19.5%)	1566(3.8%)
4 [19]	10x10	4144(19.9%)	1594(3.8%)
5 [19]	10x10	4301(20.7%)	1602(3.9%)
6 [19]	10x10	4308(20.7%)	1606(3.9%)
7 [19]	10x10	4057(19.5%)	1566(3.8%)
8 [19]	10x10	4175(20.1%)	1590(3.8%)
9 [19]	10x10	4074(19.6%)	1562(3.8%)

Table 8 Execution Time (sec).

No.	Puzzle Size	SMT Solver	Hardware	Previous Research
1 [17]	10x10	3.835	0.0011	0.116 [14]
2 [17]	10x10	3.850	0.4172	0.384 [14]
3 [17]	10x10	3.878	0.0014	0.060 [14]
4 [19]	10x10	5.536	0.0251	–
5 [19]	10x10	5.357	0.5234	–
6 [19]	10x10	5.407	0.0621	–
7 [19]	10x10	6.294	0.2214	–
8 [19]	10x10	5.510	0.1548	–
9 [19]	10x10	4.225	0.0142	–
Average Time	10 × 10	4.877	0.1578	0.104 [3]

Table 9 Execution Time (sec).

No.	Puzzle Size	SMT Solver	Previous Researches
10 [16]	14 × 24	12.251	–
11 [16]	14 × 24	28.978	–
	14 × 24 average	20.614	34.937 [3]
12 [18]	20 × 30	27.805	–
13 [18]	20 × 30	23.552	–
14 [18]	20 × 30	28.162	–
	20 × 30 average	26.506	2372.7 [3]
15 [16]	20 × 36	26.200	–
16 [16]	20 × 36	1227.437	–
	20 × 36 average	626.818	–
17 [20]	20 × 36	26.619	450 [20]
18 [17]	20 × 36	28.059	26.286 [14]

The unit is seconds. This result shows the hardware solver is over 30 times faster on average than the SMT solver, while the CPU in the PC is 50 times faster than the CPU on the FPGA. In addition, the average execution time with the hardware solver and that with the previous research are almost same.

Table 9 shows the execution times for different sizes of puzzles by the SMT solver and other works [3], [14], [20]. No.10 and No.11 are the same as No.41 and No.73 in the literature [16], respectively. No.12 to No.14 are No.31 to No.33 in the literature [18]. For the size of 20 × 36, No.15 and No.16 are No.74 and No.100 in the literature [16], respectively. In addition, we used the same puzzle as that in the literature [20]. In the literature [3] and [20], they use Core i7 3.33 GHz and Let's Note CF-W5, respectively. No.18 is in the literature [17]. We evaluated with the same puzzle as in the literature [14], which is a little faster than ours, but their processor is AMD Opteron Processor 8393, 3.09 GHz with 512 GBytes memory, whereas our processor is Core i7 2.50 GHz with 8 GBytes memory.

We used the SMT solver, Z3, whose interface is Python API. Since Python works in interpreter environment, there is a relatively large overhead of the interpreter, so that execution time is longer than the previous research for the size of 10×10. On the other hand, puzzles with larger sizes take a relatively long time to solve so that the interpreter overhead is hidden. We can conclude that the proposed method is faster for puzzles of large sizes.

6. Related Work

Slitherlink has been studied and there are several related works. In the literature [12], it is proved that deciding whether solutions exist or not belongs to a NP-complete class. Additionally, it is proved in the literature [13] that when a solution is found, deciding whether another solution exists or not is NP-complete. In the literature [3], they showed the puzzle can be formulated by using integer programming and solved faster than the existing formulation. In contrast, we proposed new two methods to solve the puzzle, one of which is using hardware and the other of which is using the SMT solver, and revealed that the puzzle can be solved faster than the previous methods. In Ref. [14], a solution with a zero-suppressed binary decision diagram (ZDD) is described. We used the same puzzle as that used in the literature and we can conclude that our method is faster considering the performance of the processors used. In the literature [20], the solution with Sugar (SAT-based Constraint Solver) is explained. We showed our approach can solve the same puzzle faster than the solution in the literature [20]. While the literature [2] describes methods for applying a ruleset to solve the puzzle, the number of conditions becomes large, so it takes five minutes for a puzzle of the size of 10×10. The literature [9] describes the way to solve the puzzle by searching all cells and checking Slitherlink partial condition, but they do not deal with the problem of eliminating the case of multiple links.

There are several studies to solve other types of puzzles with using FPGA. As for "Sudoku" puzzle, the literature [4], [10] reported an application of FPGA to Sudoku puzzles.

7. Conclusion

This paper has proposed two methods to solve Slitherlink. One is with hardware acceleration on an FPGA and the other is with using an SMT solver, Z3.

We focused on determining inside or outside for each cell instead of making a link. To confirm one link, we showed our methods for checking connectivity for hardware acceleration and a solution with SMT solver, respectively. In addition, we explained a way to confirm a region without a hole with Pick's theorem.

We applied our solution to several puzzles. With hardware acceleration, it takes 0.1578 seconds on average for solving 10 × 10 puzzles, and with an SMT solver, our solution is faster than previous researches in most cases.

Our approach for hardware acceleration and the way to determine whether a region is connected or has a hole can be used for solving other problems. Applying these methods to other problems remains for our future work.

Acknowledgments This work is partly supported by JSPS KAKENHI Grant Number 19K11873.

References

[1] Funkenbusch, W.W.: From Euler's formula to Pick's formula using an edge theorem, *The American Mathematical Monthly*, Vol.81, No.6, pp.647–648 (1974).
[2] Herting, S.: A rule-based approach to the puzzle of Slither Link, University of Kent, Technical Report (2004).
[3] Ishihama, T. and Kuno, T.: A Faster Solution to the Slitherlink Puzzle Using Integer Programming, *IPSJ Journal*, Vol.54, No.8, pp.2103–2108 (2013) (in Japanese).
[4] Malakonakis, P., Smerdis, M., Sotiriades, E. and Dollas, A.: An FPGA-Based Sudoku Solver based on Simulated Annealing Methods, *Proc. Intl. Conf. Field-Programmable Technology*, pp.522–525 (2009).
[5] Miyauchi, T. and Tanaka, K.: Building a Framework for an Application-Adaptive Processor System on FPGA-based SoC, *Proc. 21st Workshop on Synthesis And System Integration of Mixed Information Technologies*, pp.359–364 (2018).
[6] Miyauchi, T. and Tanaka, K.: A Solution to Slitherlink Puzzles Using FPGA, *Proc. 32nd International Conference on Computers and Their Applications*, pp.77–83 (2017).
[7] Miyauchi, T. and Tanaka, K.: A Solution to the Slitherlink Puzzle Using SMT Solver, *Proc. 22nd Game Programming Workshop 2017*, pp.111–118 (2017) (in Japanese).
[8] Moura, L. and Bjørner, N.: Z3: An Efficient SMT Solver, *Proc. International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp.337–340 (2008).
[9] Shirai, H., Igarashi, C., Tajima, Y. and Kotani, Y.: Solving and Making Problems of Slither Link, *Proc. Game Programming Workshop*, pp.32–39 (2006) (in Japanese).
[10] Skliarova, I., Vallejo, T. and Sklyarov, V.: Solving Sudoku in Reconfigurable Hardware, *Proc. 8th Intl. Conf. Computing and Networking Technology* (*ICCNT*), pp.10–15 (2012).
[11] Warren, Jr. H.S.: *Hacker's Delight*, pp.65–66, Addison-Wesley (2003).

[12] Yato, T.: On the NP-completeness of the Slither Link Puzzle, *IPSJ SIG Technical Reports*, Vol.2000, No.84(2000-AL-074), pp.25–31 (2000) (in Japanese).
[13] Yato, T. and Seta, T.: 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).
[14] Yoshinaka, R., Saitoh, T., Kawahara, J., Tsuruma, K., Iwashita, H. and Minato, S.: Finding All Solutions and Instances of Numberlink and Slitherlink by ZDDs, *Algorithms*, Vol.5, No.2, pp.176–213 (2012).
[15] MIPS® Architecture For Programmers, Volume II-A: The MIPS32® Instruction Set (2013).
[16] Fresh Slitherlink 1, Nikoli, ISBN978-4-89072-303-4 (in Japanese).
[17] Slitherlink 1, Nikoli, ISBN4-89072-024-3 (in Japanese).
[18] available from ⟨http://www.pro.or.jp/˜fuji/java/puzzle/numline/index.html⟩
[19] available from ⟨http://www.puzzle-loop.com/⟩
[20] available from ⟨http://bach.istc.kobe-u.ac.jp/sugar/puzzles/slitherlink.html⟩

Tetsuo Miyauchi received his M.S. degree from Japan Advanced Institute of Science and Technology in 2015. His research interests are processor architecture, reconfigurable systems, and real-time embedded systems. He is a member of the IPSJ.

Kiyofumi Tanaka received his B.S., M.S., and Ph.D. degrees from the University of Tokyo in 1995, 1997, and 2000, respectively. His research interests are computer architecture, operating systems, and real-time embedded systems. He is a member of the IEEE, ACM, IEICE, and IPSJ.