A rule-based approach to the puzzle of Slither Link

Stefan Herting
University of Kent
sh97@kent.ac.uk

ABSTRACT

The puzzle of Slither Link, proved to be NP-complete, has a large number of local rules. This paper will present an approach to Slither Link based on precomputable rules and which is transferable to other problems. It will show how a ruleset can be constructed automatically and the problem constraints implicitly encoded into it. This ruleset will then be refined to only contain non-redundant rules. The paper will describe methods for applying a ruleset to a problem instance. In order to solve the general case, SAT-solving and Constraint Logic Programming have been explored.

1. INTRODUCTION

Slither Link, also called Number Line, is a Japanese game which is played on a rectangular grid. Cells in this grid are either empty or labelled with numbers between 0 and 3. The grid is made up of edges and every cell is surrounded by four of these. The player's task is to draw a single cycle without crossings along these edges, where the numbers in the cells determine how many of the four surrounding edges have to be used. A small instance of Slither Link and its solution can be seen in Figure 1.

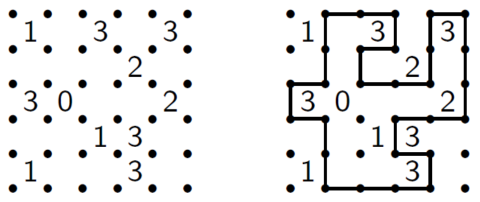

Fig. 1. Instance of Slither Link and its solution (taken from [Yat00])

2. BACKGROUND

Slither Link has been used as an example problem for complexity-theoretic concepts. Yato proved the NP-Completeness of Slither Link by reducing the Hamiltonian Path Problem with respect to certain restricted graphs to Slither Link [Yat00][1]. Yato later proved that the Another Solution Problem is NP-hard for Slither Link [Yat03]. The Another Solution problem consists of deciding, given one solution, if a problem instance has an additional solution. He also shows that this expands to n given solutions.

[1]This paper is, except for the abstract, in Japanese. An English version of the proof can be found in [Yat03].

Slither Link instances which make good puzzles for a human player should have, among other things, a unique solution. This together with the aforementioned result by Yato is the reason why Saito writes about the automatic generation of Slither Link puzzles [Sai98].

3. AIMS

The aim of this project was to develop a rule-based method to solve Slither Link instances. Such an approach had to consist of two main components. The first component consists of finding rules and the second one of applying them in an intelligent fashion to a problem instance. The original plan was to create the ruleset by evolutionary algorithms, as the ruleset creation was assumed to be the difficult part. It was also assumed that typical instances could be solved completely by rules, so that the time needed to solve an instance could have served as a fitness function. It became apparent early that a deterministic approach to the rule set creation, meant as a starting point and for comparison at first, was already performing well. Additionally these rule sets could not, despite being the product of an exhaustive search, solve most typical problem instances. Because of this evolutionary algorithms were dropped. During the course of the project the aim was expanded to also include the refinement of rulesets. As rules alone were not able to solve most problem instances completely constraint logic programming and SAT-solving were examined.

4. DEFINITIONS AND NOTATION

Before we come to rules we have to model the puzzle of Slither Link: A problem instance $I = (c, N, M)$ of Slither Link consists of

- the width N and height M of the grid,
- and a mapping of positions to cell values $c : \{1, \ldots, N\} \times \{1, \cdots, M\} \to \{empty, 0, 1, 2, 3\}$ with $(i, j) \mapsto c_{i,j}$.

The edges which can be used in a solution form a grid graph of dimensions $(N + 1) \times (M + 1)$. A possible solution S can then be described by labelling the edges of this grid graph with 0 or 1 depending on if they are part of the cycle:

- $S = (h, v)$
- horizontal edges $h : \{1, \ldots, N\} \times \{1, \ldots, M + 1\} \to \{0, 1\}$ with $(i, j) \mapsto h_{i,j}$
- vertical edges $v : \{1, \ldots, N + 1\} \times \{1, \ldots, M\} \to \{0, 1\}$ with $(i, j) \mapsto v_{i,j}$

- $h_{i,j} = 1$ iff the horizontal edge at position (i, j) is part of the cycle
- $v_{i,j} = 1$ iff the vertical edge at position (i, j) is part of the cycle

The cell (i, j) has the horizontal edge $h_{i,j}$ above and the vertical edge $v_{i,j}$ left of itself. How cells and edges are arranged can be seen in Figure 2.

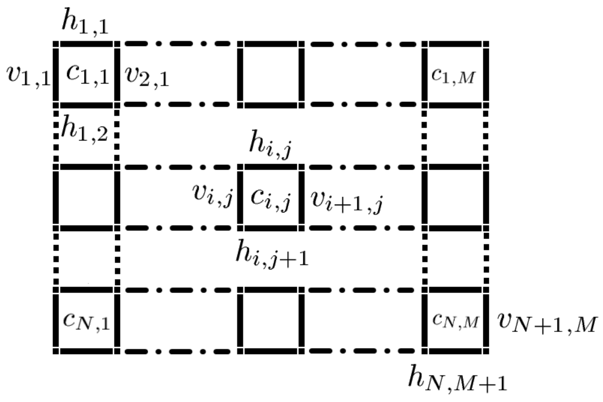

Fig. 2. Structure of a problem instance

In order to distinguish between edges in the grid graph and edges of the cycle we call the former simply *edges* and the latter *positive edges* when used in the cycle and *negative edges* when not. Later we will also deal with intermediate solutions with *unassigned edges*. We will also make no strict distinction between $h_{i,j}$ and $v_{i,j}$ having a value of zero or one and being a boolean variable in a propositional formula.

5. LOCAL CONSTRAINTS

This section will present two types of constraints. These constraints can be called local since they deal with a small number of connected edges. They are also structured in a way, so that the same configuration found at two different positions in a board results in constraints, that only differ in the indices of variables. The locality and recurring structure of constraints are the actual leverage points for rules. They allow us to find local rules which can be applied at different positions in a board.

5.1 Constraints by cell value

The first type of constraint is directly imposed by the problem itself. If a cell $c_{i,j}$ is not $empty$ the number of surrounding positive edges has to be exactly $c_{i,j}$. For $(i, j) \in \{1, \dots, N\} \times \{1, \dots, M\}$ and $c_{i,j} \in \{0, 1, 2, 3\}$ we get:

$$c_{num}(i, j) \ : \ h_{i,j} + v_{i,j} + h_{i,j+1} + v_{i+1,j} = c_{i,j} \qquad (1)$$

This type of constraint varies with the actual problem instance.

5.2 Constraints by edge degree

The second type of constraint exploits that every solution forms a cycle without crossings in the grid graph. If a node of the grid graph lies on the cycle two of its incident edges have to be positive. A cycle without crossings is equivalent to a cycle which visits every node at most once, so the number of incident positive edges can never be greater than two. On the other hand if a node does not lie on the cycle also none of its incident edges can be positive. This leads to the second constraint: *For every node either zero or two incident edges are positive in every solution.* We get one constraint per node, that is for $(i, j) \in \{1, \dots, N + 1\} \times \{1, \dots, M + 1\}$:

$$c_{ed}(i, j) \ : \ h_{i-1,j} + v_{i,j-1} + h_{i,j} + v_{i,j} \in \{0, 2\} \qquad (2)$$

Nodes at the borders of the board have less than four incident edges. For these edges variables which are not in the domains of h or v, like $h_{0,j}$, occur and have to be considered zero.
In contrast to the first type of constraint, the edge degree constraint does not change with the problem instance. The only change is the number of constraints for differently sized problems.

5.3 Constraints as propositional formulas

Both types of constraints can easily be translated into propositional formulas. A list of these formulas can be found in Appendix I. The propositional formula which is equivalent to $c_{ed}(i, j)$ is called $\varphi_{ed}(i, j)$. The propositional formulas equivalent to $c_{num}(i, j)$ are divided into four subtypes according to the actual value of $c_{i,j}$. These are called $\varphi_{num,0}(i, j)$ to $\varphi_{num,3}(i, j)$. With these we can define two sets of propositional formulas

- $\Phi_{num} = \bigcup_{i,j} \varphi_{num,c_{i,j}}(i, j)$ for $(i, j) \in \{1, \dots, N\} \times \{1, \dots, M\}$ and $c_{i,j} \neq empty$
- $\Phi_{ed} = \bigcup_{i,j} \varphi_{ed}(i, j)$ for $(i, j) \in \{1, \dots, N + 1\} \times \{1, \dots, M + 1\}$
- $\Phi = \Phi_{num} \cup \Phi_{ed}$

Every model of Φ is a candidate solution of the Slither Link instance. An variable assignment which fulfils both types of constraints for the all the given values of i and j has the correct number of edges around cells and additionally every edge lies on a cycle. The only condition missing to a solution is that we need a *single* cycle.

6. RULES

There are different ways to represent rules, but they all have a common structure. Rules consist of a prerequisite part, also called left hand side, and a conclusion part, also called right hand side. The prerequisite part contains cell values and positive and negative edges together with their position relative to each other. The conclusion part then tells us which positive and negative edges can be assigned when the prerequisite occurs somewhere in the board. We will not consider rules with conclusions like ”either edge A or edge B has to be positive”. These rules would not fit well in our concept, since we want the conclusions of one applied rule to create the prerequisites for the next applied rule.

6.1 Representation of rules

The most intuitive way to represent rules is graphically in the way that can be seen in Figure 3. When we find the left hand side of such a graphical rule somewhere in the board we can exchange it with the right hand side.

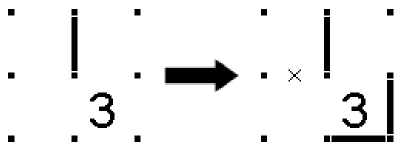

Fig. 3. Graphical rule

Such a graphical representation fails when it comes to handling rules in algorithms or as mathematical objects. We encode every cell or edge on both sides of the rule as tuples consisting of position and value. The negative edge in our example rule would be encoded as $(h, 0, 1, 0)$ where $h, 0, 1$ are orientation and position and 0 is the value. We then have three sets of tuples: cell values, left hand side edges and right hand side edges. The complete example rule is $(\{(1, 1, 3)\}, \{(vert, 1, 0, 1), (horiz, 0, 1, 0)\}, \{(vert, 2, 1, 1), (horiz, 1, 2, 1)\})$. In general a rule R can be described as:

- $R = (C_{lhs}, E_{lhs}, E_{rhs})$
- C_{lhs} finite subset of $C = \{(i, j, v) \colon i, j \in \mathbb{N}, v \in \{0, 1, 2, 3\}\}$
- E_{lhs} and E_{rhs} finite subsets of $E = \{(o, i, j, e) \colon o \in \{horiz, vert\};\ i, j \in \mathbb{N};\ e \in \{0, 1\}\}$

We say a rule *matches* a board $B = ((c, N, M), (h, v))$ at position $(\delta i, \delta j)$,

- if for all $(i, j, v) \in C_{lhs} \Rightarrow c(i + \delta i, j + \delta j) = v$,
- for all $(horiz, i, j, e) \in E_{lhs} \Rightarrow h(i + \delta i, j + \delta j) = e$,
- and for all $(vert, i, j, e) \in E_{lhs} \Rightarrow v(i + \delta i, j + \delta j) = e$

A rule is called *correct* if the conclusion, under the effective constraints, follows from the prerequisites. A rule is called *maximal* if it is correct and its conclusion is the union of the conclusions of all correct rules with the same prerequisites.

6.2 Detecting inconsistencies through rules

Rules can also be used to detect inconsistencies. This detection occurs when a rule matches a board, but the conclusion contradicts what is already assigned to an edge. When looking at Figure 4 we have a rule in the upper part that matches the board in the lower part. But when we apply the rule, we notice that one edge would change from negative to positive. This means we have found an inconsistency. If the rule is correct, the board must be unsatisfiable with the current edge assignment. Exactly this is the case in our example: We have a cell with a value of 3, which implies exactly one negative edge, but our board has two negative edges. The rule has served as a test for satisfaction of a constraint.

7. Automatic construction of rules

7.1 Rule construction algorithm

This section will present an algorithm to construct the conclusions of a rule from its prerequisites. The prerequisites

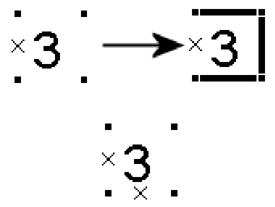

Fig. 4. Rule detecting inconsistency

of a rule are basically a part of a board in which some cell values and edges are already set. Such a subboard is subject to the same local constraints as a complete board. What we do in the algorithm is to generate all possible assignments to the edges that have not been fixed in the prerequisite and test which of these assignments fulfil the constraints. By doing this, we get the set of all consistent assignments on this subboard. In the next step, we determine those edges which have been assigned the same value in the whole set. These edges form the conclusion, as we know that they are the same in every solution.

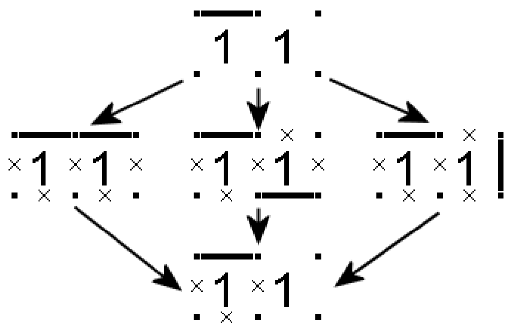

Fig. 5. Construction of a rule

Figure 5 is an example for the process: From the prerequisites three consistent assignments are calculated. Then these assignments are intersected to form the conclusion.

In order to be able to enumerate all assignments on a subboard, we have to define which edges are part of it. We will do this by restricting our rule to being rectangular and then specifying its width and height. Choosing a bigger rectangle has the advantage to include more edges that potentially could end up in the conclusion.However, the disadvantage is that with every added edge the number of assignments we have to test doubles. The restriction to a rectangle is not strictly necessary, it just makes presentation and implementation easier and ensures that we have a connected set of edges. In general we would only need to define the set of edges in the subboard.
The following pseudocode shows the algorithm. The core of the Java implementation can be found in Appendix I

```
constructRule($C_{lhs}$,$E_{lhs}$, $width$, $height$)
  $B$ is board of dimensions $width \times height$
```

```
$CA = \emptyset$
for all edge assignments of $B$ $\{e = (o, i, j, v)\colon e \in E\}$ {
    if ($ea$ is consistent){
        $CA = CA \cup ea$
    }
}
$E_{rhs} = (\bigcup_{e \in CA} e) \setminus E_{lhs}$
Output $E_{rhs}$ and $CA$
```

There are three possible outcomes of the algorithm:

- $CA = E_{rhs} = \emptyset$: There are no consistent assignments. This happens when the prerequisites imply that the subboard has no solution.
- $CA \neq \emptyset$ and $E_{rhs} = \emptyset$: We cannot conclude something from the prerequisites, but the subboard is solvable.
- $CA \neq \emptyset$ and $E_{rhs} \neq \emptyset$: We have constructed a new rule $R = (C_{lhs}, E_{lhs}, E_{rhs})$ and can add it to our ruleset.

The algorithm has a time complexity of $O(2^e)$ where e is the number of edges in the subboard that have not been fixed by the prerequisites. Since it does an exhaustive search the constructed rules are always maximal.

7.2 The standard ruleset

We are now able to construct a ruleset from a set of prerequisites. Now we need to define this set of prerequisites. Section 9 will present effective methods to remove redundant rules from a ruleset, so we do not have to be concerned that we get too many rules with a large number of prerequisites. Three sets of prerequisites have chosen and proved to be helpful:

- All 1×1-rules: This ruleset implicitly encodes φ_{num}. This set contains 115 rules that have been created from $3^4 * 5 = 405$ prerequisites.
- 2×2 rules where, apart from cell values, only the edges that are incident with the centre node are allowed to be used in the prerequisites: These edges have been chosen since they are exactly the ones needed to implicitly encode φ_{ed} ($15,632$ rules created from $3^4 * 5^4 = 50,625$ prerequisites).
- 2×2 rules in which all cells are *empty*: This ruleset also encodes φ_{ed} ($47,601$ rules created from $3^12 = 531,441$ prerequisites).

The union of these rulesets forms what we will call the *standard ruleset*.

8. Application of rulesets

8.1 Simple iterative application of rules

The application of a ruleset to a board is straightforward: We go through all positions on the board and test for every rule in the ruleset if the rule matches at that position. If it does we update the board according to the conclusions of this rule. While updating, we keep track if we actually changed the board. We also determine if the update changes a positive edge to a negative one or vice versa. If this happens we have found an inconsistency. We do this iteratively as long as updates change the board and we do not find an inconsistency. The following pseudocode describes the algorithm.

```
applyRules(Board, ruleSet)
 Changed = true
  while(Changed){
   Changed=false
   for every position $(i, j)$ of the Board{
    for every $R \in ruleSet${
     if $R$ matches Board at position $(i, j)${
       update Board at position $(i, j)$ according to $R$
       if(update discovered inconsistency){
         Output ”discovered inconsistency”
         Abort and return from method
       }
       if(update changed board){
         Changed = true
       }
     }
    }
   }
  }
 Output Board
```

The algorithm can exit in two different ways:

- It discovers an inconsistency. This means, assuming all rules are correct, that the board is not satisfiable.
- It terminates normally and outputs the board which has all the edges that could be concluded from the ruleset.

8.2 Trial and generalisation

After the ruleset has been applied to a board and the instance is not solved yet, we go over to test different assignments to edges of the board. We do this in the following fashion:

- Find an unassigned edge
- Set this edge to 1 and apply the ruleset. If we find an inconsistency we can safely set the edge to 0 and are done.
- We do the same again, but this time we set the edge to 0 and if we find an inconsistency later to 1.
- If both possibilities are consistent we compare the two board we got after applying the ruleset. If we find an edge that is assigned the same value in both and that is unassigned in the original board, we have also gained knowledge.
- If we have found new assignments, we check whether we can successfully apply the ruleset again.

We exploit here that it is not necessary to explicitly check the local constraints. They have been completely incorporated into the rule set. The Java implementation can be found in Appendix I.

9. Refining the set of rules

The automatic construction of rules by the methods of section 7 leads to large numbers of them. Since the time needed to apply a ruleset is linear to the number of rules, it is desirable to reduce this number. Our goal is to retain the full expressiveness of the ruleset. Both methods remove redundant rules, rather than replacing a number of rules with fewer new rules. The idea is to remove rules whose function can be simulated by one or more other rules. For this purpose we will define two relations: *dominates* and *composed*.

9.1 Dominated rules

A rule R *dominates* R', if R has weaker or the same prerequisites and stronger or the same conclusions as R'.
In our definition of rules this means $R = (C_{lhs}, E_{lhs}, E_{rhs})$ *dominates* $R' = (C'_{lhs}, E'_{lhs}, E'_{rhs})$ if

1) $C_{lhs} \subseteq C'_{lhs}$
2) $V_{lhs} \subseteq V'_{lhs}$,
3) and $V'_{rhs} \subseteq V_{rhs}$

The removal of rules is straightforward:

```
removeDominated($RuleSet$)
  for ($R \in RuleSet$)
      for ($R' \in RuleSet$)
          if($R \neq R'$ and $R$ dominates $R'$)
              $RuleSet = RuleSet \setminus R'$
  Output $RuleSet$
```

This reduction is already quite effective. The standard ruleset can be reduced from $63,348$ to $1,862$ rules. The time complexity of $removeDominated$, when considering the size of rules as constant, is $O(|RuleSet|^2)$. Fortunately removed rules can be ignored in further steps of the nested for-loop. This way the actually needed tests inside the loop can be reduced from about $4*10^9$ to $6.5*10^7$ for the standard ruleset. The order in which rules are tested for removal does not affect the end result since $dominates$ is transitive.

9.2 Composed rules

The application of one rule can be simulated by the sequence of applying other rules. This can for example be seen in Figure 6. The two left rules can be used to simulate the effect of the right rule.

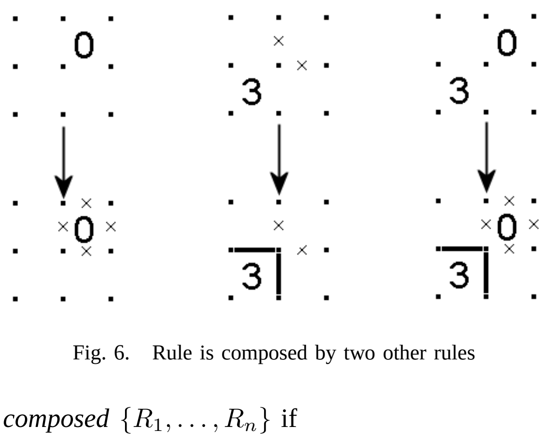

Fig. 6. Rule is composed by two other rules

R' *composed* $\{R_1, \ldots, R_n\}$ if

- Let the board B_1 contain the prerequisites of R',
- R_i matches B_i and its application results in B_{i+1}
- and every edge in the conclusion of R' is assigned the same value in B_{n+1}.

This means that ”composition” is always stronger than ”domination”,as R *dominates* R' implies R' *composed* $\{R\}$. It still makes sense to remove dominated rules first, as the test for composition needs far more time as we will see.
We remove composed rules by going through the ruleset and testing each rule. We do this by constructing a board containing the prerequisites of a rule and then applying the rest of the ruleset on this board. If this results in a board which contains all conclusions of the rule, the rest of the could simulate the effects of the rule and the rule can be removed.

```
removeComposed($RuleSet$)
  for ($R = (C_{lhs}, E_{lhs}, E_{rhs}) \in RuleSet$)
      $Board$ is constructed from $C_{lhs}$ and $E_{lhs}$
      applyRules($Board$, $RuleSet \setminus R$)
      if($e \in E_{rhs}$ implies $e \in Board$)
          $RuleSet = RuleSet \setminus R$
  Output $RuleSet$
```

In contrast to the $dominates$ relation, the order in which rules are tested and removed matters. For the standard ruleset, varying this order resulted in sizes between 193 and 224 rules. This is never the less a massive reduction from $1,862$ rules with the *dominated*-relation alone. By combining both algorithms we have achieved a reduction in the number of rules by a factor of 300, from over $60,000$ to 200.

10. Constraint Logic Programming and SAT-Solving

The already presented methods cannot solve Slither Link problems in the general case, as they are polynomial and Slither Link NP-complete[2]. For this task Constraint Logic Programming is ideal as we already have constraints. The only requirement for a solution that these constraints do not capture is that we need a *single* cycle. We can test a labeling fulfilling the constraint with logic programming in the following way:

1) Count all edges
2) Take one edge and determine the length of the cycle it lies on, by following it
3) If both numbers match, we have found a solution else find next labeling and test it

We can now combine this with the rule-based methods by using their results as starting points. This has been tested with a 35×40 instance ([Fuj] problem 34) containing 2775 edges. Using CLP without any precomputed edges does not find any labeling within 3 hours of cpu-time, after which the run was aborted. When 1754 edges which had been precomputed in 156 seconds are used as a starting point the time needed to find a first labeling drops to less than a second.

[2]Assuming $P \neq NP$

Different problem instances were also encoded for the Satzoo SAT-solver using the propositional formulas that can be found in Appendix I. The set of all candidate solutions for 10×10 instances was typically found in about five minutes.

11. Further Work

This project leaves many points further work could pick up:

- The most interesting further work would be to expand the ideas to other problems than Slither Link. Such problems should be expressible as propositional formulas and these formulas should be local and parameterised over for example (i, j).
- The construction of rules as it is presented in this paper uses Generate and Test. The scalability could be improved by moving to Test and Generate.
- Even though rotational and reflective symmetries do not provide an opportunity to refine rulesets even more (they would only reduce the size of the representation and not the number of rules that have to be tested when applying a ruleset), they improve performance when constructing it. This could be done by defining an order on prerequisites and only constructing rules from the smallest in a equivalence class. The resulting rule can then be rotated and reflected to obtain all representations.
- Further work could be done on the optimal order in which rules are tested on the $composed$-relation. It could also focus on completely different ways to remove rules based on this relation.
- The performance of the refinement of rulesets could be improved by dividing rulesets and applying the refinement on these first and then on the remaining rules. A first attempt could cut the time needed to construct and refine the standard rule set by 40%.

12. Conclusions

This paper has presented a process for the solution of Slither Link that chains together different techniques. It starts with the construction and refinement of a ruleset. This step is successful in providing a small but expressive ruleset for every Slither Link instance. The process continues with the application of the ruleset and finishes with the use of constraint solving. All in all this process is successful in solving Slither Link instances efficiently.
The presented ideas are also not specific to Slither Link and can be transferred to other similar problems.

Acknowledgements

Thanks to my project supervisor Stefan Kahrs for the advice and comments during our weekly meetings. Also thanks to Andy King for his comments on my presentation and for pointing out that the last step of the algorithm described in Section 9.2 is a well known technique called generalisation.

References

[Fuj] Hirofumi Fujiwara. Collection of Slither Link instances. http://www.pro.or.jp/ fuji/java/puzzle/numline/index-eng.html.

[Sai98] T. Saito. An algorithm for automatic generation of Slither Link problems, 1998. Senior Thesis.

[Sug] Tsuyoshi Sugiyama. SlitherLink for Palm PDAs. http://hobbit.spire.timedia.co.jp.

[Yat00] Takayuki Yato. On the NP-completeness of the Slither Link Puzzle. *IPSJ SiG Notes*, AL-74:25–32, 2000. In Japanese.

[Yat03] Takayuki Yato. Complexity and Completeness of Finding Another Solution and its Application to Puzzles. Master's thesis, Univ. of Tokyo, Dept. of Information Science, Faculty of Science, Hongo 7-3-1, Bunkyo-ku, Tokyo 113, JAPAN, Jan 2003.

Appendix I: Central algorithms in Java

```java
private static Board constructRuleFromPrerequisites(Board subboard){
    GeneratorBoard generBoard= new GeneratorBoard(subboard);
    generBoard.prepareBoard();
    GeneratorBoard resultBoard = null;
    boolean firstConsistent=true;
    int consistents=0;
    do{

        if(generBoard.consistencyTest()){
            consistents++;
            if(firstConsistent){
                //copy first consistent board to result
                firstConsistent=false;
                resultBoard= new GeneratorBoard(generBoard);
            }
            else{
                //compare consistent board to current result
                resultBoard.bpos.reset();
                generBoard.bpos.reset();
                do{
                    if(resultBoard.getEdgeAtPos() != generBoard.getEdgeAtPos()){
                        resultBoard.setEdgeAtPos(Board.EMPTYEDGE);
                    }
                }while(resultBoard.bpos.nextPosition() && generBoard.bpos.nextPosition());
            }
        }
    } while(generBoard.nextEdgeConfiguration());
    return resultBoard;
}
```

Fig. 7. Rule Construction

```java
public static void trySingleEdges(Board board, RuleSet ruleSet) {
    BoardPosition edgePos= new BoardPosition(board.getWidth(),board.getHeight());
    Board resultingBoardEDGE;
    Board resultingBoardNOEDGE;
    BoardPosition bpos= new BoardPosition(board.getWidth(),board.getHeight());
    int edge;
    boolean changed=true;
    boolean justChanged;
    while(changed){
        changed=false;
        edgePos.reset();
        while(board.nextEmptyEdge(edgePos)){
            resultingBoardEDGE = new Board(board);
            resultingBoardNOEDGE = new Board(board);
            justChanged=false;
            if(!tryEdge(board,ruleSet,edgePos,Board.EDGE,resultingBoardEDGE)){
                //falsified proposition
                board.setEdgeAtPos(edgePos, Board.NOEDGE);
                justChanged=true;
            } else if(!tryEdge(board,ruleSet,edgePos,Board.NOEDGE,resultingBoardNOEDGE)){
                board.setEdgeAtPos(edgePos, Board.EDGE);
                justChanged=true;
            } else { //see if something can be learned by comparing
                bpos.reset();
                do{
                    if((board.getEdgeAtPos(bpos)== Board.EMPTYEDGE)
                            && (resultingBoardEDGE.getEdgeAtPos(bpos)!= Board.EMPTYEDGE)
                            && (resultingBoardEDGE.getEdgeAtPos(bpos)
                                    ==resultingBoardNOEDGE.getEdgeAtPos(bpos))){
                        board.setEdgeAtPos(bpos,resultingBoardEDGE.getEdgeAtPos(bpos));
                        justChanged=true;
                    }
                }while(bpos.nextPosition());
            }
            if(justChanged){
                applyRuleset(board, ruleSet);
                changed=true;
            }
            edgePos.nextPosition(); //skip current position
        }
    }
}
```

Fig. 8.   Trying single lines