Bachelor thesis
Computing Science

Radboud University

Analysis of Akari

Author:
Bram Pulles (1015194)
`borroot@live.nl`

First supervisor/assessor:
Prof. dr., H. Zantema
`h.zantema@tue.nl`

Second assessor:
Prof. dr., H. Geuvers
`herman@cs.ru.nl`

January 9, 2021

Abstract

Logic puzzles have been gaining popularity over the years and many puzzles have been proven to be NP-complete. Akari (also known as Light Up) is proven to be NP-complete by reduction from Circuit-SAT.[15] In this paper we inspect six variants of Akari, three variants are proven to be NP-complete by reduction from Circuit-SAT and two variants are proven to be in P. We furthermore implement algorithms for solving Akari using a SAT solver and backtrack solver, and an algorithm for generating new Akari puzzle instances with exactly one unique solution.

Contents

Chapter 1

Introduction

Logic puzzles are puzzles which can be solved using deductive reasoning, like the popular puzzle Sudoku. The increasing popularity of logic puzzles also brings along scientific research on the complexity of these puzzles, see section 5. The puzzles are interesting to investigate, because they are often described using a small set of simple rules, while being difficult to solve. Solving logic puzzles and determining their complexity class can also bring new insights to real world problem solving applications.

One of the puzzles investigated is Akari (also known as Light Up), this puzzle is proven to be NP-complete by reduction from Circuit-SAT.[15] A problem is NP-complete iff it is both in NP and NP-hard. The NP-completeness of Akari provides an indication of how fast an arbitrary Akari instance can be solved using an efficient algorithm. The time to solve an Akari instance will grow exponentially (assuming P $\neq$ NP and no approximation algorithm is used).

In this paper we investigate the time complexity of six variants of Akari. The goal of this research is to find out how far we can loosen up the rules of Akari while maintaining NP-completeness. This will give more insight into what makes Akari such a difficult puzzle to solve. So for every variant we ask the question: "Is this variant NP-complete?".

In addition to the theoretical research on the time complexity of Akari variants, we also go through three different algorithms which can be used to solve Akari puzzles. We implement a trivial solver which provides a partial solution extremely fast, a SAT solver which reduces the puzzle to a satisfiability problem and a backtrack algorithm which uses backtracking to solve Akari puzzles.[1] The goal is to find out how fast we can solve Akari puzzle instances, while knowing Akari to be NP-complete.

We start of in section 2 by describing the rules of Akari. Section 3 discusses the time complexity of the six Akari variants and a mechanical way of verifying the NP-completeness proofs in section 3.1.7. Section 4 describes

[1] All implementations can be found at `https://github.com/borroot/akari`

the three different solving algorithms, a short performance comparison and an algorithm to generate new Akari puzzle instances (which was particularly useful for testing the solving algorithms). Section 5 provides a concise overview showing related work. At last, section 6 discusses the conclusions of this paper and ideas for future work.

Chapter 2

Preliminaries

2.1 Akari rules

Akari is usually played on a rectangular grid. The grid has both black cells and white cells in it. The objective is to place light bulbs on the grid such that every white/empty cell is lit. A cell is illuminated by a light bulb if they are in the same row or column, and if there are no black cells between them. Also, no light bulb may illuminate another light bulb. Some of the black cells have numbers in them. A number in a black cell indicates how many light bulbs must share an edge with that cell.[5]

See figure 2.1 for an example puzzle and its solution.

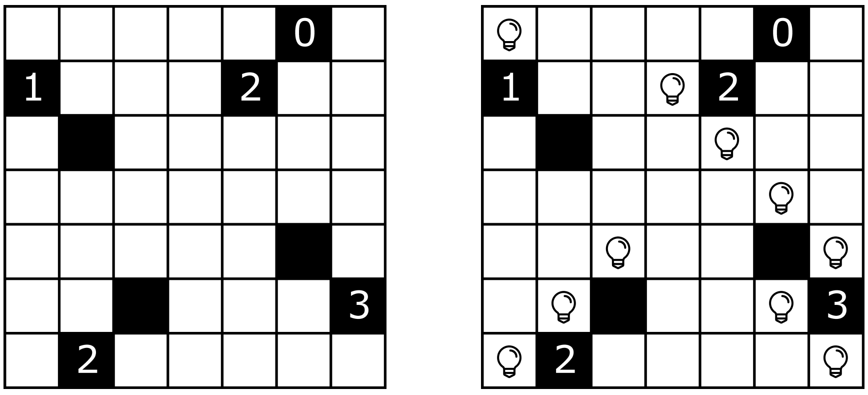


Figure 2.1: Example puzzle.

Chapter 3

Theoretical Research

3.1 NP-completeness of variants

In this section we provide NP-completeness proofs and refutations for variants on Akari. All variants are based on restricting the number constraints in the walls. We analyse the complexity of Akari when no number constraints are allowed or only a certain number constraint is allowed (0, 1, 2, 3 or 4) in addition to empty walls.

In order to show NP-completeness we need to show that the variant is NP and NP-hard. Showing that all variants are NP is easy, when a solution, i.e. a placement of light bulbs, is given, it is simple to check whether all cells are lit up and no constraints are violated.

For the variants which are NP-hard we provide a polynomial-time reduction from Circuit-SAT, where Circuit-SAT is satisfiable iff the corresponding Akari puzzle has at least one solution. Showing that a variant is in P is done by providing a polynomial-time algorithm which solves the variant.

The reduction from Circuit-SAT consists of a few gadgets which need to be constructed. These gadgets are enumerated below.[19] For satisfying 5 it suffices to have a functional complete set of logical connectives translated into gadgets, we use $\{\mathrm{NOT}, \mathrm{OR}\}$.[20] Constructing 6 is automatically done when 1-5 are made, since the crossover gadget can be created using three XOR gates, see figure 3.1 and table 3.1.

1. Wire gadget: simulates the wires in the circuit.
2. Turn gadget: redirect wires in any direction.
3. True gadget: force the output of the circuit to be true.
4. Split gadget: all output wires have the same value as the input wire.
5. Gate gadget(s): simulating the gates of the circuit.
6. Crossover gadget: have two wires cross each other without interacting.

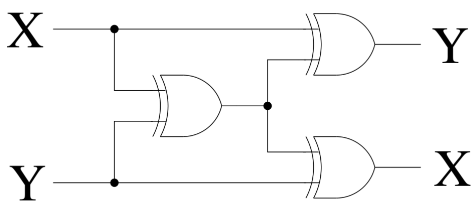


Figure 3.1: Crossover gadget constructed using XOR gates.

X	Y	$X \oplus Y$	$X \oplus (X \oplus Y) = Y$	$Y \oplus (X \oplus Y) = X$
0	0	0	0	0
0	1	1	1	0
1	0	1	0	1
1	1	0	1	1

Table 3.1: Crossover gadget constructed using XOR gates.

3.1.1 No number constraints

When there are no number constraints, it is easy to show that Akari is in P. A polynomial algorithm to solve any given puzzle without any number constraints is given in block 1.

By construction there will be no dark (non lit up) cells left. Additionally, since a dark cell is by definition not in line with any light already placed, we will never have two lights illuminating each other when placing lights on dark cells. Thus providing us with a valid solution.

Interestingly, any puzzle without number constraints can be solved this way. A puzzle with number constraints which are all satisfied by a partial solution, can be finished by the algorithm shown in block 1, see section 3.1.2.

Algorithm 1 Solving puzzles without numbers.

```
Require: puzzle without numbers
Ensure: solution
1: while dark cells left do
2:     place light on (random) dark cell
3: end while
4: return placed lights
```

3.1.2 Number constraint: 4

Puzzles with exclusively the number constraint 4 are in P. A polynomial algorithm to solve these puzzles is given in block 2.

We start of by satisfying all the number constraints. We can use the technique described in section 4.1.1, namely whenever there is an n-cell with exactly n neighbours which can have a light, exactly those neighbours will have a light. Every cell can have at most four neighbours, for a 4-cell this means exactly those neighbours need to have a light. If a 4-cell has less than four neighbours, then the puzzle has no solution. In other words, assuming the puzzle has a solution, we can satisfy all the number constraints just by placing lights on all four neighbours for every 4-cell. These steps are described with the lines 1-3 in block 2.

After all of the number constraints are satisfied we can use the algorithm from block 1 as described in section 3.1.1 to completely solve the given puzzle. These steps are described with the lines 4-6 in block 2.

In block 3 an additional check is shown which can be added between line 1 and 2 in block 2 in order to check if the puzzle is solvable. If the puzzle is not solvable, the algorithm will return null.

Algorithm 2 Solving puzzles with only 4's.

```
Require: puzzle with only 4's
Ensure: solution
1: for every 4-cell do
2:     place light on all neighbours
3: end for
4: while dark cells left do
5:     place light on random dark cell
6: end while
7: return placed lights
```

Algorithm 3 Checking solvability of puzzles with only 4's.

```
1: if 4-cell has less than 4 placable neighbours then
2:     return null
3: end if
```

3.1.3 Number constraint: 1

Puzzles containing the number constraint 1 are NP-hard. We give a construction for gadget 1-5 necessary for the reduction proof. All the constructed gadgets are mechanically verified, see section 3.1.7.

In the upcoming gadgets we will often see two variables displayed as x and x', these two variables encode one logical variable. The variables in the gadgets always have the relation $x \leftrightarrow \neg x'$, we say the logical variable is true iff x contains a light (and x' does not). The arrows indicate a certain truth value coming in or going out. The truth values come in as x (represented by the arrow) and can be catched with x' (usually right after the incoming arrow). Outgoing truth values are always send out as x (usually right before the outgoing arrow). This convention allows us to easily connect gadgets.

An example usage of the wire/turn gadget can be seen in figure 3.2. The wire has two possible states: true or false. If the x cells contain light bulbs, we say the state is true. If the x' cells contain light bulbs, we say the state is false. For the human interpreter these states are marked green and red respectively. If x is true, light is coming in and going out. If x' is false, light is not coming in neither is it going out.

The light bulbs seen in the wire/turn gadget are meant to illustrate that the wire can be stretched however seen fit, thus making sure that all gadgets can be wired up to each other.

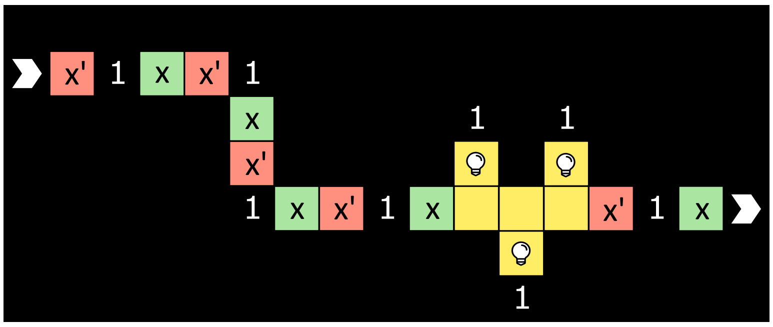
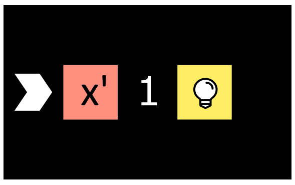


Figure 3.2: Wire/turn gadget with just 1's.

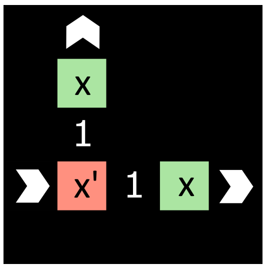
The true gadget is constructed as shown in figure 3.3a. The 1-cell with a light bulb next to it effectively changes to a 0-cell making it impossible for x' to contain a light bulb thus forcing x (indicated with the incoming arrow) to contain a light bulb. This way we force the circuit to be true.

The split gadget is constructed as shown in figure 3.3b. This one is extremely simple, we have an input, and by using the 1-cells we can immediately copy this input into two outputs.

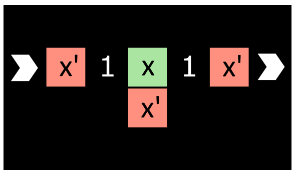
The gate gadgets are shown in figure 3.3c and 3.4. The not gadget reverses the input. If light is coming in, no light will go out, and the other way around. The or gadget is a bit more complicated.



(a) True gadget only 1's.



(b) Split gadget only 1's.



(c) Not gadget only 1's.

Figure 3.3: True, split and not gadget with just 1's.

We have two variables x and y as input and z as output. The most important part is on the complete left where you see a, b and a white cell in between. This translates to, at most one from a and b can have a light. Because if a and b would have a light bulb, they would illuminate each other. The white cell in between ensures that also if a and b both do not contain a light bulb, the cells will still be illuminated. Using this construction we can make it such that z is only true iff we have $a \vee b$, or in other words at most one out of a and b is false.

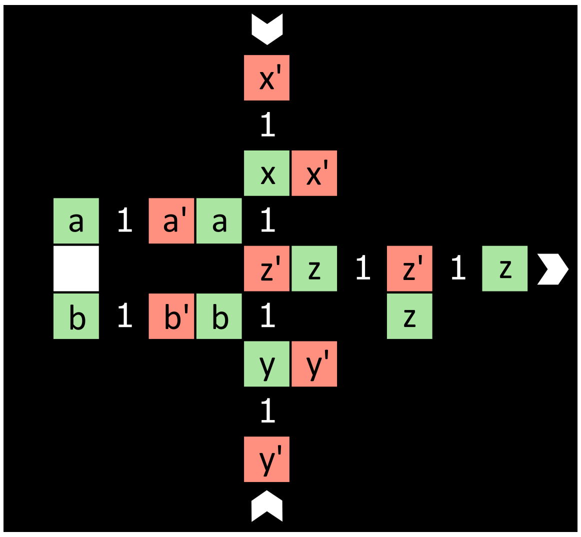


Figure 3.4: Or gadget with just 1's.

The or and not gates together provide a functionally complete set of gates and thus jointly form the gate gadget. Using all of the gadgets described above we can translate any Circuit-SAT problem into an Akari puzzle with merely 1's in polynomial-time. The Akari puzzle which is generated will have at least one solution iff the Circuit-SAT problem is satisfiable. This concludes the proof that Akari with just 1's is NP-hard.

3.1.4 Number constraint: 2

Puzzles containing the number constraint 2 are NP-hard. We give a construction for gadget 1-5 necessary for the reduction proof. All the constructed gadgets are mechanically verified, see section 3.1.7.

An example usage of the wire/turn gadget can be seen in figure 3.5. The state of the wire is defined as described in section 3.1.3. The wire gadget is logically identical to the wire gadget in figure 3.2. Notably, the light bulbs next to the 2-cells effectively convert the 2-cells to 1-cells, this tiny build reoccurs in the other gadgets as well.

The light bulb construction seen on the right is meant to illustrate that the wire can be stretched however seen fit, thus making sure that all gadgets can be wired up to each other.

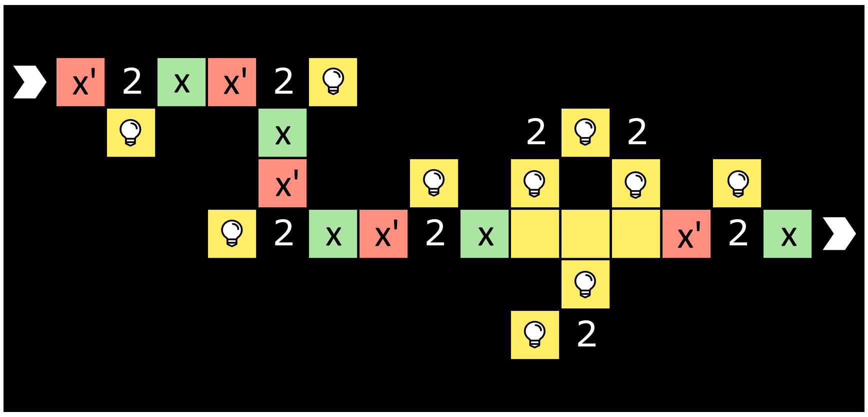
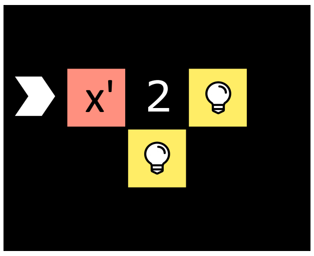


Figure 3.5: Wire/turn gadget with just 2's.

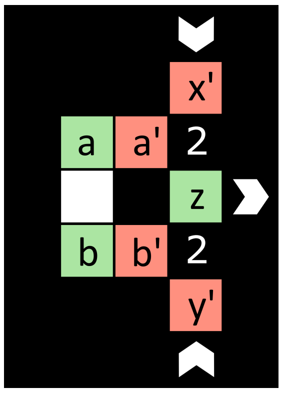
The true, split and not gadget are shown in figure 3.6a, 3.6c and 3.6d respectively. The construction of these three gadgets is analogous to the ones seen in section 3.1.3. We use the tiny build where we convert 2-cells to 1-cells as described earlier for the split and the not gadget.

The or gadget is shown in figure 3.6b. The working of this gadget is similar to the one seen in section 3.1.3. We again use the construction on the left to force that we never have both x and y false, if z is true.

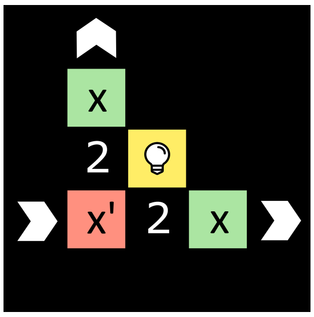
Using all of the gadgets described above we can translate any Circuit-SAT problem into an Akari puzzle with merely 2's, thus completing the proof that Akari with just 2's is NP-hard.



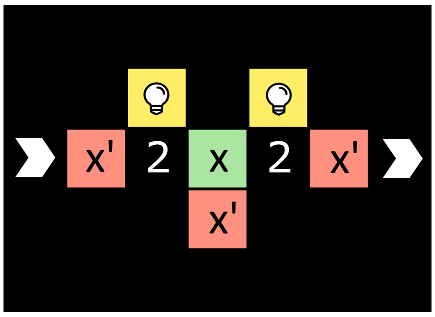
(a) True gadget with just 2's.



(b) Or gadget with just 2's.



(c) Split gadget with just 2's.



(d) Not gadget with just 2's.

Figure 3.6: True, split, or and not gadget with just 2's.

3.1.5 Number constraint: 3

Puzzles containing the number constraint 3 are NP-hard. We give a construction for gadget 1-5 necessary for the reduction proof. All the constructed gadgets are mechanically verified, see section 3.1.7.

An example usage of the wire/turn gadget can be seen in figure 3.7. The state of the wire is defined as described in section 3.1.3. The wire gadget is again logically identical to the wire gadget in figure 3.7. The light bulbs next to the 3-cells effectively convert the 3-cells to 1-cells, this tiny build reoccurs in the other gadgets as well.

The light bulb construction seen on the right is again meant to illustrate that the wire can be stretched however seen fit, thus making sure that all gadgets can be wired up to each other.

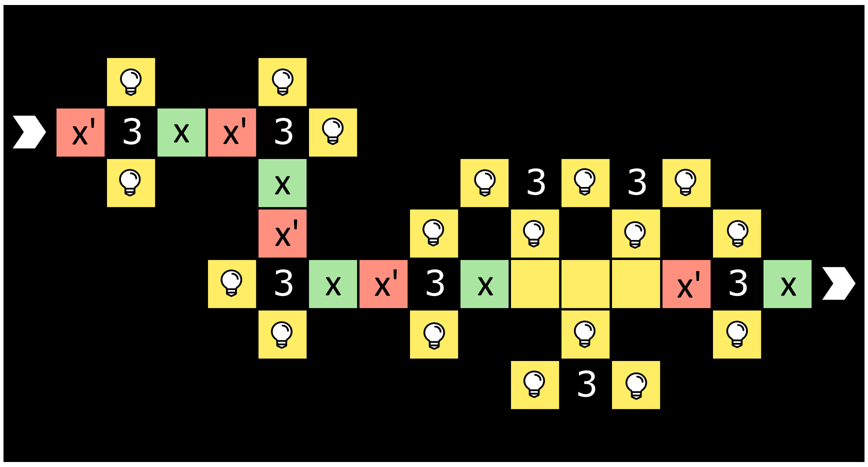
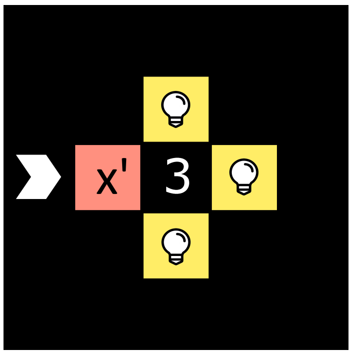


Figure 3.7: Wire/turn gadget with just 3's.

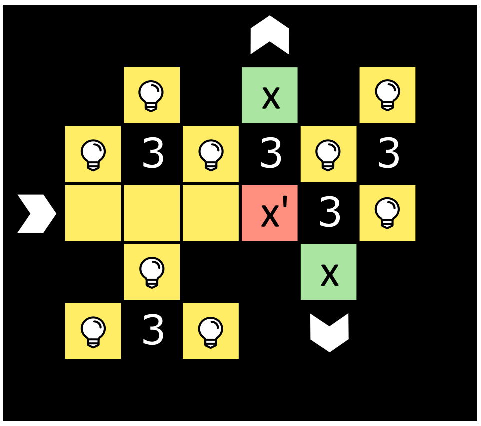
The true, split and not gadget are shown in figure 3.8a, 3.8b and 3.8c respectively. The construction of these three gadgets is analogous to the ones seen in section 3.1.3. We use the tiny build where we convert 3-cells to 1-cells as described earlier for the split and the not gadget. However, as can be clearly seen from these figures, making these gadgets is a lot harder than the ones that we have seen before.

The or gadget is shown in figure 3.8d. The working of this gadget is similar to the one seen in section 3.1.3. We again use the construction on the left to force that we never have both x and y false, if z is true.

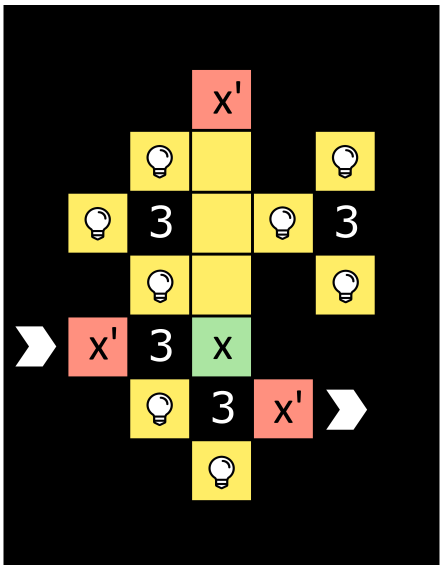
Using all of the gadgets described above we can translate any Circuit-SAT problem into an Akari puzzle with merely 3's, thus completing the proof that Akari with just 3's is NP-hard.



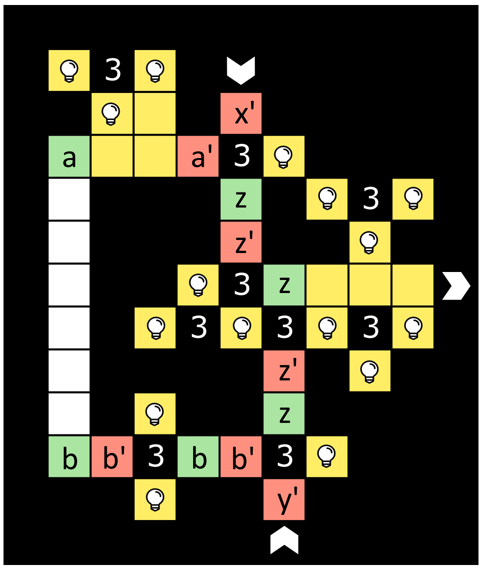
(a) True gadget with just 3's.



(b) Split gadget with just 3's.



(c) Not gadget with just 3's.



(d) Or gadget with just 3's.

Figure 3.8: True, split, or and not gadget with just 3's.

3.1.6 Number constraint: 0

Regarding puzzles containing the number constraint 0 we were unable to prove NP-hardness nor were we able to provide a polynomial-time algorithm. We provide an overview of the attempts made and the problems we ran into.

We have tried several polynomial-time reductions to show NP-hardness. The following three problems are known to be NP-hard, but we were not able to create a reduction to one of these problems

- Circuit-SAT: We were unable to create the essential split gadget.
- Planar monotone rectilinear 3SAT: We were unable to create the variable gadget which has the same truth value across all outputs.
- Planar vertex cover: Adding the 'at most k vertices' condition requires an additional vertex which is connected to all other vertices, but this is not possible in all planar graphs (we do not have a crossover gadget).

We can create a reduction to the following two problems, but they turned out to be in P (using a maximum flow algorithm).

- Given a bipartite planar graph with a number for every vertex. Can you make a subgraph where the degree of every vertex is equal to the corresponding number?
- Given an undirected planar graph with a number for every vertex. Can you give all edges a direction such that the outdegree of every vertex is equal to the corresponding number?

At last we also tried to provide a polynomial-time algorithm to solve the puzzle instances using the following two approaches.

- Maximum flow: Could not create all necessary conditions.
- Limited backtracking: Such as used in solving 2SAT, however the minimal satisfiability problem representing a puzzle instance was still NP-hard to solve, shown by a reduction from CNF-SAT.

3.1.7 Gadget verification

In order to provide robust proofs, all of the gadgets have been mechanically verified using the SAT solver from section 4.1.2.

To fully automate the verification of all gadgets we developed an annotated encoding scheme from which we can read the input and output cells of a gadget, see section A.1.3. But before we can use this scheme we have to adjust all of the gadgets a bit.

The gadget pictures have arrows pointing inwards to indicate input, and arrows pointing outwards to indicate output. We replace all of these arrows with empty cells. These empty cells are necessary so the gadgets are solvable for every valid input/output combination. This is easily seen by the observation that most gadgets are not solvable if they would receive a true input or have a false output, adding the empty cells solves this problem.

In section A.1.3 we described how the variables are encoded in the puzzle and how the input and output lists are structured. Using these lists and a clause specific to the gadget type we are testing we can verify the correctness of every single gadget.

The general structure of verifying a gadget is the same for all gadget types. We can check the correctness of every gadget in just one call to the SAT solver. If the solver says the given formula is unsatisfiable the gadget is constructed correctly. Every formula consists of three parts.

1. The rules of Akari as explained in section 4.1.2.

2. Make sure that the input/output variables, all encoded into two cells, always have two different values in the two cells, meaning one has a light and the other one does not. In order to do this we add $(a \wedge \neg b) \vee (\neg a \wedge b)$ to the formula for every input/output variable x encoded by the cells a and b, this construction is more generic and is also used in the third part to make sure that any $a \neq b$.

3. Add the restriction specific to the gadget type. We add a formula requiring the expected output of the gadget (given the unassigned input variables) to be unequal to the actual output of the gadget. To give an example, the or gadget has input x and y, and output p, we add the formula $x \vee y \neq p$ (with $a \neq b$ as described in the previous paragraph). The split gadget is described as $x \neq p \vee x \neq q$, the not gadget as $\neg x \neq p$, the wire gadget as $x \neq p$ and the true gadget as $\neg x$ (this is an exception since it has no output). Whenever the SAT solver returns that the formula is unsatisfiable we know this part of the formula is unsatisfiable, meaning the expected output of the gadget is always equal to the actual output of the gadget.

Using this fairly simple approach we have fully automatically mechanically verified the correctness of all the gadgets from the NP-hard proofs.

Chapter 4

Practical Research

4.1 Puzzle solvers

In this section we describe how Akari puzzles can be solved. We describe a trivial method which can be used to preprocess the puzzle before using another solver. We also show how a SAT solver and backtrack algorithm can be used to solve Akari puzzles.

4.1.1 Trivial solver

The trivial solver, as the name suggest, executes the trivial steps. These steps require no branching and can be determined with certainty. To give some examples, see figure 4.1. When you see a 3-cell at the wall or a 4-cell (anywhere) you can immediately place light bulbs all around this cell. Similarly for a 2-cell in a corner, we can also place the light bulbs around the other 2-cell because there are only two possible positions left.

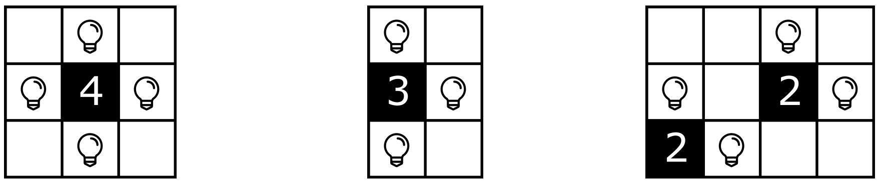


Figure 4.1: Trivial puzzles and their solutions.

To generalize this phenomenon, whenever there is an n-cell which has exactly n possible neighbours which could have a light then exactly those neighbours will have a light. At the same time we can determine for a lot of cells that they cannot have a light, due to the lights which are already placed and from 0-cells. Note that even if the puzzle has multiple solutions, the light bulbs placed by the trivial solver have to be there in every solution, thus providing a base for every solution.

Depending on the case at hand this might not solve the whole puzzle, but it can place the trivial light bulbs after which another solver can be used to place the rest. Using the trivial solver as a preprocessor the search tree to solve a puzzle can be greatly decreased in size. The algorithm for the trivial solver is shown in block 4.

If there is an n-cell with less than n placable neighbours, the puzzle does not have a solution. An additional check can be made in the algorithm to prematurely check if there is no solution possible.

Algorithm 4 Trivial Solver

```
Require: puzzle
Ensure: placement of trivial light bulbs
1: for every 0-cell do
2:     mark every neighbour as implacable
3: end for
4: while $n$-cell with exactly $n$ placable neighbours do
5:     place light on every neighbour
6:     mark newly lit cells as implacable
7: end while
8: return placed lights(, implacable cells)
```

4.1.2 SAT solver

SAT stands for satisfiability referring to the boolean satisfiability problem. We can translate any Akari puzzle to a boolean formula whose solution provides the placement of lights, if the formula is not satisfiable then there is no solution to the puzzle.

Instead of making a SAT solver ourselves we use an existing library. In our case we used the Z3 theorem prover which has been optimised tremendously over the years, well deserving an honourable mention here.

The only thing left is making the translation from an arbitrary Akari puzzle to a boolean formula whose solution provides the placement of lights. The construction of this boolean formula exists of three parts.

1. Every n-cell has exactly n adjacent lights.
2. Every cell is lit up at least once.
3. Every row/column (separated by walls) has at most one light.

Now for the actual construction, let every empty cell be represented by a boolean variable, iff the variable is true there is a light on that cell. Lets say we have a 2-cell against the wall with the neighbours a, b and c, then we get the formula as shown below or the Z3 equivalent `PbEq([a, b, c], 2)`.

$$(a \land b \land \neg c) \lor (a \land \neg b \land c) \lor (\neg a \land b \land c)$$

This example construction shows how we can create the boolean formula corresponding to part 1. The other two parts use similar logic. For part 2 we need to create a boolean formula for every cell c which makes sure that in the set of visible cells from c there is at least one cell with a light, this way every cell is lit up at least once. The visible cells for a given cell c are thus all the cells which would lit up c if a light would be placed on it, so the visible cells for a cell c always includes at least c itself. To give an example, if the visible cells for a given cell c are a, b and c, we get the formula $a \lor b \lor c$ or the Z3 equivalent `Or([a, b, c])`.

For part 3, take every row/column and chop it into parts wherever there is a wall. So a row of 5 cells with a wall at cell 2 will be chopped into the sets $\{1\}$ and $\{3, 4, 5\}$. For each of these sets, we add a constraint saying that at most one of the variables is allowed to be true. This makes sure that we never have lights illuminating each other. If we have a set consisting of a, b and c we get the formula shown below or in Z3 `PbLe([a, b, c], 1)`.

$$(a \land \neg b \land \neg c) \lor (\neg a \land b \land \neg c) \lor (\neg a \land \neg b \land c) \lor (\neg a \land \neg b \land \neg c)$$

All of these boolean formula's can be connecting using $\land$'s to get one big formula. If this final formula, consisting of part 1 to 3, is satisfiable then the variables which are true provide the solution for the translated puzzle.

Finding multiple solutions can also be done using the SAT solver. In order to get more solutions we repeat the process of finding one solution, but every time we find a solution we add a new clausule to the final formula. This new clausule will make sure that we only search for different solutions than the ones found before. The new clausule for a solution where, say a, d and f are true is $\neg(a \wedge d \wedge f)$, translated to Z3 as `Not(And([a, d, f]))`. Adding this clausule will make sure that if another solution is found, it will be different from the solution(s) found before. Repeating this until the formula is unsatisfiable will provide all solutions to the given puzzle.

The SAT solver can also be used in combination with the trivial solver. The trivial solver is used as a preprocessor providing lights and implacable cells to the SAT solver. The lights, say a, b and c, can be added to the formula using a conjunction $a \wedge b \wedge c$ or in Z3 as `And(a, b, c)`. The implacable cells, say d, e and f, can be added to the formula as follows $\neg(d \vee e \vee f)$ or in Z3 as `Not(Or(d, e, f))`. These constructions are also useful for the verification of the gadgets, see section 3.1.7.

4.1.3 Backtrack solver

The backtrack solver is based on a depth first search algorithm. It is extremely beneficial here to use the trivial solver to preprocess the puzzle and reduce the size of the search tree. A basic overview of the backtrack algorithm can be seen in block 5 and 6.

Algorithm 5 Backtrack Function

```
 1: function BACKTRACK(puzzle, candidates, solution, solutions)
 2:     if puzzle is solved then
 3:         append solution to solutions
 4:         return
 5:     end if
 6:     if no candidates or a wall is unsatisfiable then
 7:         return
 8:     end if
 9:     place a light bulb on the next candidate
10:     append candidate to solution
11:     BACKTRACK(puzzle, candidates, solution, solutions)
12:     remove light bulb from the candidate
13:     remove candidate from solution
14:     BACKTRACK(puzzle, candidates, solution, solutions)
15: end function
```

Algorithm 6 Backtrack Solver

```
Require: puzzle
Ensure: solutions
 1: find base solution using the trivial solver
 2: sort the light bulb candidates
 3: BACKTRACK(puzzle, candidates, solution, solutions)
 4: return solutions
```

When solving a puzzle with the backtrack algorithm we start by using the trivial solver which provides a base solution. We could add a check after line 1 in block 6, in case the trivial solver determines there is no possible solution. Next, we sort the cells which could have a light bulb on it, the candidates (line 2), such that all cells adjacent to walls with a number constraint are first. Because satisfying the number constraints is the difficult part.

After this preprocessing we can run the recursively defined depth first search function seen in block 5. The first base case (line 2-5), checks whether the puzzle is solved, if so the solution will be added to the solutions list. The second base case (line 6-8), checks whether the puzzle is solvable, which is not the case if there are no candidates left (in the not solved state) or if there exists a wall which is unsatisfiable. An unsatisfiable wall has a number constraint n, but less than n candidates adjacent to it.

The recursive part of the backtrack function consists of two steps. First (line 9-11), we run the backtrack function with a light bulb placed on the next candidate. Second (line 12-14), we run the backtrack function with no light bulb placed on the next candidate. Using this approach we enumerate all the possible light bulb placements on the puzzle.

Once the backtrack function is finished we will have found all the possible solutions for the given puzzle. It is often not desired to find all solutions, but some specific amount n (e.g. one to check solvability and two to check uniqueness). The backtrack function can be easily adjusted by adding an extra base case which checks whether the size of the list containing solutions is equal to n in which case we immediately return. This way we can use the backtrack solver described here for all practical purposes.

4.1.4 Performance

The performance difference between the SAT solver and the backtrack solver is quite significant. We do not provide an extensive analysis, but rather a short impression of how the performance differs.

The SAT solver is *by far* the best solver in this comparison. Solving a 40×40 puzzle takes under one second, solving a 30×30 puzzle under half a second and solving a 14×14 puzzle about a tenth of a second. The backtrack solver takes a few seconds up to minute(s) when solving a simple 14×14 puzzle, so we have a huge difference. The solvers do seem to perform similarly on the 7×7 puzzles.

This enormous difference in performance is the consequence of all the hard work people have put into optimizing Z3, turning it into a high speed SAT solver, while the backtrack solver was more so a proof of concept and is not optimized too much. Putting more effort in the backtrack solver could increase its performance, possibly making it a competitor to the SAT solver.

Interestingly, if the SAT solver is provided with an (almost) completely empty puzzle, meaning it barely has any walls (of any kind), it takes a very long time to solve the puzzle, even though to a human, it would be trivial to solve. The backtrack solver outperforms the SAT solver in this case. It is unclear why this happens.

4.2 Puzzle generator

In this section we describe how new Akari puzzles can be generated. All of the generated puzzles have one unique solution. A general overview of the algorithm is shown in block 7.

Algorithm 7 Puzzle Generator

```
Require: $w$ (width) $\geq 1$, $h$ (height) $\geq 1$, $s$ (start) $\geq 0$, $n$ (step) $\geq 1$
Ensure: puzzle solution is unique
 1: create an empty puzzle of size $w \times h$
 2: place $s$ new wall(s)
 3: solve puzzle and place numbers on all walls
 4: while puzzle solution is not unique do
 5:     remove all numbers
 6:     place $n$ new wall(s)
 7:     solve puzzle and place numbers on all walls
 8: end while
 9: remove number constraints invariant under solution uniqueness
10: return puzzle
```

When generating a new puzzle we start of with an empty puzzle, i.e. one containing only empty cells and no walls. Next we immediately start placing down s number of walls using a heuristic which is explained in section 4.2.1. Since an empty puzzle generally does not have a unique solution we place an initial number of walls to greatly speed up the generation process. When s is big the generation of the puzzle will be faster, however the puzzle will generally also contain more walls, which might make the puzzle easier.

After placing down some initial walls (line 2), *without* number constraints, we search for a solution (line 3), which it has by definition, see also section 3.1.1. We use this solution to give *every* wall a number constraint (line 3). So every wall gets the number constraint corresponding to the number of light bulbs around it. This will give us the most restrictive puzzle we can get with the current wall placement. Meaning, if there is no unique solution to the puzzle, we *have to* place more walls, since we already use all number constraints possible.

So now we start the while loop (line 4-8), where we check if there is a unique solution to the puzzle. While there is no unique solution we remove all the number constraints from the walls (line 5), place n new walls (line 6, analogous to line 2, also see section 4.2.1), solve the puzzle and add number constraints to all the walls (line 7, analogous to line 3). We continue this process until we do have a puzzle with one unique solution.

Once we have a puzzle with a unique solution we do some last post processing after which the puzzle is done. For every wall (all having a number constraint) we check if the puzzle still has a unique solution when we remove the number constraint. If the puzzle stays unique, we remove the number constraint, else we keep the number constraint. This is all described at line 9, but to make it clearer also in block 8.

Algorithm 8 Remove some number constraints.

```
1: for wall with a number constraint do
2:     remove number constraint
3:     if puzzle solution is not unique then
4:         restore number constraint
5:     end if
6: end for
```

We are now left with a puzzle which necessarily has a unique solution. Note that it is important to have a sufficiently fast solver in order to check the uniqueness of the puzzle every iteration of the while loop, the requirements depend on the size of the puzzle you are generating. Using the Z3 implementation generating a puzzle of 40×40 takes about 5 minutes, when choosing optimal values for s and n.

At last, since the puzzles could also be to interest to people, we can make the puzzles more neat looking. This is easily done by introducing different symmetries into the puzzle. In this case, when we place a new wall (following the process described above and using the heuristic from 4.2.1), we can simply mirror this wall over, for example, the x or y axis of the puzzle. This is an easy way to make the puzzle more appealing to people.

4.2.1 Heuristic

Creating a heuristic for placing the walls greatly improves the quality of the puzzles generated. This quality is determined by the sparsity of walls. The heuristic takes two kinds of data into account: proximity of walls and differing solutions (if solutions are available, not at line 2 of block 7).

Every cell is assigned a value based on the heuristic. From all of these values we create a distribution assigning to each cell a value between 0 and 1. All cell values together add up to 1. We use the distribution to semi randomly select a cell to place a new wall. Cells classified as good by the heuristic have a bigger chance to get chosen, compared to bad cells.

The current wall placement is evaluated so a new wall will be chosen preferably not in the proximity of other walls. Every cell c is assigned a value based on the walls surrounding c. Depending on how far this wall is from c it accounts for a different value. All values from the walls surrounding c add up to a sum total providing a heuristic for c.

A higher value for the heuristic of c is worse than a lower value. We look at all cells which have a Manhattan distance d with $d \leq 2$ from c. Every empty cell is assigned 0 and cells outside of the puzzle are assigned

0.1 to prevent walls from clutching up at the edge of the puzzle. Walls in horizontal or vertical alignment with c are valued at 1, all other walls are valued at 0.5. This way we stimulate diagonal placement of walls with respect to one another, which seems to be better for generating puzzles with a small number of walls but a unique solution.

When a puzzle has differing solutions, we can use these differing solutions to select walls which have a higher probability of eliminating the differing solutions. Using this information we can speed up the divergence to a puzzle with a unique solution significantly. Consider, for example, a 40×40 puzzle which has exactly two differing solutions with a variation of light placement on the bottom right corner. Now we only have to place a wall somewhere in this corner to make the puzzle unique, placing walls on other parts of the puzzle is completely unnecessary.

Implementing this is fairly easy. Since the heuristic for c should be better when lower, we can take the minimum distance to any of the cells which are different in the two solutions. If c is close to one of these cells we would get a low value. If c is far away from all of these cells we would get a high value. The proximity can again be calculated using the Manhattan distance.

Using the two heuristics described here, and combining them wherever possible, provides us with way sparser puzzles than when no heuristic is used. Yet all of these puzzle have a unique solution.

Using the algorithm described in this section to generate puzzles we have generated an example puzzle, see figure 4.2.

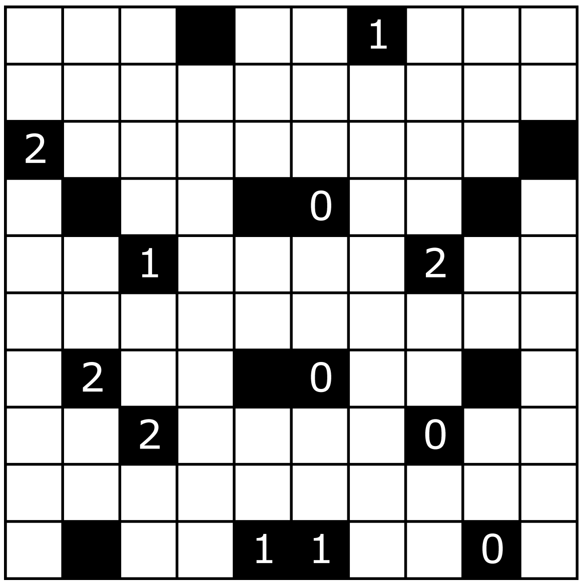


Figure 4.2: Generated puzzle example.

Chapter 5

Related Work

Table 5.1 provides a concise overview of how logic puzzles similar to Akari have been proven to be NP-complete.

Puzzle	**Reduction from**	**Author(s)**	**Reference**
Akari	Circuit-SAT	McPhail	[15]
Battleship	Bin Packing	Sevenster	[16]
Corral	Planar 3-Color	Friedman	[8]
Futoshiki	Partial Latin Square	Maarse	[13]
Hashiwokakero	Hamiltonian Cycle	Andersson	[2]
Heyawake	3-SAT	Holzer, Ruepp	[10]
Hidato	Hamiltonian Cycle	Biasi	[4]
Kakuro	1-in-3-SAT	Takahiro	[17]
Kurodoko	1-in-3-SAT	Kölker	[11]
Masyu	Hamiltonian Cycle	Friedman	[9]
Nonogram	Planar NCL	van Rijn	[18]
Numberlink	3-SAT	Adcock, et al	[1]
Nurikabe	Circuit-SAT	Holder, et al	[12]
Shakashaka	Planar 3-SAT	Demaine, et al	[6]
Slitherlink	Hamiltonian Path	Takayuki	[21]
Sudoku	Partial Latin Square	Takayuki, Takahiro	[22]
Takuzu	Planar 3-SAT	Biasi	[14]
Tatamibari	Planar 3-SAT	Adler, et al	[3]

Table 5.1: Overview NP-completeness proofs of logic puzzles.

Chapter 6

Conclusions and Future

We have shown NP-completeness of three Akari variants and provided polynomial-time solve algorithms for two other variants. We were unable to provide a time complexity for one of the variants, this could be a starting point for further work. We have implemented a rather fast SAT solver, a backtrack solver and a trivial solver which is used by the backtrack solver as preprocessor. Further research could be done on solving Akari puzzles using approximation algorithms, such as [7]. At last, we have made a puzzle generator which can be used to create a large quantity of puzzles with exactly one unique solution.

Bibliography

[1] Aaron Adcock, Erik D. Demaine, Martin L. Demaine, Michael P. O'Brien, Felix Reidl, Fernando Sánchez Villaamil, Blair D. Sullivan. Zig-Zag Numberlink is NP-Complete. *Journal of Information Processing*, 23:239–245, 2015.

[2] Daniel Andersson. Hashiwokakero is NP-complete. *Information Processing Letters*, 109:1145–1146, 2009.

[3] Aviv Adler, Jeffrey Bosboom, Erik D. Demaine, Martin L. Demaine, Quanquan C. Liu, Jayson Lynch. Tatamibari is NP-complete. `https://arxiv.org/abs/2003.08331`. [Online; accessed 23 November 2020].

[4] Marzio De Biasi. Hidato is NP-complete. `http://www.nearly42.org/cstheory/hidato-is-np-complete`, 2013. [Online; accessed 23 November 2020].

[5] Puzzle Team Club. Light Up. `https://www.puzzle-light-up.com`, 2020. [Online; accessed 27 October 2020].

[6] Erik D. Demaine, Yoshio Okamoto, Ryuhei Uehara, Yushi Uno. Computational Complexity and an Integer Programming Model of Shakashaka. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E97.A:1213–1219, 2014.

[7] M. Fitzsimmons and H. Kunze. Combining Hopfield neural networks, with applications to grid-based mathematics puzzles. `https://pubmed.ncbi.nlm.nih.gov/31254770/`, 2019. [Online; accessed 21 December 2020].

[8] Erich Friedman. Corral Puzzles are NP-complete. `https://www.researchgate.net/publication/2529937_Corral_Puzzles_are_NP-complete`, 2002. [Online; accessed 23 November 2020].

[9] Erich Friedman. Pearl Puzzles are NP-complete. `https://www.researchgate.net/publication/2532689_Pearl_Puzzles_are_NP-complete`, 2002. [Online; accessed 23 November 2020].

[10] M. Holzer and O. Ruepp. The troubles of interior design-a complexity analysis of the game heyawake. In *FUN*, 2007.

[11] Jonas Kölker. Kurodoko is NP-Complete. *Journal of Information Processing*, 20:694–706, 2011.

[12] M. Holzer, A. Klein, M. Kutrib. On The NP-Completeness of The Nurikabe Pencil Puzzle and Variants Thereof. `https://www.semanticscholar.org/paper/On-The-NP-Completeness-of-The-Nurikabe-Pencil-and-Holzer-Klein/4855b7160c651c8cc883def72348463fd77cdbed`, 2008. [Online; accessed 23 November 2020].

[13] Mieke Maarse. The NP-completeness of some lesser known logic puzzles. `https://dspace.library.uu.nl/bitstream/handle/1874/383821/Scriptie_Mieke_Maarse_5750032.pdf?sequence=2&isAllowed=y`. [Online; accessed 23 November 2020].

[14] Marzio De Biasi. Binary Puzzle is NP-complete. `https://www.researchgate.net/publication/243972408_Binary_Puzzle_is_NP-complete`. [Online; accessed 23 November 2020].

[15] Brandon McPhail. Light Up is NP-complete. `https://www.researchgate.net/publication/249927572_Light_Up_is_NP-complete`, 2005. [Online; accessed 23 November 2020].

[16] Merlijn Sevenster. Battleships as decision problem. *ICGA Journal [Electronic]*, 27:142–149, 2004.

[17] Seta Takahiro. The complexities of puzzles, cross sum and their another solution problems (ASP). `http://www-imai.is.s.u-tokyo.ac.jp/~seta/paper/senior_thesis/seniorthesis.pdf`, 2002. [Online; accessed 23 November 2020].

[18] Jan van Rijn. Playing Games: The complexity of Klondike, Mahjong, Nongrams and Animal Chess. `http://www.liacs.nl/assets/2012-01JanvanRijn.pdf`, 2012. [Online; accessed 23 November 2020].

[19] Wikipedia contributors. Circuit satisfiability problem — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Circuit_satisfiability_problem&oldid=983101933`, 2020. [Online; accessed 14 October 2020].

[20] Wikipedia contributors. Functional completeness — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Functional_completeness&oldid=967074051`, 2020. [Online; accessed 14 October 2020].

[21] Yato Takayuki. On The NP-Completeness of the Slither Link Puzzle. `https://www.researchgate.net/publication/247043452_On_the_NP-completeness_of_the_Slither_Link_Puzzle`, 2000. [Online; accessed 23 November 2020].

[22] Yato Takayuki, Seta Takahiro. Complexity and Completeness of Finding Another Solution and Its Application to Puzzles. `http://www-imai.is.s.u-tokyo.ac.jp/~yato/data2/SIGAL87-2.pdf`. [Online; accessed 23 November 2020].

Appendix A

Appendix

A.1 Puzzle storage

We use three different encodings for storing puzzles in files. Every encoding has its own advantages and disadvantages. The codex encoding is used for bulk storage of generated puzzles. The grid encoding is used for storage of handwritten puzzles, i.e. the gadgets. The annotated encoding is used for storage of the gadgets ready to be completely automatically verified.

A.1.1 Codex

The puzzles are efficiently stored as strings in files. Since most of the puzzle usually consists of empty cells we shorten the notation based on the number of empty cells we have following each other. The cells are read starting from the top left reading row by row. n consecutive empty cells are encoded into letters, with `a` $= 1$, `b` $= 2$, `c` $= 3$, etc. If $n = 27$, write down a `z` and start over with `a` and $n = 1$. Empty walls are encoded as `B` and numbered walls are encoded as their respective number. See figure A.1 for an example.

The measurements of the puzzle are not encoded in this string and need to be provided separately when converting a puzzle from a string.

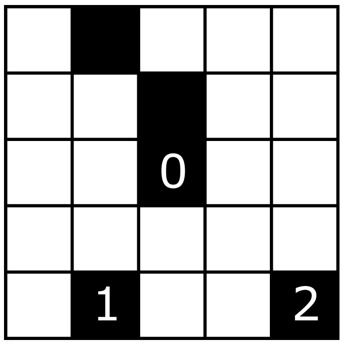


Figure A.1: Puzzle storage example `aBeBd0h1b2`.

A.1.2 Grid

The codex encoding described in section A.1.1 is efficient, but not so user friendly. We have made another type of encoding for the puzzles using a grid format. This format is intuitive and lets us create a grid in a file which can be read into our program for further analysis.

In this format every `-` indicates an empty cell, `*` is a wall without number and `0`, `1`, `2`, `3` and `4` represent a wall with the respective number. On the right we show the representation of the puzzle shown in figure A.1 using this format.

```
-*---
--*--
--0--
-----
-1--2
```

A.1.3 Annotated

The verification of the gadgets as described in section 3.1.7 requires information on where the input and output cells are for every gadget. We obviously do not want to manually enter coordinates of these cells, so we came up with an annotated version of the grid encoding as described in section A.1.2.

Every gadget can have up to two inputs and outputs. The input variables are marked as x and y, the output variables are marked as p and q. Every variable refers to two positions on the grid, one which would indicate the variable to be true, and one which would indicate the variable to be false (analogous to the x and x' used in section 3.1).

The true-cell is indicated using an uppercase letter, the false-cell uses a lowercase letter. So the variable x is annotated as `X` (true-cell) and `x` (false-cell) in this encoding. The input an output lists returned always have the format as shown below.

```
inputs  = [(X, x), (Y, y)]
outputs = [(P, p), (Q, q)]
```

All the annotated files cannot have y while not having x, and cannot have q while not having p. If a variable, say, q is not present in the file then the coordinates of this variable would be left out from the corresponding list, so in this case the output list would be `[(P, p)]`.

Below we show two examples, first we show the grid file, second we shown the annotated file. The examples correspond to figure 3.3c and 3.6c respectively. Note that all the arrows in the original pictures are empty cells as explained in section 3.1.7.

```
*******    *******
--1-1--    Xx1-1Pp
***-***    ***-***
*******    *******
```

```
*-***    *q***
*-***    *Q***
*2-**    *2-**
--2--    Xx2Pp
*****    *****
```