Electronic Preprint for Journal of Information Processing Vol.20 No.3

Regular Paper

Kurodoko is NP-Complete

Jonas Kölker[1,a)]

Received: August 1, 2011, Accepted: March 2, 2012

Abstract: In a Kurodoko puzzle, one must colour some squares in a grid black in a way that satisfies non-overlapping, non-adjacency, reachability and numeric constraints specified by the numeric clues in the grid. We show that deciding the solvability of Kurodoko puzzles is **NP**-complete.

Keywords: puzzles, Combinatorics, Computational Complexity, NP-completeness

1. Introduction and Definitions

Pen-and-paper puzzles are a popular pastime. Many puzzles are available in book form [3], as web applications [7] and as downloadable software applications [6].

A recent survey paper [1] reviews the complexity results of many such pen-and-paper puzzles, in addition to the complexity of games and the relationship between those complexities. Many commonly played puzzles are NP-complete. Also, for many kinds of puzzles it is NP-complete to find a second solution to an instance, given a first solution. This *Another Solution Problem* hardness is studied in Ref. [8]. Another collection of hardness results is Ref. [2].

One particular puzzle for which the hardness is not known is that of Kurodoko. According to Ref. [7] Kurodoko was invented by the Japanese publisher Nikoli [4]. The name comes from "kuro" meaning black and "doko" meaning where, i.e., "where [are the] black [squares]?." We will show that solving Kurodoko is **NP**-complete. From a bird's eye view, our proof is very much similar to many other puzzle hardness proofs, in that we produce subpuzzles which capture some part of the problem we reduce from, then combine these subpuzzles in ways that make the global solutions derived from local solutions correspond to solutions to the problem we reduce from.

Kurodoko is played on a rectangular grid of size $w \times h$, $V := \{0, \ldots, w-1\} \times \{0, \ldots, h-1\}$. The squares are initially blank, except for a subset of squares $C \subseteq V$ which contain clues, integers given by a function $N\colon C \to \{1, \ldots, w+h-1\}$. We can think of the rectangular grid as a grid graph $G = (V, E)$, where $(v, v') \in E$ if and only if v and v' are horizontally or vertically adjacent, i.e., $E := \{((r, c), (r', c')) \mid |r - r'| + |c - c'| = 1\}$.

The player's task is to come up with a set of black squares $B \subseteq V$, the rest being white, $W := V \setminus B$, such that the following four rules are satisfied:

(1) The clue squares are all white, $C \subseteq W$ (or equivalently, $B \cap C = \emptyset$).

(2) None of the squares in B are adjacent to any other square in B. i.e., $(B \times B) \cap E = \emptyset$.

(3) All white squares are connected via paths of only white squares, i.e., the induced subgraph on the white squares $G_W = (W, E \cap (W \times W))$ is connected.

(4) The number at each clue square equals the number of white squares reachable from that square, going in each of four compass directions and never off the board nor through a black square, i.e., $\forall (r, c) \in V : N(r, c) = hz + vt - 1$ where hz is the length of the longest horizontal run of white squares going through (r, c), i.e., $hz := \max\{k \in \mathbb{N} \mid \exists b : 0 \le b \le c \le b + k - 1 < w \wedge \{(r, b + i)\}_{i=0}^{k} \subseteq W\}$. Similarly, vt is the length of the longest such vertical run. We say that (r, c) *touches* the squares in these runs, including itself.

We show that the Kurodoko Decision Problem, "given w, h, C and N and, is there a set $B \subseteq V$ satisfying the above criteria?," is **NP**-complete. We take w and h to be represented as binary encodings, C as a list of points (vertices) and N as a list of integers; the i'th integer in the list representing N is the function's value at the i'th point in the list representing C.

To help the reader gain an understanding of these rules and their implications we offer a simple observation:

Theorem 1. *If (w, h, C, N) is solvable and $\exists v \in C : N(v) = 1$, then $1 \in \{w, h\}$ and $w + h \le 4$.*

Proof. Let (w, h, C, N) be given and suppose that $w, h \ge 2$. Let v be given with $N(v) = 1$. Then, since $w \ge 2$, either v has a right or left neighbour, or both; let us call any one such neighbour v'. Since $h \ge 2$ this neighbour v' has a neighbour below or above, or both. In either case, call such a neighbour v''. By rule 4, all neighbours of v must be black. By rule 2, all those neighbours' neighbours must be white, including v''. But since v is surrounded by black squares, there cannot be a path from v to v'' that steps on only white squares, violating rule 3. Therefore, w and h can't both be at least 2; assume by symmetry that $h < 2$. It cannot be that $h = 0$ or there would be nowhere for v to be found ($V = \emptyset$), so $h = 1$. Assume $w > 3$; then v must either be a corner square with a black neighbour, or have two black neighbours, at least one of which has a white neighbour that isn't v. In either case, there is

1 Department of Computer Science, Aarhus University, Aarhus, Denmark
a) epona@cs.au.dk

at least one white square that doesn't have a white-only path to v, which rule 3 requires. So $w \leq 3$, and hence $w + h \leq 4$. □

2. Proof of NP Membership

We are now ready to state and prove the first part of our claim of **NP**-completeness.

Theorem 2. *The Kurodoko Decision Problem is in **NP**.*

Proof. We show there is a polynomial time witness-checking algorithm. The witnesses will be solution candidates, i.e., candidates for B represented as a list of points. Given such a list, we can easily verify rule 1 in time $O(|B|\,|C|)$, by two nested loops. We can also verify rule 2 easily in time $O(|B|^2)$ by testing for every (r, c) and (r', c') in B that $|r - r'| + |c - c'| \neq 1$. Verifying rule 4 is also fairly easy: for each clue square v, find the closest black square in each of the four compass directions. Then, compute $hz + vt - 1$ and check that it equals $N(v)$. This takes time $O(|C|\,|B|)$.

To check rule 3 if $w > 2\,|B| + 1$ or $h > 2\,|B| + 1$, compress the grid by merging adjacent rows that don't contain any black squares and do the same for columns. Then $w, h \leq 2\,|B|+1$. Find a white square by going through each square until a white square is found. If no white square is found, accept if and only if $w = h = 1$ (if not $w = h = 1$ then rule 2 is violated). If a white square v_w is found, verify by DFS that each white square has a path to v_w (or equivalently, that the number of white squares reachable from v_w plus the number of black squares equals the total number of squares). Accept if and only if this is satisfied. □

We note that if w and h were given in unary then the compression step would not be necessary since the size of the board would be polynomial in the length of the input. The rest of this article is devoted to proving the next theorem.

Theorem 3. *The Kurodoko Decision Problem is **NP**-hard.*

3. Overview of the Hardness Proof and Reduction

We prove the **NP**-hardness of Kurodoko-solvability, by showing how to compute a reduction from one-in-three-SAT. This problem was shown to be **NP**-complete in Ref. [5].

We do the reduction by describing a set of *gadgets*, a set of 17×17 square-of-squares containing clues, which are very amenable to combination and act in circuit-like ways, e.g., as wires, bends, splits, sources, sinks and so forth. These are combined to form three kinds of *components*: one kind acting like SAT variables, a second kind acting like SAT clauses, and a third kind acting like a matching between the components of the two first kinds, such that each clause component gets routed to the components corresponding to exactly those variables referenced in the clause. When taken as a whole we refer to the gadgets as the *board* (as in "printed circuit board").

We will show that in every solution to the Kurodoko instance resulting from the reduction, the gadgets will behave according to fairly simple descriptions which capture their circuit properties (e.g., wires behave like the identity function). This in turn implies that the components each behave according to a description which captures the connection to that SAT problem, such that SAT problem must have a solution.

For the reverse direction, we show how to compute a Kurodoko solution from a given SAT solution, and verify that this Kurodoko solution is indeed a solution, i.e., that it satisfies the four rules which define solutions.

4. The Reduction

In this section, we describe in more detail first the gadgets, then how to manipulate and combine them into components. Next, we describe which components we combine gadgets into, how the components are combined into boards, and finally how to handle a deferred issue which requires global information about the board. In Appendix A.1 we provide an implementation of the reduction in order to ensure there is no ambiguity.

4.1 Gadgets

The reduction uses nine different gadgets named Wire, Negation, Variable, Zero, Xor, Choice, Split, Sidesplit and Bend. As an example see **Fig. 1** for an illustration of the Wire gadget.

Diagrams of the remaining gadgets are provided in Appendix A.2. Note that in all of them, the outermost rows and columns are either empty, or contain the centered sequence {a question mark, no clue, the clue 2, no clue, a question mark}.

Formally, we can think of the Wire gadget as a Kurodoko instance $wire = (17, 17, C, N)$, where C is the set of clue squares and N is the set of clue values. Some squares $v_?$ contain question marks; formally, we give them the value 1 for now. Their true value will be established once the pattern of how gadgets are combined is known, which depends on the SAT instance given to the reduction. When we speak of deductions about the board, we mean with the true value in place.

The eight marked squares $(\{n, w, e, s\}_{\{i,o\}})$ are not elements of C, and are in fact not a part of the gadget. The behavior of the gadgets can be succinctly described in terms of the colours of these squares. We think of d_i as an input square (for $d = n, w, e, s$) and d_o as an output square. In some sense we want to think of the square in the center row or column immediately outside the gadget as the true output square, but this square will have the same colour as d_o. We treat black as 1 and white as 0. With this, we offer a description of the gadgets:

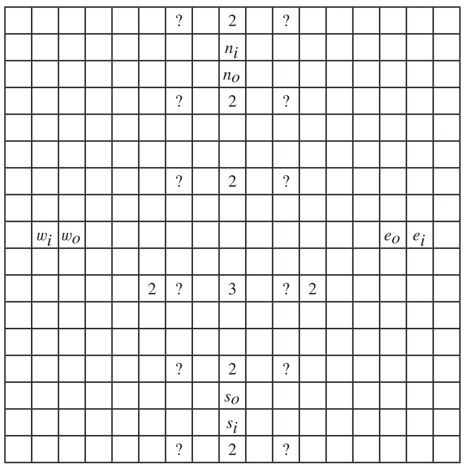


Fig. 1 The Wire gadget.

- Wire: $n_o = s_i$: the north output equals the south input.
- Negation: $n_o = 1 - s_i$: the north output is the opposite of the south input.
- Variable: $n_o \in \{0, 1\}$: the north output is always 0 or 1.
- Zero: $n_o = 0$: the north output is always 0.
- Xor: $n_o = e_i \oplus w_i$, the north output equals the xor of the east and west inputs.
- Choice: $w_i + s_i + e_i = 1$, exactly one input is 1.
- Split: $w_o = e_o = s_i$: the west and east output equals the south input.
- Sidesplit: $n_o = e_o = s_i$: the north and east output equals the south input.
- Bend: $e_o = s_i$: the east output equals the south input.

We claim now—and prove later—that in any instance created by our reduction, these relationships have to hold or the instance doesn't have a solution.

4.2 Manipulation and Combination of Gadgets

We note that the gadgets are square and can be mirrored and rotated. For compactness, let us introduce some notation for this: if G is a gadget, then G^+ is G rotated 90° clockwise, G^- is G rotated 90° counterclockwise, G^2 is G rotated 180° and $\overline{G}$ is G mirrored through a vertical line. We call gadgets by their first letter, except for Split which we refer to as T (in Appendix A.2, it looks like a tee).

Rotating and mirroring gadgets has the expected consequences to their operation. For example, if we mirror and then rotate clockwise a Bend gadget, the result is a gadget that takes an input on the west edge and outputs this on the north edge, i.e., $n_o(\overline{B}^+) = w_i(\overline{B}^+)$. Note that this is different from $\overline{B^+}$, where $s_o = e_i$.

Note that if an input square x_i is adjacent to a clue 2, then its adjacent output square x_o must contain the opposite colour of the input square, $x_i \oplus x_o = 1$, or else rule 2 (two blacks) or rule 4 (two whites) would be violated. This implies a curious property of the wire gadget: $s_o = 1 - s_i = 1 - n_o = n_i$; that is, it works in the other direction too. The same is true for Bend and Negation. For the Xor gadget, we see that $n_o = e_i \oplus w_i \Rightarrow n_o \oplus e_i = w_i \Rightarrow (1 \oplus n_o) \oplus e_i = 1 \oplus w_i \Rightarrow n_i \oplus e_i = w_o$.

This is why there is no Side-Xor: for X^+ we have $n_o = s_i \oplus e_i$, i.e., X^+ works as a hypothetical Side-Xor would. Likewise, X^- behaves as a $\overline{\text{Side-Xor}}$, in that $n_o = s_i \oplus w_i$.

Let us next define what it means to combine multiple gadgets. Let's say we have a set $K = \{(g, r, c)_i\}_{i=1}^k$ of k gadgets, the i'th gadget $g = (w_g, h_g, C_g, N_g)$ having coordinates $(r, c) \in \mathbb{N}^2$ on the combined board. Then, in the combined instance,

$$h_K = \max\{16r + w_g \mid ((w_g, h_g, C_g, N_g), r, c) \in K\}$$
$$w_K = \max\{16c + h_g \mid ((w_g, h_g, C_g, N_g), r, c) \in K\}$$
$$C_K = \bigcup_{(g,r,c)\in K} \{(16r + i, 16c + j) \mid (i, j) \in C_g\}$$

Furthermore, for each $(r, c) \in C_K$ where $r = 16i + r'$ and $c = 16j + c'$ and $0 \le r', c' \le 16$ and $(g, i, j) \in K$ we have $N_K(r, c) = N_g(r', c')$.

In other words the gadgets are placed next to one another, such that the rightmost column of each gadget overlaps the leftmost column of its right neighbour, and similarly in each other compass direction. The set of clues in the instance (w_K, h_K, C_K, N_K) is the union of the clues in each of the gadgets, suitably translated. For this to be well-defined, the clue values in the overlap areas must agree between the two overlapping gadgets.

If two overlap areas are empty (in both gadgets), this clearly isn't an issue. If one gadget is non-empty in the overlap, it simply "overwrites" the (overlap-)empty gadget, although this will never happen. If they are both non-empty, then they are in fact equal in the overlap area, per our previous remark about their outermost row and column structure. So this will always be well-defined.

4.3 Components and the Board

We have just seen how to combine a collection of gadgets K into an instance (w_K, h_K, C_K, N_K). As it happens, components are just such instances. Let us consider as an example the clause component. It has several variants; we show first the variant corresponding to a clause (x, y, z):

B	W^+	C	W^-	$\overline{B}$
W		W		W

Next the variant for a clause $(x, \neg y)$:

B	W^+	C	W^-	$\overline{B}$
W		N		W

Finally the same clause, with variables gadgets added to show the component in a context:

B	W^+	C	W^-	$\overline{B}$
W		N		W
V_x		V_y		Z

Formally speaking, the first clause component can be described as a combination (by the above rule) of $\{(B, 0, 0), (W^+, 0, 1), (C, 0, 2), (W^-, 0, 3), (\overline{B}, 0, 4), (W, 1, 0), (W, 1, 2), (W, 1, 4)\}$.

Hopefully it is clear how these work. Assuming the W- and N-gadgets have neighbours to the south which provide input (as in the latter example), this input goes through the W or N and either directly into the C gadget, or through a bend that points it towards the C which it reaches after going through either W^+ or W^-. The pattern is this: the top row is identical in all variants of the clause component. In the bottom row, columns one and three are always empty. For a clause with k variables where $k < 3$, the rightmost $3-k$ even-indexed columns contain W-gadgets (these will be connected to a Z input elsewhere). The remaining columns contain W or N, depending on whether the corresponding variable in the clause is negated or not. For instance, the central N is the middle component is there (rather than a W as in the left component) because y is negated. The rightmost W is connected to a Z since the clause only has two variables.

Next we show the zero component. This one is rather simple:

Z

As we will see later, instances of this will be connected to clause components for those clauses that refer to less than three variables.

Next, let us consider the variable component. Its structure depends on how many times the variable is used. We show the structure for variables used once, twice, thrice and four times:

V

S	W^+	$\overline{B}^+$		W		W
S	W^+	W^+	W^+	$\overline{B}^+$		W
S	W^+	W^+	W^+	W^+	W^+	$\overline{B}^+$
V						

S	W^+	$\overline{B}^+$
V		

S	W^+	$\overline{B}^+$		W
S	W^+	W^+	W^+	$\overline{B}^+$
V				

Once again, it should hopefully be clear what happens. The output produced by V is repeatedly split to go both up and to the right. The copies continue to the right until they can be bent upwards and continue up, such that the variable component produces k equal outputs with a non-outputting square between each output.

Also, the structural pattern should be fairly clear: a variable used k times produces a component of height k and width $2k - 1$. The bottom row always has a V in its leftmost square—only this, and nothing more. Other than this, row i from the top has (from left to right) an S, then $2i + 1$ times W^+, then $\overline{B}^+$, then $k - i - 1$ times {nothing, followed by W}.

Finally, we have the routing component. This is built up of multiple copies of two kinds of subcomponents, wires and swaps:

W
W

X^+	T	X^-
S	X	$\overline{S}$

Clearly the (northern) output of the wire component is the southern input. Let a denote the input to S and b the input to $\overline{S}$. Then the output of X is $a \oplus b$. This is fed through T as input to both X^+ and X^-; the input to X^+ is a (from S) and $a \oplus b$ (from T), and so the northern output must be $a \oplus (a \oplus b) = b$. Similarly, the northern output of X^- is a.

In order to best motivate the structure of the routing component, we need to look at the structure of the Kurodoko instance produced by the reduction. In the top, there will be one clause component for every clause in the SAT instance; each pair of adjacent components has one column of "air," unused space, between them. At the bottom, there is one variable component for every variable that occurs in a clause somewhere (with the appropriate number of repetitions built into the variable component), as well as k zero components, where k is the amount of "extra space" in the clauses, $k = 3m - \sum_{j=1}^{m} \left|C_j\right|$, three times the number of clauses minus the total number of literals.

What the routing component does is essentially label each output of the bottom part with the index of its goal column and then sort the signals.

Internally, create an array A (initially $3m$ copies of $\perp$) corresponding to the outputs of the bottom part of the board. For $i = 1, \ldots, 3m$, if the $\lfloor \frac{i}{3} \rfloor$'th clause has at least $(i \bmod 3) + 1$ variables, store i in the leftmost non-$\perp$ entry of A which corresponds to the $(1 \bmod 3 + 1)$'th variable; if not, store i in the leftmost non-$\perp$ entry of A which corresponds to a zero component.

Then, run some comparison-based sorting algorithm which only compares adjacent elements [*1]. For $t = 1, \ldots, t_{\max}$, place a row of swap and wire subcomponents on top of the bottom part of the board, putting swap subcomponents between entries that are swapped at time t and wire subcomponents at entries that are left unchanged at time t. Here, $t_{\max}$ is the smallest number such that the algorithm never makes more than $t_{\max}$ layers of swaps. In the case of the odd-even transposition sorting network, $t_{\max}$ is $3m$, the number of elements to be sorted. Note that once A is sorted we are free to stop early (i.e., the height of the routing component may depend on more features of the SAT instance than just its size).

These are the components. They are combined into a board by the same rule used to combine gadgets into components. To summarise the description of the structure of the board, given a SAT instance: on top there's one clause component for every clause in the SAT instance with at least one variable. At the bottom, there's a variable component for every variable contained in at least one clause, and at the bottom right k zero components for every clause with $3 - k$ variables ($k = 1, 2$). In the middle, there's a routing component. As an example, consider the SAT instance $(\neg v_2, v_1)$. The corresponding Kurodoko instance looks like this:

B	W^+	C	W^-	$\overline{B}$
N		W		W
X^+	T	X^-		W
S	X	$\overline{S}$		W
V		V		Z

Note that every gadget with one or more outputs is placed adjacent to other gadgets that have these outputs as their inputs (and vice versa). In other words, in the areas where gadgets overlap, either both (all) gadgets contain no clues in the overlap, or they do contain clues and the clues match up. Also, no gadget contains clues in the outermost rows or columns of the board.

4.4 Post-processing the Board to Handle $v_?$

In Appendix A.2 we show the gadgets. For each of the gadgets, we can make deductions about the colour of some of the squares, under the assumption that the instance containing the square has a solution. In other words, some squares are necessarily the same colour in all solutions. In this section, *white* (*black*) *squares* refers to squares that are white (black) in all solutions.

For some squares containing a question mark in three directions going out from that square there are black squares within the question mark's gadget (whose blackness can be deduced inde-

*1 Examples include insertion sort, bubble sort and the odd-even transposition sorting network.

pendent of the value in the square with the question mark), while in a fourth *open* direction there are no black squares within the gadget. We call these squares *end squares* (e.g., there are six of those in S). Suppose that we draw a line from each end square in the open direction and its opposite until we hit the edge of the board or a black square. We call these lines *rays*. Then, we want to show:

Lemma 1. *Every square containing a question mark lies on a ray.*

Proof. Observe that some question marks can be seen to lie on a ray just by examining the gadget containing the question mark. Call the rest of the question marks *non-obvious*. Observe (by inspection) that in every gadget G which contains a non-obvious question mark, one can draw one or two lines which are 17 squares long and don't contain any black squares such that every non-obvious question mark lies on one of these lines. It can be seen (again by inspection) that every such line contains two non-obvious question marks which lie in the outermost rows or columns of the gadget, and thus also in the gadgets adjacent to G.

If the question marks can be directly observed to lie on a ray, then the question marks in G must lie on the same ray (observe that the rays point in the right directions for this to follow, i.e., they aren't completely contained within gadget overlap areas). If not, continue into the next neighbour. As argued earlier, this cannot stop by running out of neighbours or running off the board, so this must stop in a gadget containing an end square, with its ray fully containing these 17 square long lines, and thus the non-obvious question marks in G (at least on one of the lines; repeat this argument for the other line, and for all other gadgets). □

As a consequence of this two-directional inductive argument, every ray contains at least (in fact exactly) two end squares. Extend the ray from one end square; it will eventually hit an end square. This means that in each of the four directions out from the end squares, we have found a square that is certainly black. If we assume that every square on the ray is white, then for every end square v and every value of $N(v)$ except one, rule 4 must be violated. Assign to $N(v)$ the remaining value, such that every square on a ray must be white.

Observe that every non-end square $v_?$ with a question mark is adjacent to a black square (in a non-ray direction). As this square is on a ray (by lemma 1), it is surrounded by two end squares and thus also two black squares in the ray direction. There might be a black square in the last direction inside the gadget containing $v_?$. If there is, let $N(v_?)$ be the value such that the other squares in this last direction must be white. If there isn't, let $N(v_?)$ have the unique value such that $v_?$ must touch zero squares in this last direction.

5. Proof of Correctness of the Reduction

We have described how to produce a Kurodoko given a SAT instance. We want to show that the Kurodoko instance has a solution if and only if the SAT instance has a solution. To do this, we first make some observations about the set of solutions to the Kurodoko instance and the properties of its elements.

Consider again the Wire gadget (Fig. 1). In every Kurodoko instance containing it that has a solution, the squares between 2 and ? must be black, or else rule 4 is violated. Their neighbouring squares will have to be white, or rule 2 is violated. Similar deductions can be made around the 2s adjacent to the ?s. This means that in any solution, the wire gadget must look like in **Fig. 2**. The ■ represent squares that are black in all solutions, and · represent squares that are white in all solutions. Not all possible deductions are made—some squares are necessarily white because of the ray property, but these are not marked as such, as this information is not necessary to make the deductions we need. The square marked w must be white, or else the 3 would touch either too few or too many squares and violate rule 4. The squares marked x can be seen to all have the same value, likewise for $\overline{x}$, and the two values must be different. Otherwise, either rule 4 or rule 2 would be violated.

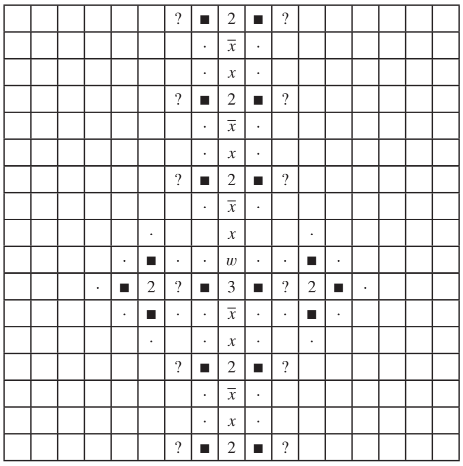


Fig. 2 The Wire gadget.

Also, the square north (in the board) of the northernmost 2 in the Wire gadget ought to be marked x, as it must be consistent with its value. Partial solutions for the remaining gadgets are to be found in Appendix A.2, along with arguments that they work as described previously.

5.1 Proof that Kurodoko Solvability Implies SAT Solvability

We are now ready to prove the following:

Theorem 4. *Let a SAT instance be given, and let $K = (w, h, C, N)$ be a Kurodoko instance produced by the reduction described previously. If K has a solution, then the SAT instance also has a solution.*

Proof. As we just stated, if K has a solution, then the gadgets work as claimed. It is clear from the structure of the components that if the gadgets work as claimed, the components do as well.

Let a solution be given. The reductions specifies an obvious bijection between SAT variables and variable components in K: assign to each SAT variable the value of x in its component (i.e., v_1 is true iff x is black in the leftmost variable component).

Exactly one input to each clause gadget must be black (i.e., true), which by the structure of clause components implies that there is exactly one "good" input to the gadget's containing component, one that is either true (black) and non-negated or false (white) and negated.

The routing component matches the clause components with variable components in the same way clauses are matched with variables in the SAT instance. Since clause components each have one good input, the variable assignment ensures that each SAT clause have exactly one literal that is satisfied. That is, the assignment is a solution of the SAT instance. □

5.2 Proof that SAT Solvability Implies Kurodoko Solvability

Next, we want to establish that if the SAT instance that produces a given Kurodoko instance K has a solution, then K has a solution as well. We will do this by suggesting a solution candidate and then showing that none of the four rules are violated.

Theorem 5. *Let a SAT instance be given, and let K be the Kurodoko instance produced by the reduction when run on the given SAT instance. If the SAT instance has a solution, then K has as solution as well.*

Proof. Let $v \in \{0, 1\}^n$ be a variable assignment which satisfies the SAT constraints. For every live variable i in the SAT, there is one corresponding variable gadget in K; in each such gadget, let x be black if v_i is 1 and white if v_i is 0.

Also, let every square be black if it contains a ■ in its partial solution in Appendix A.2. Let a square be black if the consistency of x and $\overline{x}$ requires it, and let squares be black according to the description of the gadgets (i.e., if two inputs x and y to an Xor are black, let the squares corresponding to xy and $\overline{z}$ be black). For purposes of (opposite) consistency, consider the black square in the next-to-top row in the Zero gadget to be a named square (x, $\overline{x}$, y, etc.), and cross the gadget boundary: if an outermost x of a gadget is white, the square on the other side of the 2 is black and vice versa. Finally, some squares $v_?$ marked ? are not end squares and aren't flanked by two intra-gadget black squares in the partial solutions in Appendix A.2. One of the two squares adjacent to $v_?$ in the non-ray direction is marked ■ in the partial solutions; let the other be black. Let the remaining squares be white.

First, rule 1 is clearly satisfied: one can see by inspection that no square in a gadget marked ■ nor any named square also contains a clue.

Secondly rule 2: no squares marked ■ are adjacent and no square marked ■ is adjacent to a named square. The set of squares adjacent to named (black) squares that could potentially be black are the negations of said named squares, which obviously don't pose a problem, and one square in Xor marked xy: it contains the value $x \wedge y$ and is adjacent to $z = x \oplus y$. Note that if $x \oplus y = x \wedge y$ then $x = y = 0$, so this doesn't violate rule 2 either. So there is no pair of adjacent black squares.

Connectedness is required by rule 3. If we can establish that every square is connected to the top left square(s) of the gadget(s) containing it, we can conclude the rule can't be violated, as due to symmetry and transitivity of the connectedness relation each square in a gadget is connected to every other square in that gadget. In particular, each square is connected to the edge squares, which are connected to every square in the neighbouring gadget. But then we can reach any target square, starting at any other square: go through neighbouring gadgets to the gadget containing the target square, and then (staying inside that gadget, going via the top left corner) go to the target square.

It can be seen by inspection [*2] that every square in a gadget has such a path to the top left corner of the gadget, except for one class of square: a 2 in an outermost row or column, with three adjacent black squares—either three squares marked as black, in the case of Zero, or two such squares and one named square. In both cases, assuming this "trapped" square is on the north edge, the next square north of 2 is white by the oppositional consistency of $(x, \overline{x})$, having considered the "trapping" black square from Zero as a named square. If there was no such trapping black square, not only would the 2 have an intra-gadget path to the top left square, it would also have a five square path to its closest two ?s. When there is such a trapping black square, the 2 therefore has a five square path to its nearby ?s through the adjacent gadget. From there, it then has a path to the top left square of both its containing gadgets.

Lastly, rule 4 requires each clue to touch a number of (white) squares denominated by that clue. The clue squares can be divided into three sets, namely those marked ?, those which are flanked by a named square and its negation (possibly with some squares marked w adjacent on one side), and finally the rest, which (loosely speaking) serve to necessitate the flanking of the squares in the middle group by two black squares.

The last group can be seen to all obey rule 4 by inspecting the partial solutions in Appendix A.2, with one exception: the topmost 2 in Zero, but this clue is fulfilled since we treat the black square on the south as named with respect to oppositional consistency—in other words, as every Z has a gadget to the north of it, the square to the north of this 2 is white, and the square to the north of that is black.

It should be obvious from how the values of $N(v_?)$ are chosen in the reduction that the first group, the squares marked ?, also obey rule 4: the values are chosen such that if every square on a ray is white (which it is), the squares orthogonal to the ray going out from the end squares are white (which they are), and each $v_?$ that isn't "framed" by four necessarily black squares are given black neighbours to frame them (which they are), the clues are satisfied.

Lastly, the clues extended by w and flanked by named squares. These can all fairly straightforwardly be seen to be satisfied by the oppositional consistency of the named squares (and their black flanks). Two notable exceptions are the center squares of every Choice and Xor gadget, respectively. A simple case analysis shows that every choice of x, y and their implied values of xy and z will satisfy the central 3.

Lastly, since v is a solution to the SAT instance, each clause has one true literal. This implies by the structure and combination of the gadgets and components that every choice gadget has one black input. But then it is clear that exactly one of the central $\overline{x}$, $\overline{y}$, $\overline{z}$ are white, satisfying the central clue.

Thus, under the assumption that the given SAT instance has a solution, so does the Kurodoko instance K produced by the re-

[*2] The inspection may in some cases be easier if one cuts the gadgets into quadrants and see each quadrant to be connected, then realises that the dividing lines can be crossed by paths that only contain white squares.

duction. □

6. Discussion of the Reduction, the Result and Future Work

We have seen (with proof) a mapping from SAT instances to Kurodoko instance which preserves solvability. In fact, we have given a map from SAT solutions to Kurodoko solutions. Note, however, that for every SAT solution there are multiple Kurodoko solutions: every clause component contains gadget-free board positions. In such a "null gadget," one can freely choose the colour of the center square (and one in fact has many more degrees of freedom).

This means that the map from SAT solutions to Kurodoko solutions (given our choice of polynomial time witness checking turing machines) isn't injective. One might as future work try to find a reduction from other problems to Kurodoko where there is an injective solution map.

Also, the use of ? is somewhat unsatisfactory: this makes the atomic parts of the reduction depend on how they're combined. It would make for a simpler and more easily understood reduction if this requirement was eliminated.

However: note that each ? value, and in fact each other integer in the representation of the Kurodoko instance produced by the reduction, is at most linear in the size of the SAT instance (and also the Kurodoko instance). In other words, the Kurodoko solvability problem is in fact *strongly* **NP**-complete.

One can do better than linear, though. If one adds kinks to the wire subcomponents (connect four Bends so as to act the same) and adds layers of wire subcomponents between each sorting layer, each ? is on a ray of length $O(1)$. The details of proving this are left as an exercise to the reader.

References

[1] Demaine, E.D. and Hearn, R.A.: Playing Games with Algorithms: Algorithmic Combinatorial Game Theory, *Games of No Chance 3*, Albert, M.H. and Nowakowski, R.J. (Eds.), Mathematical Sciences Research Institute Publications, Vol.56, pp.3–56, Cambridge University Press (2009).

[2] Eppstein, D.: (Personal page) (online), available from ⟨http://www.ics.uci.edu/~eppstein/cgt/hard.html⟩ (accessed 2011-07-28).

[3] Nikoli: Purchase Nikoli Books (online), available from ⟨https://www.nikoli.co.jp/howtoget-e.htm⟩ (accessed 2011-07-28).

[4] Nikoli: Rules of Kurodoko (online), available from ⟨http://www.nikoli.co.jp/en/puzzles/where_is_black_cells/⟩ (accessed 2011-07-28).

[5] Schaefer, T.J.: The complexity of satisfiability problems, *Proc. 10th Annual ACM Symposium on Theory of Computing*, pp.216–226 (1978).

[6] Tatham, S.: Portable Puzzle Collection (online), available from ⟨http://www.chiark.greenend.org.uk/~sgtatham/puzzles/⟩ (accessed 2011-07-28).

[7] Weiss, S.: Kuromasu (online), available from ⟨http://www.lsrhs.net/faculty/seth/Puzzles/kuromasu/kuromasu.html⟩ (accessed 2011-07-28).

[8] Yato, T.: Complexity and completeness of finding another solution and its application to puzzles, Master's thesis, Graduate School of Science, the University of Tokyo (2003).

Appendix

A.1 A python Implementation of the Reduction

We have attempted to give a semi-formal, unambiguous description of the reduction in sufficient detail. However, nothing can be quite as unambigous and sufficiently detailed as an implementation, so we give one.

```
#!/usr/bin/env python

from sys import argv, exit
from itertools import chain

####################

null = None
gadget_size = 17

def unpack(gadget_str):
    k = gadget_size
    gadget = [null for _ in range(k**2)]
    dim, stream = gadget_str.split(':')
    assert dim == '17x17'
    ptr = 0
    for (i, c) in enumerate(stream):
        if c in '_': pass
        elif c.islower(): ptr += 1 + ord(c) - ord('a')
        elif c.isdigit() or c in '?!':
            if c.isdigit(): assert not stream[i+1].isdigit()
            gadget[ptr] = c
            ptr += 1
        else: assert False
    assert ptr == len(gadget)
    return [gadget[i:i+k]
            for i in range(0, len(gadget), k)]

def rotate(old):
    k = gadget_size
    new = [[null for _ in range(k)] for _ in range(k)]
    for r in range(k):
        for c in range(k):
            new[k - 1 - c][r] = old[r][c]
    return new

def flip(old):
    k = gadget_size
    new = [[null for _ in range(k)] for _ in range(k)]
    for r in range(k):
        for c in range(k):
            new[r][k - 1 - c] = old[r][c]
    return new

####################

wire, neg, var, zero, xor, choice, \
    tee, ltee, bend, ngadgets = range(10)

# exclamation marks indicate where 'rays' end.
gadgets = [
    # wire
    "17x17:e!?!2!?!e qq e!?!2!?!e qq e!?!2!?!e qqq "
    "d!2?!3!?2!d qq e!?!2!?!e qq e!?!2!?!e",
    # neg
    "17x17:e!?!2!?!e qq e!?!2!?!e qq e!?!2!?!e qqq "
    "e!?!2!?!e qq e!?!2!?!e qq e!?!2!?!e",
    # var
    "17x17:e!?!2!?!e qq d3!?!2!?!3d d3a!c!a3d"
    "qqqqqqqqqqqq",
    # zero
    "17x17:e!?!2!?!e qq d3!?!2!?!3d d3a!c!a3d h7h"
    "qqqqqqqqqqq",
    # xor
    "17x17:qqqq c!d!d!c !b2d2d2b! ?b?d?d?b? !b!d!d!b!"
    "2b4d3d4b2 !b!i!b! ?b?!g!?b? !b4_4a!c!a4_4b! c!i!c"
    "d!2?!3!?2!d qq e!?!2!?!e",
    # choice
    "17x17:qqqq c!i!c !b2d!d2b! ?b?d?d?b? !b!d!d!b!"
    "2b4d2d4b2 !b!i!b! ?b?!g!?b? !b4_4a!c!a4_4b!"
    "c!i!c d!2?!4!?2!d qq e!?!2!?!e",
    # tee
    "17x17:qqqqq !b!b!c!b!b! ?b?b?c?b?b?"
    "!b!b!c!b!b! 2b2b2c2b2b2 !b!d4d!b! ?b?!g!?b?"
    "!b4_4a!c!a4_4b! c!i!c" "d!2?!3!?2!d qq e!?!2!?!e",
    # ltee
    "17x17:qqqq c!f!f !b2b4c3b!b! ?b?a!2!b?b?b?"
    "!b!b!c!b!b! 2b2b3c2b2b2 !b!d4d!b! ?b?!g!?b?"
    "!b4_4a!c!a4_4b! c!i!c d!2?!2!?2!d qq e!?!2!?!e",
```

```
        # bend
        "17x17:qqqq j!f j2b!b! e2!c?b?b? j!b!b! j2b2b2"
        "e4b4a!b!b! 1!?b? d3a!c!a4_4b! m!c d!2?!3!?2! d"
        "qq e !?!2!?! e",
    ]
gadgets = [s.replace(' ', '') for s in gadgets]
assert len(gadgets) == ngadgets

bend_dr = unpack(gadgets[bend])
bend_dl = flip(bend_dr)
bend_lu = rotate(rotate(bend_dr))

tee_d = unpack(gadgets[tee])
tee_l = rotate(flip(unpack(gadgets[ltee])))
tee_r = flip(tee_l)

xor_r = rotate(unpack(gadgets[xor]))
xor_d = rotate(xor_r)
xor_l = rotate(xor_d)

wire_up = unpack(gadgets[wire])
wire_right = rotate(wire_up)
wire_left = flip(wire_right)

gchoice = unpack(gadgets[choice])
gwnot = unpack(gadgets[neg])
gzero = unpack(gadgets[zero])
variable = unpack(gadgets[var])

xgadgets = [bend_dr, bend_dl, bend_lu, tee_d, tee_l, tee_r, xor_r,
            xor_d, xor_l, wire_up, wire_right, wire_left, gchoice,
            gwnot, gzero, variable]

def main(argv):
    sat_instance = parse(argv)
    kurodoko_instance = reduction(sat_instance)
    encoding = encode(kurodoko_instance)
    print encoding # a la 'Range' in Simon Tatham's puzzle collection

def parse(argv):
    sets = []
    for s in argv:
        v = map(int, s.split('.'))
        if len(v) > 3:
            raise SystemExit("bad arg: %s (bigger than 3)" % s)
        sets.append(v)
    return sets

def reduction(clauses):
    counts = count_vars(clauses)
    vars = sorted(counts.keys())
    sorting_network = make_sorting_network(vars, counts, clauses)
    (w, h) = compute_gadget_count(counts, sorting_network)
    ww, hh = (gadget_size - 1) * w + 1, (gadget_size - 1) * h + 1
    grid = [[null for c in range(ww)] for r in range(hh)]

    add_variables(grid, w, h, vars, counts)
    add_sorting_network(grid, w, h, counts, sorting_network)
    add_clauses(grid, w, h, clauses)
    print_grid(grid)
    fix_deferred_work(grid)

    return grid

def print_grid(grid):
    for line in grid:
        buf = []
        for x in line:
            if x != None: buf.append(str(x))
            else: buf.append(' ')
        print ''.join(buf)

####################

def add_variables(grid, w, h, vars, counts):
    k = max(counts.values())

    c = 0
    for vidx in range(len(vars)):
        n = counts[vars[vidx]]
        r = h - k

        for i in range(n - 1):
            solder(grid, w, h, r+i, c, tee_l)
            solder(grid, w, h, r+i, c+1, wire_right)
            for j in range(1, i+1):
                solder(grid, w, h, r+i, c+2*j+0, wire_right)
                solder(grid, w, h, r+i, c+2*j+1, wire_right)
            solder(grid, w, h, r+i, c+2*i+2, bend_lu)
            for j in range(i+2, n):
                solder(grid, w, h, r+i, c+2*j, wire_up)
        solder(grid, w, h, r + (n-1), c, variable)

        c += 2*n
    assert c == w + 1

def add_sorting_network(grid, w, h, counts, sorting_network):
    base = h - 1 - max(counts.values())
    def swap(r, c):
        br, bc = base - 2*r, 2*c
        solder(grid, w, h, br - 0, bc + 0, tee_l)
        solder(grid, w, h, br - 0, bc + 1, xor_d)
        solder(grid, w, h, br - 0, bc + 2, tee_r)
        solder(grid, w, h, br - 1, bc + 0, xor_r)
        solder(grid, w, h, br - 1, bc + 1, tee_d)
        solder(grid, w, h, br - 1, bc + 2, xor_l)

    def mkwire(r, c):
        for dr in range(2):
            solder(grid, w, h, base - (2*r + dr), 2*c,
                   wire_up)

    for i in range(len(sorting_network)):
        for j in range(len(sorting_network[i])):
            if sorting_network[i][j]: swap(i, j)
            elif j == 0 or not sorting_network[i][j-1]:
                mkwire(i, j)

def add_clauses(grid, w, h, clauses):
    for i in range(len(clauses)):
        solder(grid, w, h, 0, 6*i + 0, bend_dr)
        solder(grid, w, h, 0, 6*i + 1, wire_right)
        solder(grid, w, h, 0, 6*i + 2, gchoice)
        solder(grid, w, h, 0, 6*i + 3, wire_left)
        solder(grid, w, h, 0, 6*i + 4, bend_dl)

        clause = clauses[i]
        assert len(clause) <= 3
        for j in range(len(clause)):
            v = clause[j]
            if v < 0: solder(grid, w, h, 1, 6*i + 2*j,
                             gwnot)
            else:     solder(grid, w, h, 1, 6*i + 2*j,
                             wire_up)
        for j in range(len(clause), 3):
            solder(grid, w, h, 1, 6*i + 2*j, gzero)

def fix_deferred_work(grid):
    for (r, row) in enumerate(grid):
        for (c, elt) in enumerate(row):
            if elt != '?': continue
            n = 1
            for (dx, dy) in [(-1, 0), (1, 0),
                             (0, -1), (0, 1)]:
                y, x = r + dy, c + dx
                while grid[y][x] != '!':
                    n += 1
                    x += dx
                    y += dy
            grid[r][c] = n
    for (r, row) in enumerate(grid):
        for (c, elt) in enumerate(row):
            if elt in (None, '!'): grid[r][c] = 0
            else: grid[r][c] = int(elt)

####################

def solder(grid, w, h, r, c, gadget):
    w, h = (gadget_size - 1) * w + 1, (gadget_size - 1) * h + 1
    base_y = (gadget_size - 1) * r
    base_x = (gadget_size - 1) * c
    for y in range(gadget_size):
        for x in range(gadget_size):
            gy, gx = base_y + y, base_x + x
            if gy < 0 or gx < 0 or gy >= h or gx >= w: continue
            assert grid[gy][gx] in (null, gadget[y][x])
```

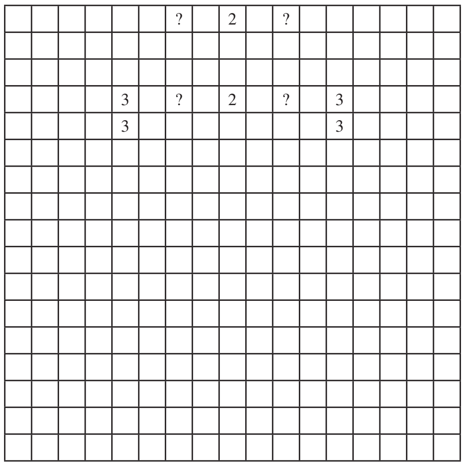
```
            grid[gy][gx] = gadget[y][x]

###################

def count_vars(clauses):
    counts = {}
    for clause in clauses:
        for v in clause:
            counts[v] = 1 + counts.get(v, 0)
    return counts

def make_sorting_network(vars, counts, clauses):
    clauses = [[abs(v) for v in clause] for clause in clauses]
    terminals, goals = sum(clauses, []), dict()
    for (i, v) in enumerate(terminals): goals.setdefault(v, []).append(i)
    wires = []
    for v in vars:
        for j in range(counts[v]):
            wires.append(goals[v][j])
    n = len(wires)

    net = []
    for i in range(n):
        if wires == sorted(wires): break
        row = []
        if i % 2: row.append(0)
        for j in range(i % 2, n - 1, 2):
            if wires[j] > wires[j+1]:
                row.extend([1, 0])
                wires[j], wires[j+1] = wires[j+1], wires[j]
            else: row.extend([0, 0])
        if i % 2 != n % 2: row.append(0)
        net.append(row)
    return net

def compute_gadget_count(counts, sorting_network):
    vals = counts.values()
    n, k = sum(vals), max(vals)
    sorting_network = 2 * len(sorting_network)
    variable_branching = k
    clauses = 2
    return (2*n - 1, variable_branching + sorting_network + clauses)

def encode(instance):
    stream = sum(instance, [])
    buf = []
    runlength = 0
    for elt in stream:
        if runlength == 26 or elt != 0 and runlength > 0:
            buf.append(chr(ord('a') - 1 + runlength))
            runlength = 0
            if elt != 0: buf.append(str(elt))
        elif elt == 0: runlength += 1
        else: buf.append('_%d' % elt)
    return ''.join(buf)

if __name__ == '__main__': main(argv[1:])
```

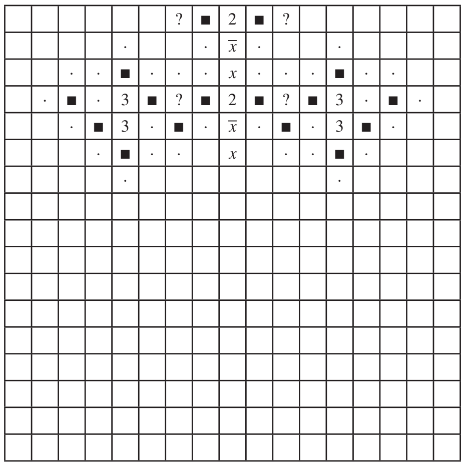
A.2 The Gadgets

Here we display all the gadgets (except Wire, which is shown in Fig. 1 and Fig. 2), along with color deductions that are true in every solution. As earlier, ■, ·, w are black, white, white; all xs are equal, and all $\overline{x}$s are unequal to the xs (easily seen by rule 4).

A.2.1 The Variable Gadget

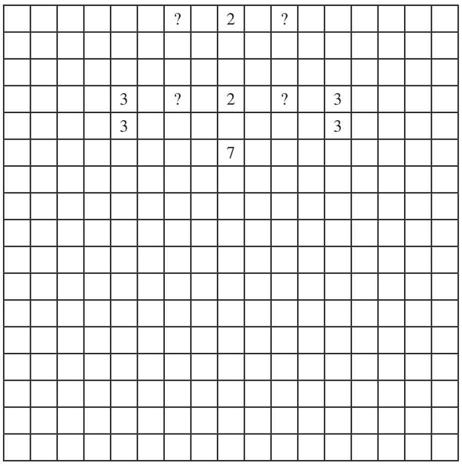
The deductions about black and white squares might be easiest to see if one looks at squares horizontally adjacent to a ? and considers rule 4—it is typically violated if such a square is white.

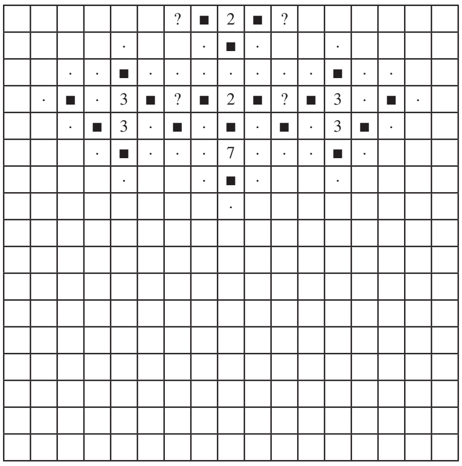




A.2.2 The Zero Gadget

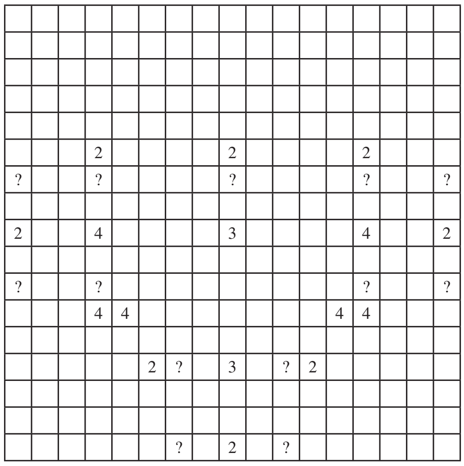
Note the similarity with Variable. We simply force x to be white by putting a clue in its square. As a learning exercise, the reader is encouraged to attempt designing a One gadget.

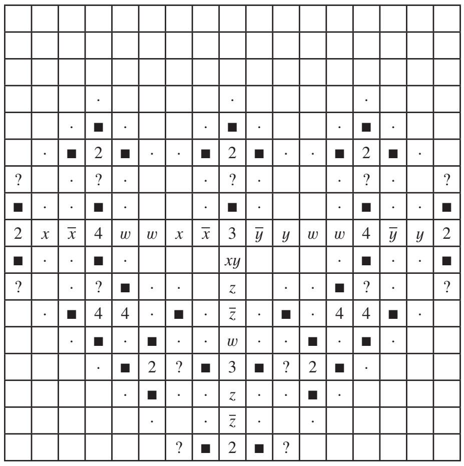




A.2.3 The Xor Gadget

We present the Xor gadget rotated 180° (X^2). Making deductions around the 2s from rule 4 often enables deductions around the 4s.

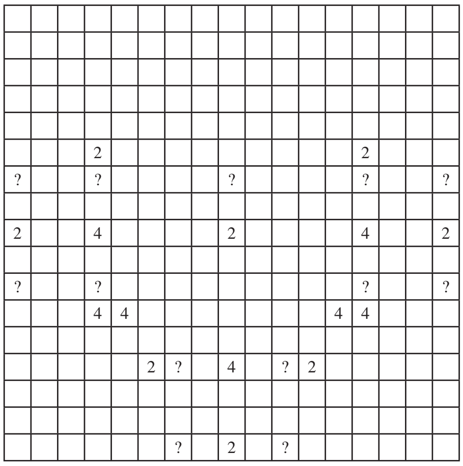


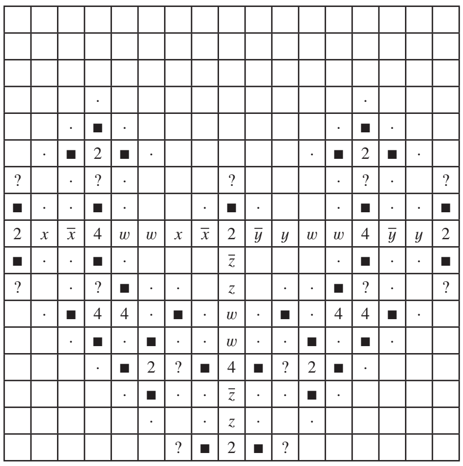


Like $x, \overline{x}$, squares marked $y, \overline{y}$ and $z, \overline{z}$ form consistent opposite pairs. The square marked xy is black if and only if x and y are both black. By rule 4 and by considering the four cases of blackness of $\overline{x}$ and $\overline{y}$ adjacent to the central 3, one can see that $z = x \oplus y$.

A.2.4 The Choice Gadget

Note the similarity with the Xor gadget.

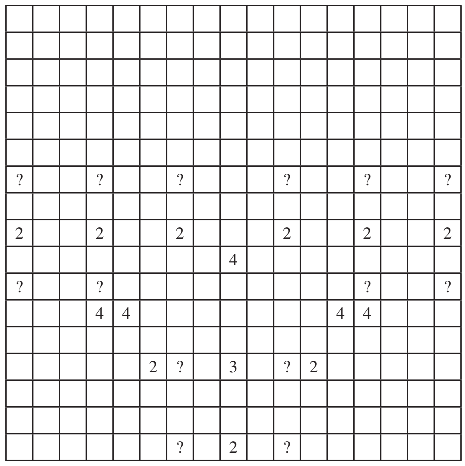


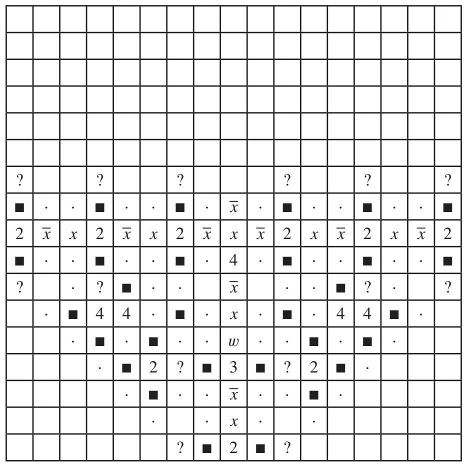


Consider the center clue 2. If (say) x is black then $\overline{x}$ is white. By rule 4 $\overline{y}$ and $\overline{z}$ must be black, and so y and z must be white. This is symmetric, so at most one is black. They can't all be white, or rule 3 is violated as the center clue is trapped.

A.2.5 The Split Gadget

Note the similarity with N^+ in the upper half, and with Xor and Choice in the lower half.

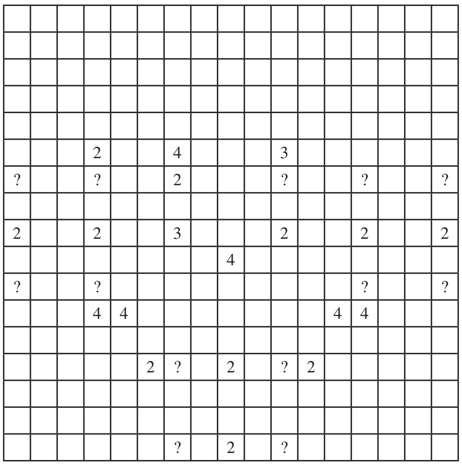


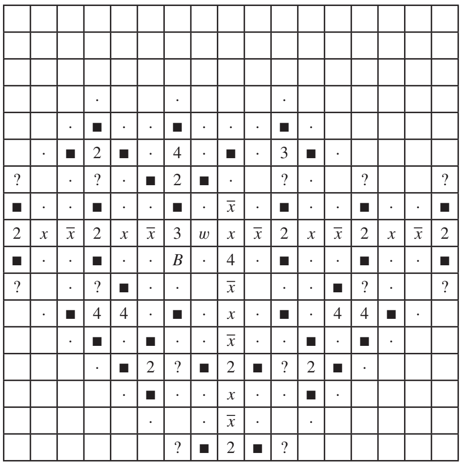


Note that the 3 is flanked by two black squares. If the 2s weren't adjacent to the ?s, connecting the 3 to a ? would not violate rule 4, so the 2s can't immediately be dispensed with.

A.2.6 The Sidesplit Gadget

We present the Sidesplit gadget rotated 90° clockwise (S^+), to make comparison with the Split gadget easier. Note in particular how the 3 has moved.

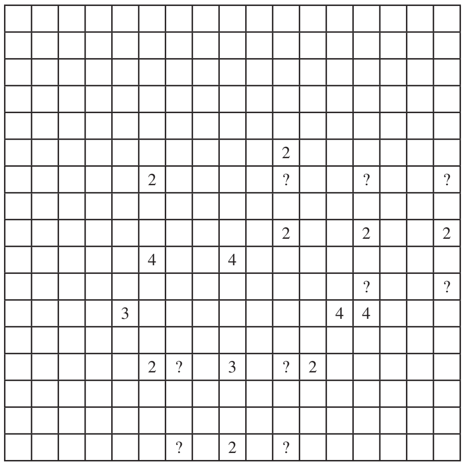


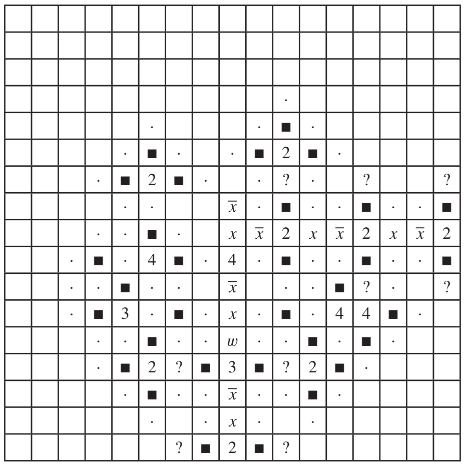


Note that the whiteness of the square below B can be derived independent of B's color. If B is white, then its adjacent 3 has black horizontal neighbours and B's horizontal neighbours are white. This connects the central 4 to more than four white squares, violating rule 4. Thus, B is black.

A.2.7 The Bend Gadget

Note the similarity to the Split gadget in the lower and right hand parts.





A.2.8 The Negation Gadget

						?		2		?						
						?		2		?						
						?		2		?						
						?		2		?						
						?		2		?						
						?		2		?						

						?	■	2	■	?						
							·	x	·							
							·	$\bar{x}$	·							
						?	■	2	■	?						
							·	x	·							
							·	$\bar{x}$	·							
						?	■	2	■	?						
							·	x	·							
								$\bar{x}$								
							·	x	·							
						?	■	2	■	?						
							·	$\bar{x}$	·							
							·	x	·							
						?	■	2	■	?						
							·	$\bar{x}$	·							
							·	x	·							
						?	■	2	■	?						

Jonas Kölker is a Ph.D. student at Aarhus University and a spare time free software developer.