Minesweeper is difficult indeed!

Technology scaling for minesweeper circuits

Alex Thieme[1] and Twan Basten[1,2,⋆[0000−0002−2274−7274]]

[1] Eindhoven University of Technology, Eindhoven, Netherlands
[2] ESI (TNO), Eindhoven, Netherlands

Abstract. Various aspects of playing minesweeper have been proven to be (co-)NP-complete through reductions from circuit-SAT and UNSAT. The proofs use quite involved minesweeper templates to simulate Boolean formulas and circuits. We provide a set of much simpler synthesis templates, leading to much smaller circuit simulations in minesweeper.

Keywords: Games · Complexity · Circuit simulation.

1 Introduction

Minesweeper was and is a popular single-player computer game first released by Microsoft in Windows 3.1 [14]. As for many games, the question was raised whether or not the game is efficiently solvable, i.e., whether or not there is a polynomial-time algorithm to compute a winning strategy. For many games, this is actually not the case (assuming P $\neq$ NP). Minesweeper is among the games for which it turns out that there is no efficient solution strategy.

Minesweeper consistency is the decision problem asking the question whether or not a partial minesweeper configuration can be completed to a valid minesweeper instance. Kaye [6] introduced minesweeper consistency and showed its NP-completeness through reduction from circuit-SAT. circuit-SAT is the problem to decide whether or not a one-output Boolean circuit is satisfiable, i.e., whether or not a Boolean valuation of the circuit's inputs exists such that the output becomes 1. Kaye's reduction uses a set of minesweeper templates, corresponding to gates and wiring elements, to simulate Boolean circuits in minesweeper.

Scott et al. [8] argue that repeatedly checking minesweeper consistency is not the only possible way of playing minesweeper, implying that Kaye's reasoning does not show NP-completeness of playing minesweeper. They define the minesweeper inference problem, asking the question whether or not for a given partial minesweeper instance that is known to be consistent, there is a covered square that is derivably safe or unsafe. Scott et al. use templates similar to those of Kaye to show that minesweeper inference is co-NP-complete, by reduction from Boolean unsatisfiability, UNSAT. UNSAT is the problem to decide whether a Boolean formula built from and, or, and not operators and variables

⋆ Corresponding author, a.a.basten@tue.nl

cannot be satisfied. UNSAT is the complement of Boolean satisfiability, SAT, the problem whether or not a Boolean formula is satisfiable.

All the circuit-simulation minesweeper templates proposed by Kaye and Scott et al. use safe squares as boundaries to create and isolate information flows. We provide circuit-synthesis templates that use mines to create information flows. Mines form a strict isolating boundary so that no information leaks to neighboring squares. This leads to more compact and easier to understand templates and circuits. For instance, a 2-input circuit with three literals and two or gates that fits within a 57x21 minesweeper instance with the templates and synthesis approach of this paper, takes 204x84 squares with the templates of Scott et al.[3]. The key ideas underlying the minesweeper circuit-simulation templates and circuit-synthesis approach developed in this paper are the following:

- Mines are used to create the information flow in Boolean circuits and to isolate these flows from their environment.
- Inversion (logical negation) is done with a 3x3 kernel that also serves as a building block for wires. Consequently, explicit not gates are not needed for circuit synthesis; inversion can be done in the circuit wiring as appropriate.
- A simple 3x3 kernel suffices to create four logic states; this kernel serves as a basis for compact gate templates and wire crossings.

We illustrate how the provided templates can be used to prove NP-completeness of minesweeper consistency, by reducing SAT to minesweeper consistency. This re-establishes the result of [6], but in contrast to Kaye, we consider the original version of minesweeper in which the number of hidden mines is given. This leads to some extra constraints on the templates and the synthesis to ensure that the number of mines in the minesweeper instance generated from a SAT instance is predetermined and independent of the valuation of the Boolean variables in the SAT instance. These constraints are similar to those introduced by Scott et al. for the templates used for proving co-NP-completeness of minesweeper inference. Also Scott et al. consider the original minesweeper version with a given number of mines. Our templates and synthesis approach can replace the templates and synthesis of Scott et al. in their co-NP-completeness proof.

The paper is organized as follows. The next section discusses relevant related work. Section 3 introduces some notations. Section 4 precisely defines the minesweeper consistency and inference problems. In Section 5, we provide templates for circuit simulation in minesweeper. Section 6 elaborates templates for synthesis. These templates satisfy design principles that facilitate automated synthesis and that ensure that the number of mines in the generated minesweeper instance is predetermined. Section 7 provides the synthesis approach and proves NP-completeness of minesweeper consistency. Finally, Section 8 concludes.

[3] The exact dimensions of such a circuit with the templates of [8] depends on the number of inversions needed. With our templates the dimensions only depend on the numbers of variables, literals, and (or-)gate layers.

2 Related work

Information on the history of minesweeper and the rules of the game can be found on Wikipedia [14]. The book by Garey and Johnson [4] is the classical text on NP-completeness. Boolean satisfiability, SAT, is the original decision problem shown NP-complete by Cook [2]. Brief introductions to NP-completeness and proving NP-completeness through reduction can be found in, for instance, [8] and on Wikipedia [16], which also contains a list of NP-complete games and puzzles [15]. Ref. [8] also provides a nice tutorial-style introduction to co-NP-completeness and its relation to NP-completeness. The complexity of games is a popular and fun topic of study. The work of Demaine and colleagues [5,1], for example, provides systematic frameworks for analyzing the complexity of games, at the same time proving complexity results for a wide selection of well-known games, including many of the classic Nintendo games.

Kaye [6] introduced minesweeper consistency and showed its NP-completeness through reduction from circuit-SAT. Kaye's circuit templates are quite involved. An and gate, for instance, has 23x13 squares and a wire crossing is built from and and not gates (24 in total). He later published some further, simplified templates [7], but they remain quite large. Kaye's reduction from circuit-SAT to minesweeper does not guarantee a predefined number of hidden mines. We re-establish the NP-completeness of minesweeper consistency by reduction from SAT for the original version of minesweeper with a given number of hidden mines. This illustrates that this extra piece of information does not fundamentally simplify minesweeper.

The latter was also already observed by Scott et al. [8]. Scott et al. argue that Kaye's reasoning does not prove NP-completeness of playing minesweeper. Kaye assumed that minesweeper is played by iteratively solving minesweeper consistency. Scott et al. observe that there may be other strategies to play minesweeper. They therefore introduce the minesweeper inference problem, which precisely captures the essence of minesweeper game play. They show that minesweeper inference is co-NP-complete (and hence that playing minesweeper is most likely not NP-complete). They do so by reducing UNSAT to minesweeper inference, for the original version of minesweeper with a given number of hidden mines.

The templates of Scott et al. are similar to those of Kaye, but they are designed for synthesizing circuits for the original minesweeper game with a given number of hidden mines. The dimensions of all templates are multiples of three, the number of mines in each template is always the same, independent of the valuation of inputs of the circuit element being simulated, and all the predefined mines are derivable. An or gate, for instance, consists of 24x18 squares with exactly 58 mines of which 43 are predefined and derivable; a crossing consists of 15x9 squares with 21 mines, of which 14 predefined and derivable. Our gate templates are also designed for synthesis and ensure that the number of mines in a synthesized circuit is known. But our templates use mines to create and isolate information flows, where Scott et al. use safe squares to create flows. The use of mines to create flows leads to substantially smaller templates than the templates of Scott et al.. Our or gate has 7x9 squares with 32 mines, of which

29 predetermined and derivable; our crossing has 6x6 squares with 19 mines, 16 derivable and 3 depending on the valuation.

The synthesis approach of Scott et al. is based on rectangular tiles. Our synthesis is based on layers, corresponding to the levels in a tree representation of the Boolean formula being synthesized. Moreover, we integrate negation in the wiring. With the already mentioned choice to use mines to create information flows, these aspects lead to substantially smaller minesweeper circuits than those of Scott et al..

3 Notations

This section introduces some notations needed in the remainder.

First, we define some notations for natural numbers and grids. Let $\mathbb{N}$ denote the natural numbers and $\mathbb{N}_0$ the natural numbers extended with 0. For any natural numbers $k, l \in \mathbb{N}$, let $[k] = \{n \in \mathbb{N}_0 \mid n < k\}$ and $[k,l] = [k] \times [l]$; the neighborhood $N : [k,l] \to 2^{[k,l]}$ is defined for any $(i,j) \in [k,l]$ as $N(i,j) = \{(i+p, j+q) \in [k,l] \setminus \{(i,j)\} \mid p,q \in \{-1,0,1\}\}$.

Second, we introduce some notations for Boolean formulas and circuits. Let $\mathbb{B} = \{0,1\}$ be the set of Boolean values; let V be a set of variables. A Boolean formula f is an expression built from variables from V, the (infix) binary operators $\cdot$ (and, often left implicit in formulas) and $+$ (or), the (postfix) unary operator $'$ (not), and parentheses. A (Boolean) valuation is a function $b : V \to \mathbb{B}$ that assigns a Boolean value to all variables. Boolean formula f is satisfiable if and only if a valuation b of its variables $x_0, \ldots, x_{n-1} \in V$ (for some $n \in \mathbb{N}$) exists so that the formula evaluates to true, i.e., $f(b(x_0), \ldots, b(x_{n-1})) = 1$. Boolean circuits generalize Boolean formulas by allowing shared subformulas and multiple outputs. We omit a precise definition, because our reasoning is based on the subset of Boolean circuits that correspond to Boolean formulas. The Boolean operators are also referred to as (logic) gates in the context of Boolean circuits.

4 Minesweeper consistency and inference

The notations introduced provide a basis for defining both minesweeper consistency and minesweeper inference. We use a $\star$ to denote mines.

Definition 1 (The minesweeper consistency problem [6][4]). *Assume given a $k \times l$ grid, for $k,l \in \mathbb{N}$ and a number of hidden mines $M \in [kl+1]$. A consistent minesweeper instance is a function $m : [k,l] \to [9] \cup \{\star\}$ such that $M = |\{(i,j) \in [k,l] \mid m(i,j) = \star\}|$ and, for all $(i,j) \in [k,l]$, $m(i,j) = \star$ or $m(i,j) = |\{(p,q) \in N(i,j) \mid m(p,q) = \star\}|$. The minesweeper consistency problem then is the question whether a partial minesweeper solution, given in the form*

[4] Kaye does not include the number of still hidden mines in his problem definition; in line with the original minesweeper game, we include this information, following Scott et al. [8].

of a partial function $mp : [k,l] \hookrightarrow [9] \cup \{\star\}$ *and a number of mines* $\#m \in \mathbb{N}_0$ *hidden in the covered squares* $[k,l] \setminus \mathrm{dom}(mp)$*, can be extended to a total function* $m : [k,l] \to [9] \cup \{\star\}$ *with* $|\{(i,j) \in [k,l] \setminus \mathrm{dom}(mp) \mid m(i,j) = \star\}| = \#m$ *that is a consistent minesweeper instance. If so, that partial minesweeper solution is said to be consistent.*

Definition 2 (The minesweeper inference problem [8]). *Assume given a partial minesweeper solution* $mp : [k,l] \hookrightarrow [9] \cup \{\star\}$ *derived from a consistent minesweeper instance* $m : [k,l] \to [9] \cup \{\star\}$ *such that* $mp(i,j) = m(i,j)$ *for all* $(i,j) \in \mathrm{dom}(mp)$*, with the number of still hidden mines* $\#m = |\{(i,j) \in [k,l] \setminus \mathrm{dom}(mp) \mid m(i,j) = \star\}|$*. The minesweeper inference problem is then the question whether there is a covered grid square* $(i,j) \in [k,l] \setminus \mathrm{dom}(mp)$ *for which it can be inferred from* mp *and* $\#m$ *whether* $m(i,j) = \star$ *(i.e., the square is unsafe and contains a mine) or* $m(i,j) \in [9]$ *(i.e., the square is safe).*

Kaye [6] showed that minesweeper consistency is NP-complete, although he did so for the minesweeper version in which the number of hidden mines is not given. In Section 7, we prove NP-completeness of the above version of minesweeper consistency. The fact that also this version of minesweeper consistency is NP-complete, despite the extra available information, was already observed by Scott et al. in [8] and can be proven using the minesweeper circuit-simulation templates provided in that paper. The main contribution of Scott et al. is that they show co-NP-completeness of minesweeper inference. The reason to re-establish NP-completeness of the above version of minesweeper consistency in this paper is to illustrate the use of the provided minesweeper templates in a well defined circuit-synthesis approach.

5 Simulating circuits in minesweeper

The essence of the complexity proofs for minesweeper consistency and inference is the observation that it is possible to simulate circuits in minesweeper. Fig. 1[5] shows minesweeper templates for the three Boolean operators defined earlier. The designs are based on the following principles:

1. Mines are used to isolate gates from their environment. This is essential for the compactness of their design.
2. A mine is interpreted as a logic 1 and a value in [9], i.e., a safe square – no mine, as a logic 0. This is opposite to the interpretation of Kaye and Scott et al.. Because of duality, the interpretation of the Boolean constants in terms of mines and safe squares is not essential though, and it could be swapped.
3. The leftmost template in Fig. 1 shows a not gate. At its core is a triplet of two covered squares and one safe square. The triplet is bordered by mines. This pattern provides logical negation and it returns frequently (in adapted forms) in the other templates to be presented.

[5] All minesweeper figures have been made using the logigames minesweeper solver, https://www.logigames.com/minesweeper/solver.

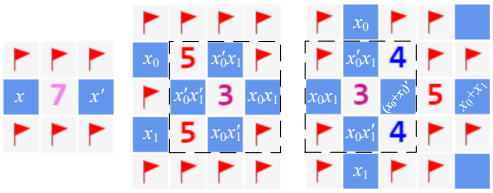

Fig. 1. Boolean operators/logic gates in minesweeper. Predefined mines are denoted by red flags, safe squares by grey squares numbered with the number of neighboring mines (omitting 0s, not seen in this figure), and covered squares by blue squares.

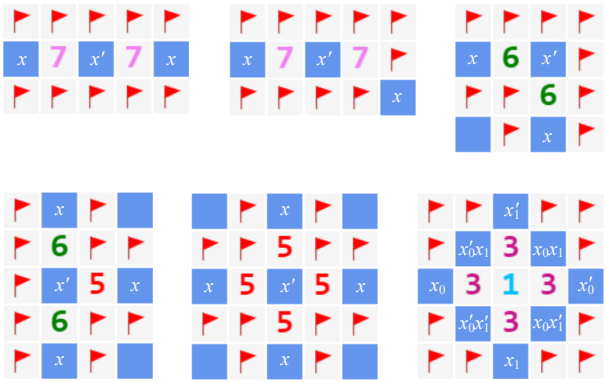

Fig. 2. Various wiring elements

4. The two rightmost templates in Fig. 1 show an and and an or gate. At their core is a 3x3 kernel with an uncovered 3 at its center, two mines and two safe squares in the corners, and four covered squares. This design ensures that the four covered squares can contain only a single mine. By appropriately connecting inputs and outputs to this core, four different states can be represented, allowing to code any two-variable Boolean function. Inputs and outputs follow the not pattern. For instance, the $(x_0, 5, x_0'x_1)$ and $(x_0, 5, x_0'x_1')$ input patterns in the and gate and the $((x_0+x_1)'(= x_0'x_1'), 5, x_0+x_1)$ output pattern of the or gate are instances of the not pattern.

The templates in Fig. 1 are annotated with possible consistent valuations of the (relevant) covered squares. The correctness of the annotations is easily verified against the minesweeper rules (see [14]). The annotations confirm that the templates simulate not, and, and or gates, respectively. The 3x3 not gate and the 4x5 and gate are much smaller than the 13x4 and 23x13 not and and gates given originally by Kaye in [6]. Kaye did not provide an or gate in [6].

To create circuits, we need wires to connect gates. Fig. 2 shows a variety of wiring elements. The top three elements show wires, the two leftmost elements at the bottom show splits, and the bottom right element shows an inverting crossing. The wire and split elements repeatedly use the not pattern. A wire is simply

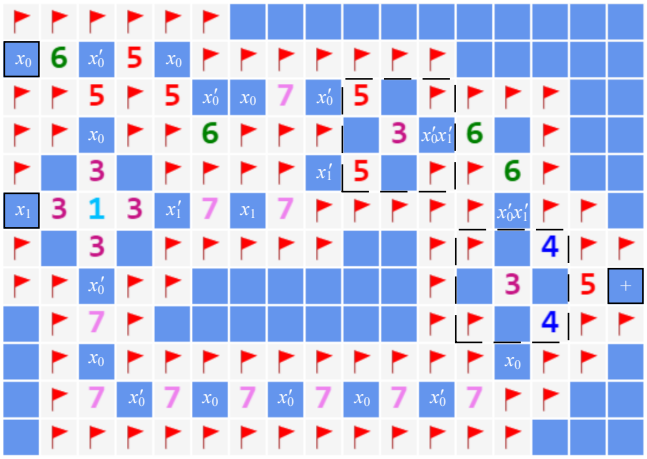

Fig. 3. $x'_0x'_1 + x_0$ in minesweeper

an even number of concatenated not gates; an odd number of concatenated not gates gives an inverting wire/inverter. This can be used to obtain either positive or negative instances of a variable or subformula in circuit construction, as appropriate. The inverting crossing uses a 3x3 kernel very similar to the kernels used for the gates. The crossing can again be combined with not kernels in any appropriate way to obtain positive or negative instances of subformulas.

Fig. 3 shows an example of a circuit simulation in minesweeper, using the gate templates and (variants of the) wiring elements.

6 Templates for circuit synthesis

The templates given in the previous section allow the manual construction of circuits in minesweeper, and they suffice to prove the original NP-completeness result of [6]. They cannot be (easily) used for automated synthesis though, and they are not suitable to prove the earlier mentioned complexity results for the version of minesweeper in which the number of hidden mines is known.

To support synthesis of Boolean circuits and derive the needed templates, we observe that any Boolean formula can be represented as a binary tree. Fig. 4 (left) shows the tree representation of the example circuit of Fig. 3. We use the ideas presented in the previous section and the tree representation of Boolean formulas to develop templates that support synthesis of circuits along the lines of the tree representation. We define four groups of templates, one for gate layers, one for wires, one to create a layer of literals, and one to create wiring layers. But first, we provide a straightforward transformation on Boolean formulas that simplifies the synthesis and set out some design principles for templates.

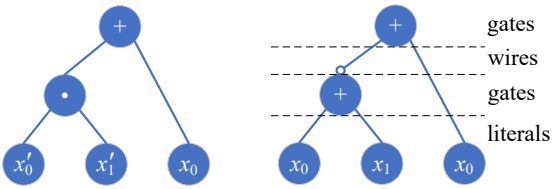

Fig. 4. Tree representations for Boolean formula $x_0'x_1' + x_0$ (left) and the equivalent formula $(x_0 + x_1)' + x_0$ (right)

6.1 Removing and gates from Boolean circuits

Two key laws in Boolean algebra are De Morgan's laws: $x_0x_1 = (x_0' + x_1')'$ and $x_0 + x_1 = (x_0'x_1')'$. The first one shows that and gates can be replaced by or gates if inputs and outputs are inverted. Fig. 4 (right) shows the result of this transformation for the example circuit. Any resulting double negations can be removed. After this transformation, the tree representation of the Boolean circuit consists of layers of (or) gates and wires. Wires may be inverting. The tree is built on leaves of literals. The root may or may not be inverted; it is not for the example circuit because it already had an or gate at the root. The transformation simplifies the synthesis problem, because the resulting representation has only one type of gate. Moreover, negation is a basic building block in the minesweeper emulation of circuits that can be integrated in the wires where needed.

6.2 Design principles for templates

Our synthesis templates satisfy a number of design principles to facilitate synthesis, with a few small exceptions as explained later.

1. The templates are designed per layer in the tree representation of a Boolean formula and have a fixed width to support the layer structure. Trees are laid out from left (leaves, inputs) to right (top, output).
2. The inputs of the simulated circuit element are part of the template; the outputs are not. Templates can be connected by overlapping output and input squares as appropriate.
3. Information flows are isolated by single-file rows of mines, bordered on one side by safe squares. This ensures that all predefined mines in the templates are derivable when the predefined safe squares in the generated minesweeper instance are uncovered.
4. The templates ensure that the content of precisely the covered squares is not derivable.
5. The templates have a given fixed number of mines on the covered squares for all possible assignments of mines to these covered squares. This ensures that the number of mines in a generated minesweeper instance is known.

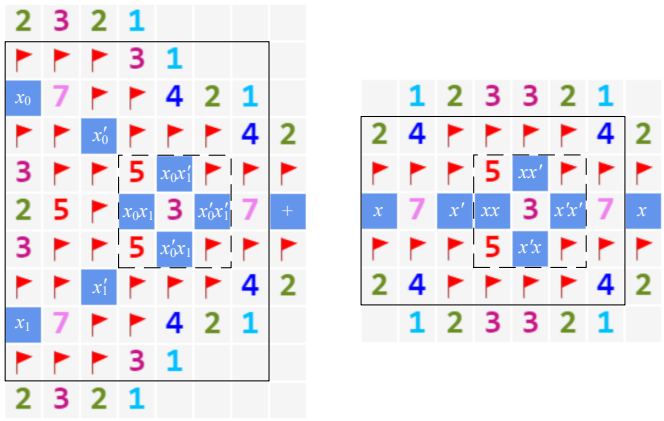

Fig. 5. Gate templates for synthesis

6.3 Templates for gate layers

We create two gate templates to build gate layers from the tree representation of a circuit, one or gate and one wire. The 'wire gate' is needed to cross a gate layer when the tree representing a circuit has no gate at a particular position in that layer. Fig. 5 shows the two gate-layer templates. The templates consist of the squares inside the solid rectangles. They are shown with some additional context to ease understanding.

1. The gate templates have a fixed width of seven squares, meaning that a gate layer is seven squares wide. Both templates use the 3x3 and-gate kernel already presented in the previous section. The or template is built from this 3x3 kernel following De Morgan's law for the or operator. The inputs of the or template are five squares apart, to easily connect to the literal layer, elaborated in the next subsection. The wire template is essentially an and kernel with a single input. The input and output are inverted so that the template has the same width as the or template.
2. The or template has 29 predefined mines; the wire template has 18 predefined mines. As mentioned, these predefined mines form single-file lines bordered on one side by safe squares, which ensures that all these mines are derivable.
3. The or template has precisely 3 mines on the covered squares, one for each pair of covered squares forming the inverted inputs and one in the and kernel. The wire template has precisely 2 mines on the covered squares, one for the inverted input and one for the and kernel. Note that the covered squares marked xx' and $x'x$ are derivably safe, with value 5, deviating from the design principles outlined earlier. But uncovering these two squares does not reveal any extra information about the other covered squares. The template can be redesigned by uncovering these squares or with mines on these squares, resulting in a straight wire. We chose to present this template for gate layers based on the and-gate kernel.

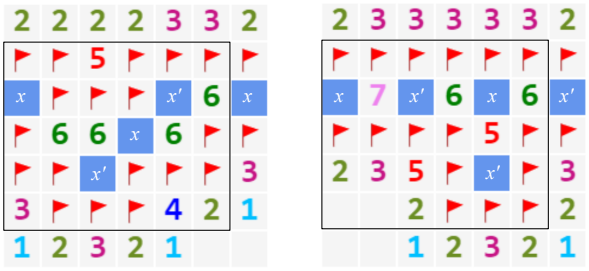

Fig. 6. Wire templates for synthesis: wire (left), inverting wire (right)

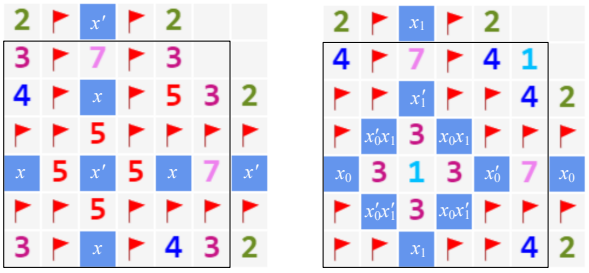

Fig. 7. Templates for the literal layer: split (left), crossing (right)

6.4 Wire templates

Fig. 6 shows two wire templates for synthesis, a wire and an inverting wire. The wire templates are used in connecting the literal layer to the first gate layer and in wiring layers between gate layers, as further explained below.

1. The wire templates have a width of six squares, to support fixed-width layers.
2. The inverting wire has an extra not kernel, the vertical $(x, 5, x')$ triple, to ensure that the number of mines in the template is predetermined, independent of the content of the covered squares.
3. The wire template has 18 predefined mines, the inverting wire 16. Both templates have 2 additional mines on the covered squares, as can be seen from the annotations.

6.5 Templates for the literal layer

Fig. 7 shows two templates for constructing the literal layer for a circuit. These templates are inspired by the split and crossing already presented in Fig. 2.

1. Both templates have a width of six squares. This ensures that all lines of mines in a literal layer are single-file lines, so that the predefined mines in such a layer are derivable.
2. The crossing has 16 predefined mines and 3 additional ones on the covered squares.

Fig. 8. Fixing the mine count in the literal layer

3. The split also has 16 predefined mines, but it either has 1 or 4 mines on the covered squares. This mine count therefore depends on the valuation of x. When used for synthesis, the difference needs to be compensated to ensure that the number of mines in the synthesized circuit is predetermined.
4. Fig. 8 shows two connected splits. The valuations of the covered squares in these two splits are duals. Hence, the combination of the two splits always has 5 covered mines. This is even so when the splits are connected through a (normal, non-inverting) wire. This can be used to create a literal layer with a predetermined number of mines, independent of the valuation of variables.

Fig. 9 shows the layout of a literal layer in minesweeper. A literal layer has a vertical wire for each variable in the Boolean formula for the circuit being synthesized. For each literal in the formula, a horizontal wire is created. The vertical wires create the variables sublayer of the literal layer and are built from crossings stacked on top of each other, with one split at the appropriate place to derive the needed literal. The resulting horizontal wires have connections both to the left and to the right. Fig. 10 shows the construction for the running example.

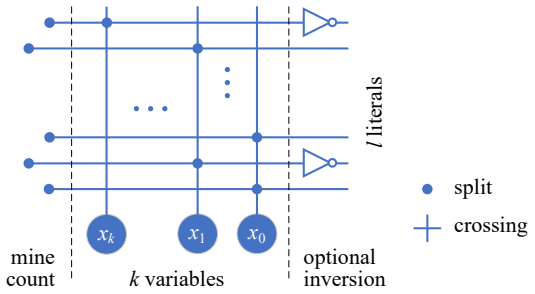

Fig. 9. The layout of the literal layer

The variable sublayer is connected to splits in a mine-count sublayer on its left. Adding one split for every horizontal wire implies that each horizontal wire

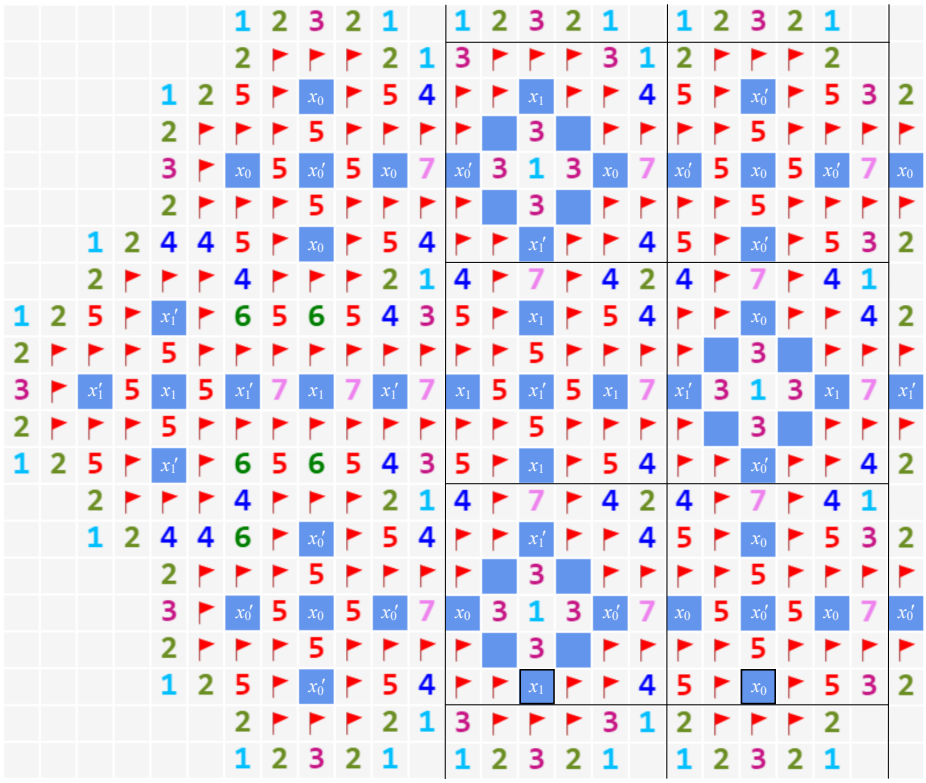

Fig. 10. The literal layer of the example circuit: variables and mine count

has precisely two splits. This ensures a predetermined mine count in this pair, as explained. The additional splits are laid out alternatingly in two vertical stacks to make sure that all predefined mines are in single-file lines (and hence derivable). This alternating layout is achieved by shifting every other split to the left through a simple wire as already shown in Fig. 2, top left. Note that this simple wire can be seen as a 4x3 template with 8 predefined mines and 1 additional covered mine (see also Fig. 12, top left). As a result of the construction up to this point, every horizontal wire in the mine-count and variable sublayers is built from crossings, simple wires, and precisely two splits; see Fig. 10 for the running example. This ensures a predetermined mine count in this part of the literal layer. (The borders seen in Fig. 10 are explained and accounted for in Section 7, that elaborates the reduction from SAT to minesweeper consistency.)

To complete the literal layer, we need to ensure that the literals produced as inputs for the circuit are properly inverted where needed. This can be done by creating a sublayer with the wire templates given already in Fig. 6. Fig. 11 shows the inversion sublayer for the running example. The figure illustrates how it connects the literal layer to the first gate layer of the circuit. Since the wiring

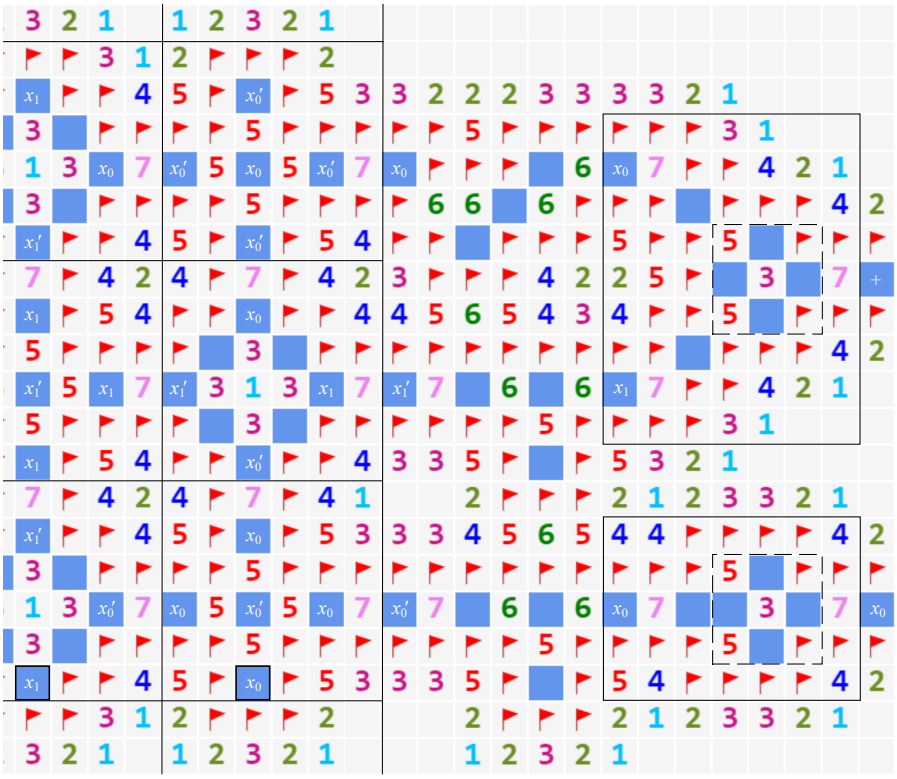

Fig. 11. The literal layer of the example circuit: connection to the first gate layer, optional inversion

templates have a predetermined number of mines, the literal layer as a whole also has a predefined number of mines.

6.6 Templates for connecting layers

One more set of templates is needed for circuit synthesis, namely templates to create wiring layers between gate layers. The layers need to account for vertical displacements, preferably using as little horizontal space as needed (for compactness of the resulting circuits). Two important observations are, first, that the literals coming from the literal layer are vertically 6 squares apart, and, second, that the output of an or gate is 3 squares lower or higher than its inputs. As a result, the vertical displacements that need to be realized in generating a tree-shaped circuit in line with the tree representation of Fig. 4 are all $3 + 6n$ squares, for some $n \in \mathbb{N}_0$. Fig. 12 shows the templates for vertical displacement, including a simple wire template to be used when no displacement is needed. Fig. 13 shows their use in the example circuit.

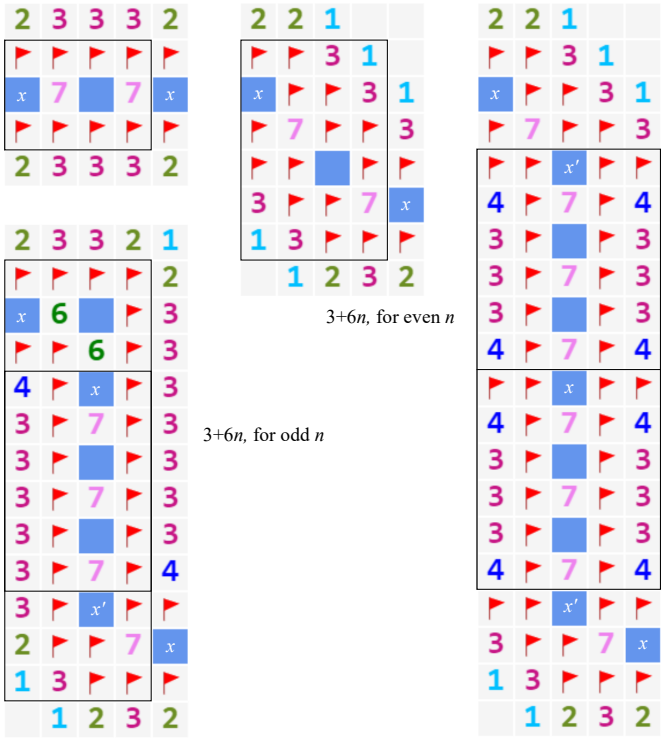

Fig. 12. Wiring elements for vertical displacements of 0 resp. $3 + 6n$ rows ($n \in \mathbb{N}_0$)

1. All templates act as normal wires. As explained, inversion may be needed in the wiring layers. This could be done in the displacement wires. For simplicity, this is not done though. Optional inversion can be achieved using the earlier wire templates also used in connecting the literal layer to the first gate layer of the circuit. Fig. 13 shows that this optional inversion is applied directly to the outputs of the gate layer being connected to the next gate layer (in line with the tree representation in Fig. 4). The displacement templates can then be used to realize the needed displacement. Note that the bottom half of a wiring layer uses templates that are mirrored vertically. This is needed to avoid undesired connections between circuit parts that would prevent derivability of the predefined mines.
2. The wire template without displacement (Fig. 12, top left) is four squares wide, has 8 predefined mines and 1 covered mine; it is used when the input gate layer does not have an or gate but only a wire, as in the bottom part of Fig. 13.
3. The templates for non-zero displacements are split in two cases. This is needed to ensure the combination of derivability of predefined mines, a predetermined number of mines in total for each of the templates, and compactness of the templates.

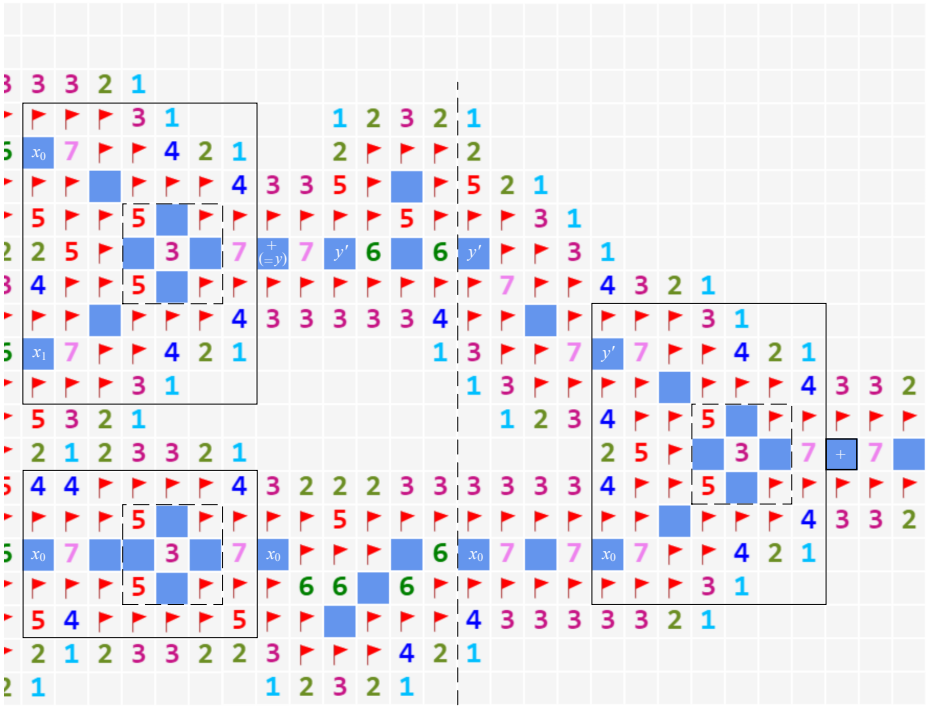

Fig. 13. Synthesized circuit

4. The template for a displacement of $3+6n$ squares, for even $n \in \mathbb{N}_0$, including $n = 0$, consists of a 4x6 kernel template that realizes a displacement of 3 (Fig. 12, top middle) and a 5x6 optional block that can be repeated n times, as needed (Fig. 12, right). It has $14+14n$ predefined mines and $1+3m$, with $m = n/2$, additional mines on its covered squares.
5. The template for $3 + 6n$, for odd $n \in \mathbb{N}$, also consists of a 4x6 kernel and a 4x6 repeatable block that is included n times. It has $14 + 12n$ predefined mines and $3+3m$, with $m = \lfloor n/2 \rfloor$, additional mines on its covered squares.
6. The displacement sublayer of a wiring layer is four squares wide. The repeatable block in the template for even n has a width of five squares, meaning that these blocks extend into the gate layer to the right of the displacement sublayer when used in a circuit. This does not cause any problems because, due to the tree-shaped construction, that is empty space where no other circuit elements appear.

Fig. 13 shows the synthesized circuit for our example. It has two gate layers, in line with the tree representation of Fig. 4. These two layers are connected by a wiring layer, consisting of an inversion sublayer and a displacement sublayer.

Note that the circuit has an inverter connected to its output, despite the fact that this inversion is not needed. The reason for including this not kernel at the

output, also if the circuit does not need it, is that in this way also at the output the number of mines is independent of the Boolean valuation of the output. As a result, the number of mines is predetermined for the entire circuit (shown in its entirety in Figs. 10, 11, and 13).

7 NP-completeness of minesweeper consistency

To illustrate the use of the templates given in the previous section, we prove the NP-completeness of minesweeper consistency by reduction from SAT. An approach to synthesize minesweeper circuits from a Boolean formula, as already sketched in the previous section, is the key ingredient of the reduction.

Definition 3 (SAT). *SAT is the decision problem whether or not a Boolean formula is satisfiable.*

The reduction from SAT to minesweeper consistency consists of two steps. Given a Boolean formula, first, the conversion illustrated in Fig. 4 is applied. Second, a consistent minesweeper instance is generated that simulates the Boolean circuit corresponding to the resulting formula, as follows.

Assume that the circuit to be synthesized has $k \in \mathbb{N}$ variables, $l \in \mathbb{N}$ literals, and $h \in \mathbb{N}$ gate layers. The literal layer determines the height and layout of the minesweeper instance, so we start with the construction of this literal layer.

1. The height of the generated minesweeper instance is determined by the number of literals l. Since the split and crossing templates for the literal layer are 6x6 squares, the height becomes $3 + 6l$ squares, including a bottom border of two squares and a top border of one square.
2. Each variable needs precisely one vertical wire in the variables sublayer. Each wire is constructed from crossings with one split at the appropriate point to split off the horizontal literal wire. The resulting variables sublayer has a width of $6k$ squares.
3. The mine-count sublayer is constructed following the alternating layout of splits shown in Fig. 10. With at least one gate layer, we have at least two literals, meaning that the width of the mine-count sublayer is 12 squares.
4. The inversion sublayer creating the correct literals and connecting the literal layer to the first gate layer is constructed appropriately from wires and inverting wires, leading to a width of 6 squares.
5. The dimensions of the resulting literal layer thus are $(18 + 6k) \times (3 + 6l)$.

Before proceeding with the circuitry of the circuit being synthesized, we compute the mine count in the literal layer.

1. We start with the borders. The bottom border has three mines for each variable. The top border has no mines itself, but the top splits and crossings have one extra mine compared to the standard templates on the position of the topmost safe square marked 7 in those templates. Hence, for each variable, we have 4 mines due to the borders of the generated minesweeper instance, $4k$ mines in total.

2. Next, observe that we have $kl - l$ crossings, with 19 mines each. So the crossings contribute $19l(k-1)$ mines.
3. We also have precisely l split pairs that each have 5 mines on their covered squares. A split in the mine-count sublayer has 23 mines and a split in the variables sublayer has 16 mines. So the splits contribute $44l$ mines.
4. A wire used in the alternating layout of the splits in the mine-count sublayer has 9 mines. We have $\lfloor l/2 \rfloor$ such wires, contributing $9\lfloor l/2 \rfloor$ mines in total.
5. The inversion sub-layer contributes $20l_w + 18l_i$ mines, where l_w is the number of normal wires used and l_i is the number of inverting wires used.
6. The total mine count of the literal layer is $4k + 19l(k-1) + 44l + 9\lfloor l/2 \rfloor + 20l_w + 18l_i = 4k + 19kl + 25l + 9\lfloor l/2 \rfloor + 20l_w + 18l_i$.

The core circuitry of the circuit being synthesized is built from gate and wiring layers as indicated in Fig. 4.

1. The gate and wiring layers can be attached one by one to the literal layer. At the end, an inverter is added and the output of the circuit is chosen appropriately to be either the input or the output of this inverter. All parts of the grid not covered by instances of templates contain only safe squares of which the correct valuation can be derived from the mine assignment for the predefined mines in the templates.
2. Each gate layer is 7 squares wide; each wiring layer is 10 squares wide. The final inverter results in an additional 3 squares width. With h gate layers, this leads to a width of $7h + 10(h-1) + 3 = 17h - 7$ squares.
3. The mine count in the circuitry depends on the layout of the tree representation of the formula being converted. It can be computed by counting the number of used gate, wire, and displacement templates, multiplying these template counts with the appropriate mine counts derived earlier in Section 6, and adding 7 for the final inverter. The two-square bottom and one-square top boundaries that extend from the literal layer to the circuitry part do not contain any mines (besides the ones that are part of a template and hence already accounted for).

Summarizing the above, given a Boolean formula f with k variables, l literals, and h gate layers in its tree representation, the outlined synthesis approach results in a partial minesweeper instance pm_f with dimensions $(11+6k+17h) \times (3+6l)$, a predetermined mine count M_f (that depends on the characteristics of the circuit being synthesized), and a designated output square $(i,j)_f$ for some $(i,j) \in [11+6k+17h, 3+6l]$, with all squares in all the template instances, including the borders in the literal layer, uncovered/flagged as indicated in the templates, and all grid squares outside the scope of the used templates (which are all safe) uncovered as well. The mine count of still hidden mines $\#m_f$ of this partial minesweeper solution is obtained by subtracting the number of flagged squares in the partial solution from M_f. We now have the following result.

Proposition 1. *A given Boolean formula f is satisfiable if and only if the partial minesweeper solution $pm_f \cup \{(i,j)_f \mapsto \star\}$ with mine count $\#m_f - 1$, i.e., the*

generated partial minesweeper solution with a mine allocated to the designated circuit output, is consistent.

The two-step conversion presented up to this point is polynomial in the size of the Boolean formula, because it consists of simple substitutions of nodes and edges in the tree representation of the Boolean formula (by other nodes and edges in the first step and minesweeper templates in the second step).

Finally, it is clear that minesweeper consistency is in NP. Given a minesweeper instance m with mine count M as defined in Definition 1, the consistency conditions given in Definition 1 can be checked in a single traversal of the grid.

Combining Proposition 1 with the observations in the last two paragraphs then gives the following result.

Theorem 1 (Complexity of minesweeper consistency). *Minesweeper consistency is NP-complete.*

8 Conclusions

In this paper, we have given minesweeper templates for simulating and synthesizing circuits. The templates and synthesized circuits are much smaller than the templates and circuit simulations proposed in earlier work. We used the templates to prove NP-completeness of minesweeper consistency, for the original version of minesweeper, in line with Kaye's original result for the version of minesweeper without a given number of hidden mines presented in [6]. Since our templates and synthesis approach predetermine the number of mines in the generated minesweeper instance and because all predefined mines are derivable, the templates can also be used in the proof of Scott et al. in [8], which shows that minesweeper inference is co-NP-complete. Given the small 3x3 kernels that form the basis for our templates, we do not expect that any further (substantial) reduction of the size of minesweeper circuits is possible.

Frits Vaandrager. This paper is part of a Festschrift to celebrate the 60th birthday of Frits Vaandrager. Twan Basten had the pleasure of collaborating with Frits on several occasions. He got to know Frits as a person that looks beyond the purely scientific aspects of his work. Frits is a socially involved person, to whom it is important to not only communicate the societal importance of computer science but also the beauty and the fun of it. Already in 1998, Frits published an opinion piece [10] in which he argued the need to bring logic and theoretical computer science closer to society, through independent academic programs and collaboration with industry. He invested time and effort in developing educational setups (e.g., [3]) and in developing and teaching a model-checking module for high schools [9]. And he actively advocates the importance of theoretical computer science for society at large, as illustrated through this TV fragment on Dutch national TV [12]. Fun is never far away with Frits. When writing a Uppaal tutorial [11], he decided to take a puzzle of gossiping girls as

an illustrative example. And in a column on recreational formal methods [13], he explored the use of SAT solvers to analyze vacuum cleaning scenarios. The current paper intends to be recreational as well. We hope it is a fun read for Frits and other readers alike, and that it as such may contribute to popularizing (theoretical) computer science.

References

1. Aloupis, G., Demaine, E., Guo, A., Viglietta, G.: Classic Nintendo games are (computationally) hard. Theoretical Computer Science **586**, 135–160 (2015). https://doi.org/https://doi.org/10.1016/j.tcs.2015.02.037
2. Cook, S.: The Complexity of Theorem Proving Procedures. In: Proc. 3rd ACM Symposium on Theory of Computing, STOC. pp. 151–158. ACM (1971). https://doi.org/10.1145/800157.805047
3. Fehnker, A., Vaandrager, F., Zhang, W.: Modeling and Verifying a Lego Car Using Hybrid I/O Automata. In: Proc. 3rd Int. Conf. on Quality Software, QSIC 2003. pp. 280–289. IEEE Computer Society (2003)
4. Garey, M., Johnson, D.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W.H. Freeman (1979)
5. Hearn, R., Demaine, E.: Games, Puzzles, and Computation. A K Peters/CRC Press (2009). https://doi.org/10.1201/b10581
6. Kaye, R.: Minesweeper is NP-Complete. Mathematical Intelligencer **22**(2), 9–15 (2000). https://doi.org/10.1007/BF03025367
7. Kaye, R.: Richard Kaye's Minesweeper Pages (2022), https://web.mat.bham.ac.uk/R.W.Kaye/minesw/minesw.htm
8. Scott, A., Stege, U., van Rooij, I.: Minesweeper May Not Be NP-Complete but Is Hard Nonetheless. Mathematical Intelligencer **33**(4), 5–17 (2011). https://doi.org/10.1007/s00283-011-9256-x
9. Vaandrager, F., Jansen, D., Koopmans, E.: Een Module over Model Checking voor het VWO. In: Proc. NIOC 2009. pp. 135–137. Hogeschool Utrecht (2009), in Dutch.
10. Vaandrager, F.: Logica is Prachtig Hulpmiddel bij Oplossen Informaticaproblemen. Automatisering Gids **32**(13), 17 (March 1998), in Dutch.
11. Vaandrager, F.: A First Introduction to Uppaal. In: Quasimodo Handbook. Deliverable D5.12. ICT-FP7-STREP-214755 project Quasimodo (2011)
12. Vaandrager, F.: Alan Turing: Grondlegger van Informatica. In: RTL Late Night. RTL (February 2015), https://www.rtlxl.nl/programma/rtl-late-night/65b22fda-4ef4-bb8d-0c90-80f670646220, TV program (fragment), in Dutch.
13. Vaandrager, F., Verbeek, F.: Recreational Formal Methods: Designing Vacuum Cleaning Trajectories. Bulletin of the EATCS **113**, 100–109 (2014)
14. Wikipedia: Microsoft Minesweeper (2022), https://en.wikipedia.org/wiki/Microsoft_Minesweeper
15. Wikipedia: NP-Complete Games and Puzzles (2022), https://en.wikipedia.org/wiki/List_of_NP-complete_problems#Games_and_puzzles
16. Wikipedia: NP-Completeness (2022), https://en.wikipedia.org/wiki/NP-completeness