Minesweeper May Not Be NP-Complete but Is Hard Nonetheless

ALLAN SCOTT, ULRIKE STEGE, AND IRIS VAN ROOIJ

In volume 22 of *The Mathematical Intelligencer*, Richard Kaye published an article entitled "Minesweeper is NP-Complete." We point out an oversight in Kaye's analysis of this well-known game. As a consequence, his NP-completeness proof does not prove the game to be hard. We present here an improved model of the game, which we use to show that the game is indeed a hard problem; in fact, we show that it is co-NP-*complete*. We explain why our result *does* prove hardness of Minesweeper. We also take the opportunity to discuss the open "NP = co-NP?" question, and explain why NP-completeness of the Minesweeper game under our formulation may not hold, and indeed would be surprising.

Minesweeper[1] is a computer game played on an $(n \times m)$-board of squares. These squares start covered, and some of them contain mines. The goal of the game is to locate all the hidden mines without stepping on any. The number of hidden mines, k, is given to the player. Figure 1 presents an example for $n = m = 4$ and $k = 5$.

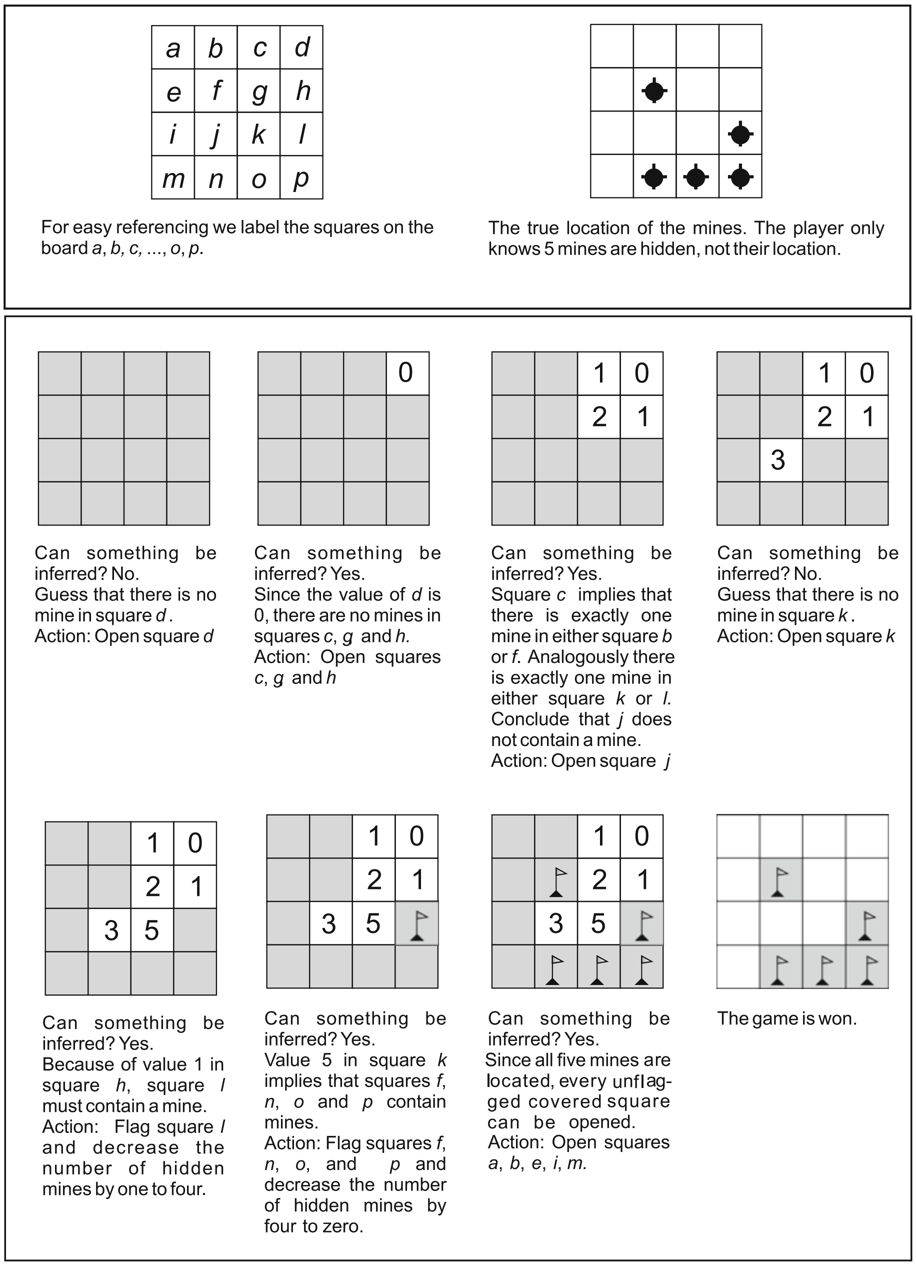
During the game, the player can make moves of two sorts: either flag a square as containing a mine, or open a square. If the square that is opened contains a mine, the player loses the game. If the player opens a square that does not contain a mine, a digit between 0 and 8 is revealed. This digit is the number of adjacent squares that contain mines. Whenever the player flags a square, the number k is reduced by 1. After all the mines are flagged and all the squares without mines are opened, the player wins. (Minesweeper, as offered by Microsoft, also displays a counter for the time played. Because time does not affect the outcome (win or loss) of the game, we ignore this parameter in our analysis.)

How do people play Minesweeper? Different people may have different strategies, but the game has to go roughly as follows: At any stage of the game, the player starts by systematically trying to infer the content of some covered square(s) from the available information. If the player determines that some square must be empty he or she opens it, and if the player determines that some square must contain a mine the player flags it. (The flags serve solely as a memory support for the player. Throughout our analysis we take for granted that the flagging is done, and done correctly.) If at some point the player is unable to infer the content of any square, he or she is forced to guess and to open a covered square thought likely to be safe. If the player is lucky, the square is safe (empty) and the game continues. This process is repeated until either all mines are located (i.e., all squares not containing a mine are opened and the player wins), or the player opens a square containing a mine, in which case the player loses.

In this article, we investigate the complexity of the deterministic part of playing Minesweeper. In our analysis, we assume that the player is given a *consistent* Minesweeper

[1]Minesweeper is a trademark of Microsoft. The game comes as a standard feature of Microsoft Windows. The authors have no relationship with Microsoft, and nothing in this article should be seen as a comment on any of Microsoft's products.

DOI 10.1007/s00283-011-9256-x



For easy referencing we label the squares on the board *a*, *b*, *c*, ..., *o*, *p*.

The true location of the mines. The player only knows 5 mines are hidden, not their location.

Can something be inferred? No.
Guess that there is no mine in square *d*.
Action: Open square *d*

Can something be inferred? Yes.
Since the value of *d* is 0, there are no mines in squares *c*, *g* and *h.*
Action: Open squares *c*, *g* and *h*

Can something be inferred? Yes.
Square *c* implies that there is exactly one mine in either square *b* or *f*. Analogously there is exactly one mine in either square *k* or *l*. Conclude that *j* does not contain a mine.
Action: Open square *j*

Can something be inferred? No.
Guess that there is no mine in square *k*.
Action: Open square *k*

Can something be inferred? Yes.
Because of value 1 in square *h*, square *l* must contain a mine.
Action: Flag square *l* and decrease the number of hidden mines by one to four.

Can something be inferred? Yes.
Value 5 in square *k* implies that squares *f*, *n*, *o* and *p* contain mines.
Action: Flag squares *f*, *n*, *o*, and *p* and decrease the number of hidden mines by four to zero.

Can something be inferred? Yes.
Since all five mines are located, every unflag-ged covered square can be opened.
Action: Open squares *a*, *b*, *e*, *i*, *m*.

The game is won.

Figure 1. Example of a Minesweeper game. The squares of the (4 × 4)-board are labeled for easy referencing (top left). The locations of the hidden mines are shown (top right). At the beginning of the game, the player only knows that $k = 5$ mines are hidden and every square of the board is covered. A possible game sequence (bottom). At each stage, the player solves a version of Minesweeper Inference. If an inference can be made, the appropriate action is taken: if for a covered square the content "no mine" can be inferred, the square is uncovered, and the number of mines adjacent to it is revealed; if, on the other hand, it can be inferred that a covered square contains a mine, the square is flagged, and the mine counter k is decremented. Otherwise, the player makes a guess. Note that here the player is lucky in guessing.

board configuration. That is, when completely revealed, the board presented has in each square either a mine or a number corresponding to the correct number of neighboring mines. Further, we assume an ideal player who makes no mistakes in reasoning. Note that these assumptions imply that the player can only lose (try to open a square containing a mine) when forced to guess. Therefore, our optimal player who follows the strategy described previously will avoid guessing whenever possible. However, to do so she must be able to decide whether it is possible to make progress on the board without guessing. This problem is what we call the *Minesweeper Inference* problem.

Minesweeper Inference

The Minesweeper Inference problem comes in two versions, a *search* version and a *decision* version. We consider first the search version. Its *input* is the information that is available to the player at a particular time during the Minesweeper game. Its *output* is a covered square *s* whose content (i.e., mine or not) can be inferred from the available information if such a covered square exists; otherwise, the problem outputs the statement "The player is forced to guess."

Minesweeper Inference (*search version*)
Input: A consistent board configuration with some revealed digits and some correctly flagged mine locations derivable from the revealed numbers. The number of hidden mines, $k \geq 0$.
Output: A covered square *s* and its content (either "mine" or "no mine"), if there exists a covered and non-flagged square whose content can be inferred from the available information. Otherwise output "The player is forced to guess."

Note that Kaye's article established that it is NP-hard (a notion treated below) to verify that a board is consistent. This means that Minesweeper Inference is what is called a *promise problem*. Readers curious about this technical point are referred to Oded Goldreich's survey "On Promise Problems (a survey in memory of Shimon Even [1935–2004])."

We next introduce the decision version of Minesweeper Inference, as this is the version that we analyze. The decision version has the same input as the search version, but simply asks whether *any* safe covered square exists.

Minesweeper Inference (*decision version*)
Input: A consistent board configuration, with some revealed digits and some correctly flagged mine locations derivable from the revealed numbers. The number of hidden mines, $k \geq 0$.
Question: Does there exist at least one covered and non-flagged square whose content (either "mine" or "no mine") can be inferred from the available information?

Figure 1 illustrates how playing Minesweeper involves solving successive instances of the Minesweeper Inference problem. For each of the shown configurations, the player first solves the decision version, returning either a Yes or No answer. If the answer is Yes, the player subsequently solves the search problem (for one or more squares). If the answer is No, the player makes a guess.

Note that the player naturally assumes that the board she is playing is consistent. Therefore, the assumption of a consistent board configuration in the input of Minesweeper Inference is realistic. The creation of a consistent Minesweeper board configuration is not a hard problem: just select some squares to contain mines, and then for each cell not containing a mine compute the number of neighboring mines. However, given only a board configuration, telling whether it is consistent is indeed hard, as we will soon see.

Minesweeper Consistency

In his article, Kaye presented a different analysis of the Minesweeper game. He considered another decision problem, called *Minesweeper Consistency*.

Minesweeper Consistency
Input: A board configuration, with some digits and flagged mine locations.
Question: Does there exist a placement of mines that is consistent with the visible digits and flagged locations?

Kaye argued that the ability to solve Minesweeper Consistency yields a way of solving what we call the Minesweeper Inference problem. Let us describe his argument. (We do not keep track of *k*, the number of hidden mines, just because Kaye's article did not.) Assume we have a method *M* for solving the Minesweeper Consistency problem. Then we can use method *M* to also solve Minesweeper Inference as follows:

1. We pick an arbitrary covered square, call it *s*.
2. We use *M* to solve Minesweeper Consistency for the configuration C_m, where C_m represents the present configuration with the change that *s* is defined to contain a mine.
   (a) If the answer is No for Minesweeper Consistency, then we know there is no mine in *s*, and we output for Minesweeper Inference square *s* with inferred content "no mine."
   (b) If the answer is Yes for Minesweeper Consistency, then we use *M* to solve Minesweeper Consistency for each of the nine configurations $C_0, C_1, C_2, \ldots, C_8$, where C_i represents the present configuration with the change that *s* has as content the number "*i*," representing the number of mines adjacent to *s*.
   - If the answer is No for Minesweeper Consistency for each such C_i, then we know there is a mine in *s*,

AUTHORS

ALLAN SCOTT received his Ph.D. at the University of Victoria, and continues as a post-doctoral fellow at the same institution. His research focuses on computational complexity, especially fixed-parameter tractabi-lity and games.

Department of Computer Science
University of Victoria
Victoria, BC, V8W 3P6
Canada
e-mail: aescott@gmail.com

ULRIKE STEGE, after obtaining her doctorate in computer science at the ETH in Zurich, came to the University of Victoria as a post-doctoral fellow. She joined the faculty there in 2001 and is now an associate professor.

Department of Computer Science
University of Victoria
Victoria, BC, V8W 3P6
Canada
e-mail: stege@cs.uvic.ca

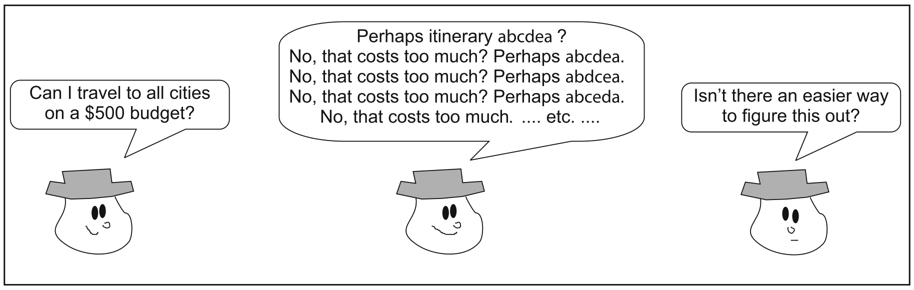

Can I travel to all cities on a $500 budget?
Perhaps itinerary abcdea ? No, that costs too much? Perhaps abcdea. No, that costs too much? Perhaps abdcea. No, that costs too much? Perhaps abceda. No, that costs too much. .... etc. ....
Isn't there an easier way to figure this out?

Figure 2. No simple (polynomial time) procedure for solving TSP is known.

and we output square *s* with inferred content "mine" for Minesweeper Inference.
- If the answer is Yes for Minesweeper Consistency for at least one of the C_i, then we cannot conclude anything for square *s*, and we repeat Steps 1-2 for a different covered square.

3. If we run out of squares to consider, then we output that nothing can be inferred from the available information for Minesweeper Inference (i.e., "The player is forced to guess.").

We note that, although this strategy proposed by Kaye provides one possible way of solving Minesweeper Inference, it is not necessarily the only strategy for doing so.

Is Minesweeper NP-Complete?

In his article, Kaye proved that the Minesweeper Consistency problem is NP-*complete*. Deferring for a bit the definition and discussion of NP-completeness, we note just that it means that Minesweeper Consistency is among the hardest problems in the class NP. From this, Kaye inferred—or so the title of his article suggests—that playing the Minesweeper game is hard, too. This is the oversight we mentioned in our introduction. Minesweeper Consistency being hard does not imply that Minesweeper Inference is hard. Although it is possible to use the Minesweeper Consistency problem to solve Minesweeper Inference, some other strategies for solving it might be more efficient. We conclude that the complexity of Minesweeper Inference, and thus also the complexity of the game, is still an open question!

In this article we present a new result that puts this open question in an altogether different light. In particular, we prove that the Minesweeper Inference problem is *co*-NP-complete. In the next section we will explain in detail what this means, but anticipating a little, we mention some consequences. First, our result proves what Kaye's proof did not, viz., that the Minesweeper Inference problem (and arguably, the game) is as hard as the hardest problems in NP. But second, it also proves that the Minesweeper Inference problem cannot even be a member of the class NP—and *a fortiori* cannot be NP-complete—unless a famous, widely believed conjecture is false.

P, NP, co-NP, and Completeness

To understand our result, one needs to know about the open question "NP = co-NP?" that is related to the famous million-dollar "P = NP?" question, for which see http://www.claymath.org/prizeproblems/pvsnp.htm for more details, and Devlin, *The Millennium Problems*, Basic Books, New York, 2002. The starting point is the class co-NP and its relation to P and NP. In this section we walk you through the basics using the Traveling Salesperson problem as a running example. We will return to the Minesweeper game in the following section.

Consider a salesperson who wishes to know if there exists an itinerary visiting every one of *n* cities such that the total cost of travel is within budget. More precisely, the salesperson wants to solve the following problem:

Traveling Salesperson problem (TSP)
Input: A set of cities. For each pair of cities, *a* and *b*, there is a cost associated with travel from *a* to *b*. Further, there is a budget constraint *B*.
Question: Does there exist an itinerary visiting all cities such that the total incurred cost does not exceed *B*?

Although this may be very hard to figure out (Figure 2), it is easier to double check that a suggested itinerary is indeed within budget (Figure 3). The TSP problem is a favorite example of such discrepancy.

IRIS VAN ROOIJ, after studying cognitive psychology at Radboud University Nijmegen, came to the University of Victoria for her doctorate. After three years as a post-doctorate at Eindhoven, she returned to Nijmegen, where she is an assistant professor of Artificial Intelligence as well as working at the Donders Institute.

Donders Institute for Brain, Cognition, and Behavior
Radboud University Nijmegen
6500 HB Nijmegen
The Netherlands
e-mail: i.vanrooij@donders.ru.nl

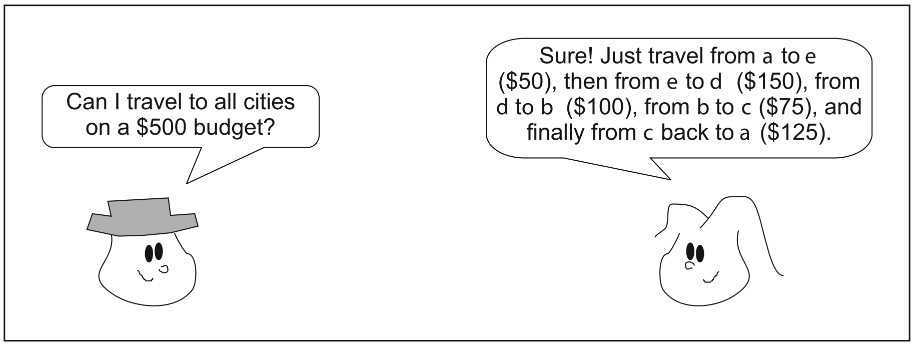

Can I travel to all cities on a $500 budget?
Sure! Just travel from a to e ($50), then from e to d ($150), from d to b ($100), from b to c ($75), and finally from c back to a ($125).

Figure 3. Proofs of Yes-answers to TSP can be verified in polynomial time.

In complexity theory, one often asks whether a problem can always be solved in *polynomial time*, that is, whether the number of steps for solution can be bounded by a polynomial in the size of the problem. If finding a solution to a problem can be done in polynomial time, the problem is said to belong to the class P. If verifying a solution (a Yes-answer) can be done in polynomial time, the problem is said to belong to the class NP. In line with intuition, the TSP is easily seen to belong to NP, but, for reasons we will recall in the following text, it seems unlikely that it is in P.

NP and co-NP

Although not as famous as the "P = NP?" question, the "NP = co-NP?" question is of comparable importance to mathematics. To explain what it is about, we again consider TSP.

A travel agent can either return a Yes or a No answer to the salesperson. If the answer is Yes, she can give the salesperson a possible itinerary as proof, which the salesperson can verify in polynomial time (Figure 3). But what if the answer is No? What if there does not exist any itinerary that satisfies the salesperson's budget constraint? To prove correctness of *this* answer to the salesperson, the travel agent could present the cost associated with each possible itinerary, and the salesperson could then verify that none of them is within budget (Figure 4). However, this verification procedure would take a number of steps that is factorial in the number of cities, and thus not polynomial time. Problems for which No answers can be verified in polynomial time are said to belong to class co-NP.

The question of whether co-NP is the same as NP is open and important. Many a problem, including TSP, may appear to require much more work for verifying No answers than Yes answers, yet the possibility of an elusive polynomial-time verification method for No answers may not be excluded with certainty.

Given that we already know TSP is in NP, it is not difficult to come up with a problem in co-NP. We simply change the question as follows: Does there *not* exist an itinerary that is within budget? Or, equivalently and perhaps more intuitive: Do *all* possible itineraries exceed the budget constraint? Let us call this problem the co-Traveling Salesperson problem (co-TSP, here the prefix "co-" stands for *complement*). If the answer is Yes for TSP then it is No for co-TSP, and if it is No for TSP then it is Yes for co-TSP. But Yes answers to TSP can be verified in polynomial time, so the same holds for No answers to co-TSP. Hence, co-TSP is a member of the class co-NP. In general, for any decision problem Q in NP there exists a complement problem co-Q in co-NP, which has the answers Yes and No reversed.

It is unknown whether NP = co-NP. Mathematicians mostly believe that this is not the case; see, for example, Jon Kleinberg & Eva Tardos, *Algorithm Design*, Addison-Wesley, 2005.

Among the problems in NP, some are known to be key, in the sense that they are the "hardest" problems in NP in the sense of computational complexity; they are called NP-complete. Similarly for co-NP-complete. We will be going through the definitions in the next subsection, but let us first orient you by displaying the conclusions schematically.

Figure 5 presents an overview of the possible relationships among the classes P, NP, and co-NP. There are only three possible scenarios. The first possible scenario is the one conjectured by most living mathematicians, viz., that P $\neq$ NP and that NP $\neq$ co-NP. The second possible scenario is that P $\neq$ NP but that NP = co-NP. The last possible scenario is that P = NP, in which case NP = co-NP and thus all three classes collapse.

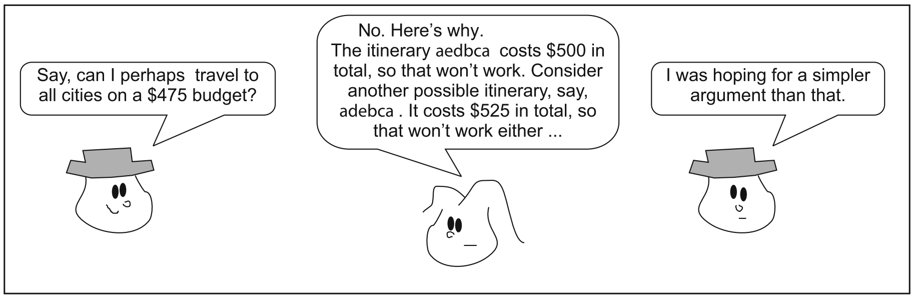

Say, can I perhaps travel to all cities on a $475 budget?
No. Here's why. The itinerary aedbca costs $500 in total, so that won't work. Consider another possible itinerary, say, adebca . It costs $525 in total, so that won't work either ...
I was hoping for a simpler argument than that.

Figure 4. No answers to the traveling salesperson may not have simple (i.e., polynomial-time verifiable) proofs.

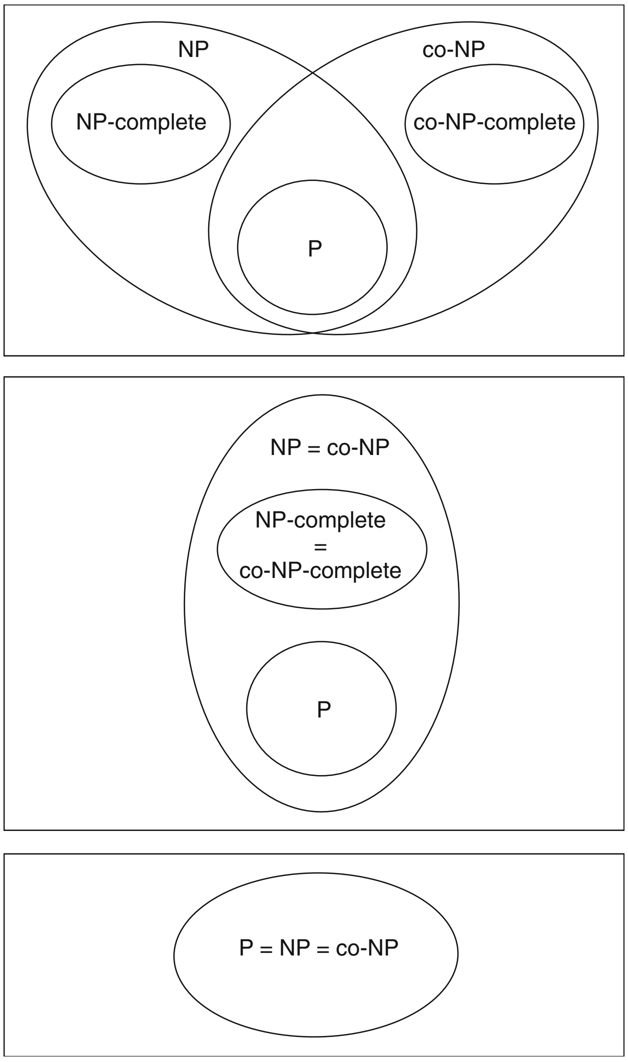


Figure 5. The three possible relationships among P, NP, and co-NP.

Completeness and Reductions

We have used the terms NP-complete and co-NP-complete and mentioned that problems in NP or co-NP that have this property are among the hardest of their class; in particular, they are crucial in trying to prove that P $\neq$ NP.

What exactly does it mean for a decision problem to be NP-complete? A decision problem Q is said to be NP-*complete* if it is (1) in NP and (2) NP-*hard*. To understand what an NP-hard problem is we first discuss *polynomial-time reduction*, a concept on which our main proof below rests.

A *polynomial-time reduction* from a decision problem Q to a decision problem R is a polynomial-time algorithm M for transforming any given input I for Q into an input I' for R in such a way that the answer to I' is a Yes for R if and only if the answer to I is a Yes for Q.

A decision problem Q is said to be NP-*hard* if every problem in NP can be *polynomial-time reduced* to Q. Thus any NP-complete problem, being NP-hard, must be as hard (in this sense) as anything in NP. Finding a polynomial-time solution for any one NP-complete problem would imply, via these polynomial-time reductions, that every other NP problem had a polynomial-time solution too—hence that P = NP. In 1972, R. M. Karp showed that the Hamiltonian cycle problem is NP-complete. This in turn implied NP-completeness of TSP.

Analogously to NP-completeness we define co-NP-completeness: A problem Q is said to be co-NP-*complete* if (1) Q is a member of co-NP and (2) Q is co-NP-*hard*, that is, if every problem in co-NP can be polynomial-time reduced to it. Thus if any one co-NP-complete problem could be solved in polynomial time then all other problems in co-NP would be solvable in polynomial time also (this too would imply P = NP). We further know that, if we can construct a polynomial-time algorithm M that reduces some "new" decision problem R to a known co-NP-complete problem Q, we then can conclude that R is co-NP-hard. Namely, then we can reduce an arbitrary S in co-NP to R in polynomial time as follows: We just reduce S to Q (known possible because Q is co-NP-complete), and then reduce Q to R using method M.

Now suppose that one day we could prove an NP-complete problem, say TSP, to be in co-NP. Then in particular, co-TSP would have a polynomial-time reduction to TSP. Then we could start with any co-NP problem Q, get a polynomial-time reduction of co-Q to TSP by NP-completeness, and this would also be a polynomial-time reduction of Q to co-TSP, yielding in turn such a reduction to TSP: TSP would be proved co-NP-complete, and we would have to be in one of the two bottom cases of Figure 5. This is why we began the paper by expressing doubt that the problem we will prove co-NP-complete is also in NP.

Minesweeper Inference Is co-NP-Complete

In this section we prove that the decision version of Minesweeper Inference is co-NP-complete. We first show that Minesweeper Inference is in co-NP. We then show that it is co-NP-hard by giving a polynomial-time reduction from a problem known to be co-NP-complete. Namely, we rely on the famous *Satisfiability* problem, shown to be NP-complete by Stephen A. Cook, "The Complexity of Theorem-Proving Procedures," 1971. Then our task will be to reduce its complement, called *Unsatisfiability*, to Minesweeper Inference in polynomial time: instances of the Unsatisfiability problem must be transformed into Minesweeper boards.

Minesweeper Inference Is in co-NP

We now show that the Minesweeper Inference problem is a member of the class co-NP. We distinguish between two types of instances of the Minesweeper Inference problem: those for which the answer is Yes (called *Yes-instances*) and those for which the answer is No (called *No-instances*). We invoke two fictional characters, an argument-maker, let's call her Anna, and an argument-verifier, let's call him Vince. To prove that Minesweeper Inference is a member of co-NP, we need to show that for every *No*-instance there exists an argument A that Anna can make such that Vince can verify A in polynomial time; that is, Anna always has an argument A that is easy for Vince to verify. Our proof is as follows.

Imagine Anna is presented with a No-instance of Minesweeper Inference. After searching for some (possibly long!)

time, she comes up with an argument A for why the given instance is indeed a No-instance. The argument has the following form: For each covered square s on the board, Anna presents two *different* consistent board configurations. One of these configurations assigns a mine to s, and the other one assigns to s a value that is *not* a mine. This effectively shows that the content of no covered square s can be inferred from the available information.

Now we show how Vince can verify the correctness of Anna's argument A in polynomial time.

For every covered square s, Vince considers the two possible assignments provided by Anna and makes sure that one of them puts a mine in s and the other one does not. For each of the two possible assignments, Vince considers each square t on the board one by one and checks if its (assigned) content is consistent with the (assigned) content of its 8 neighboring squares (if it is for all t then he knows that the entire board is consistent). Why is this a polynomial-time check? There are at most nm squares on an $n \times m$ board, so this procedure takes on the order of $2 \cdot 8 \cdot nm = 16nm$ steps. Vince repeats the checking procedure nm times, once for each covered square s of the board. Thus, Vince requires at most $16n^2m^2$ steps in total to verify argument A.

A Polynomial-Time Reduction from Unsatisfiability

To show next that Minesweeper Inference is co-NP-hard, we will give a polynomial-time reduction from the known co-NP-complete problem Unsatisfiability, the complement problem of Satisfiability.

Both Satisfiability and Unsatisfiability take as input a Boolean formula in conjunctive normal form. A *Boolean formula* is written using only ANDs, ORs, NOTs, variables, and parentheses. A Boolean formula is in *conjunctive normal form* if it is a conjunction of disjunctions; for example, the formula (u OR v OR w) AND ($\bar{v}$ OR w) AND ($\bar{u}$ OR $\bar{v}$ OR $\bar{x}$ OR y) is a Boolean formula in conjunctive normal form, where $\bar{z}$ indicates the negation of z for any variable z. Each disjunction in the formula is also called a *clause* (e.g., (u OR v OR w) is a clause), and the variables or their negations appearing in the clauses are also called *literals* (e.g., $\bar{v}$ and w are literals). An assignment where each variable is given the value either TRUE or FALSE is called a *truth assignment*. A formula can be *satisfied* by assigning the values TRUE and FALSE to the variables in such a way that every clause is TRUE. Considering our example shown previously, assigning TRUE to u, w, and y, and FALSE to v and x satisfies the formula. We call a formula that can be satisfied *satisfiable*; otherwise, the formula is said to be *unsatisfiable*. An example of an unsatisfiable formula is ($\bar{u}$ OR $\bar{v}$) AND (u OR v) AND ($\bar{u}$ OR v) AND (u OR $\bar{v}$). No matter how we assign the values TRUE and FALSE to the variables u and v this formula is never satisfied.

For a given Boolean formula F in conjunctive normal form, Satisfiability asks, "Is F satisfiable?" Unsatisfiability asks, "Is F unsatisfiable?"

> **Unsatisfiability**
> *Input:* A Boolean formula F in conjunctive normal form.
> *Question:* Is F unsatisfiable?

We now present a polynomial-time algorithm that reduces Unsatisfiability to Minesweeper Inference. This reduction takes place in two steps. First, we transform the input formula F into a formula F' that is more suited to our purposes while remaining unsatisfiable if and only if F is unsatisfiable. Then we show how to implement formula F' as a Minesweeper board $B_{F'}$ for which we can infer the content of a covered square if and only if F' is unsatisfiable. After the description of the reduction we prove its correctness, showing that a formula F is a Yes-instance for Unsatisfiability if and only if $B_{F'}$ is a Yes-instance for Minesweeper Inference.

Transforming Formula F to Formula F'

This transformation consists of three steps.

First, we remove all clauses that are *tautologies*, that is, clauses that are always TRUE no matter the truth assignment. Since a clause is a disjunction of literals, it can only be a tautology if it contains both the affirmation of a variable v and its negation $\bar{v}$. Such clauses are satisfied in *all* possible truth assignments and can safely be assumed absent; indeed, a formula F for which some clauses are tautologies is unsatisfiable if and only if formula F^*, that is F with the tautologies removed, is unsatisfiable.

Second, we add one extra clause to the tautology-free formula F^*, consisting of a single new variable that does not occur in F. The purpose of this extra clause will become apparent only later, but note that it does not change the unsatisfiability of the formula: If F^* is unsatisfiable, then $F^{**} = (v)$ AND F^* is also unsatisfiable. Conversely, any truth assignment that satisfies F and therefore F^* also satisfies F^{**} when we additionally set v to TRUE.

Third, we build F' from F^{**} in replacing every AND in F^{**} by its logical equivalent of ORs and NOTs according to DeMorgan's law: for Boolean formulas F_1 and F_2, formula F_1 AND F_2 is satisfiable if and only if formula $\overline{\overline{F_1} \text{ OR } \overline{F_2}}$ is satisfiable.

We remark that all steps of the transformation from F to F' can be done in polynomial time. Further, F is satisfiable if and only if F' is satisfiable.

Constructing Board $B_{F'}$ from Formula F'

The manner in which we construct Minesweeper board $B_{F'}$ from F' utilizes the fact that every Boolean formula can be encoded by a Boolean circuit, resulting in circuit $C_{F'}$ (Figure 6) that is satisfiable if and only if formula F' is satisfiable. Specifically, $C_{F'}$ can be constructed from F' as follows: We first feed the extra clause and the first clause of F into an or-gate. Then we take the output of that gate and feed it into another or-gate, together with the second clause. Then that result is fed into another or-gate with the third clause, and so on, until we have included all the clauses.

We must now give the prescription for building a Minesweeper board $B_{F'}$ corresponding to a $C_{F'}$. We partition $C_{F'}$ into *units* (e.g., splitters, crossovers, units of wires, joints, or-gates, and not-gates), and then construct for each unit a *board tile*, that is a partial Minesweeper board. The tiles are put together such that the resulting Minesweeper board $B_{F'}$ exactly encodes $C_{F'}$ that in turn encodes formula F' (Figure 7). Our construction then will ensure that $C_{F'}$ is unsatisfiable if

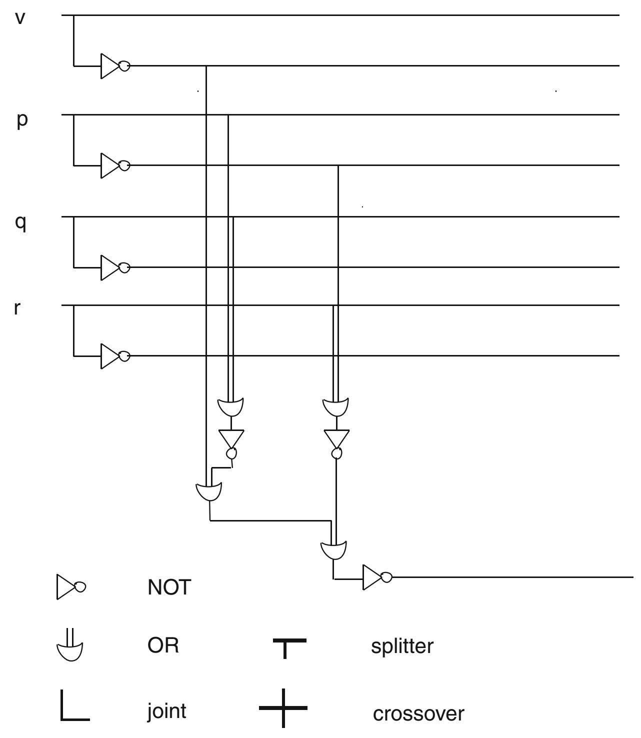


Figure 6. A Boolean circuit representation of formula F'. Here, $F' = (v)$ AND F with $F = (p \text{ OR } q)$ AND $(r \text{ OR } \bar{p})$. Equivalently, without the use of ANDs, $F' = (\overline{v})$ OR $(\overline{p \text{ OR } q})$ OR $(\overline{r \text{ OR } \bar{p}})$.

and only if for the constructed board a mine can be inferred. Proof of correctness of our construction is left for last.

Our proof construction is similar to Kaye's reduction from Satisfiability to Minesweeper Consistency: Like him, we construct a Minesweeper board that models a circuit representing Boolean formula. There are, however, important differences. First, our reduction is from Unsatisfiability instead of Satisfiability, which makes our proof a co-NP-hardness reduction instead of an NP-hardness reduction. Second, we reduce to Minesweeper Inference instead of Minesweeper Consistency, which makes our co-NP-hardness result relevant to the hardness of the Minesweeper game. Third, in line with the game Minesweeper, the number k of hidden mines is given along with the other information on the constructed Minesweeper board. In Kaye's model of the game this parameter was ignored.

The general setup of the Minesweeper board construction is shown in Figure 8. Analogous to the setup of the Boolean circuit, one can think of the information for each variable being sent via wires (composed of wire-tiles) and other board tiles from the input terminals on the left border of the Minesweeper board to the output terminal at the right-bottom end of the Minesweeper board. Each input variable of a circuit is transformed into a (input) terminal tile. For each such variable, a wire is transmitting its value and a wire transmitting the negation of this value is attached underneath (cf. Figure 9).

Subsequently, each clause is constructed using tiles for or-gates, not-gates, splitters, crossovers, wire-units, and joints (cf. Figure 10). The clauses are then "conjuncted" using, again, the tiles for or-gates, not-gates, crossovers, splitters, and joints. When the complete board $B_{F'}$ encoding a formula F' has been constructed, the number k of hidden mines is computed by adding up the number of hidden mines per gadget used in our construction.

The final step in constructing our board is to take the conjunction of all the clauses according to the layout of $C_{F'}$.

Representation of the Board Tiles

In the Minesweeper board whose general layout has been described, a piece of the board will correspond to each of the many tiles. How do we construct the different tiles that go into the board? We first list general properties of the tiles, each being a partial Minesweeper board consisting of covered and uncovered squares, and then we describe the construction of tiles of the various types.

1. For each tile at least one covered square is defined as an *input* square, and at least one covered square is defined as

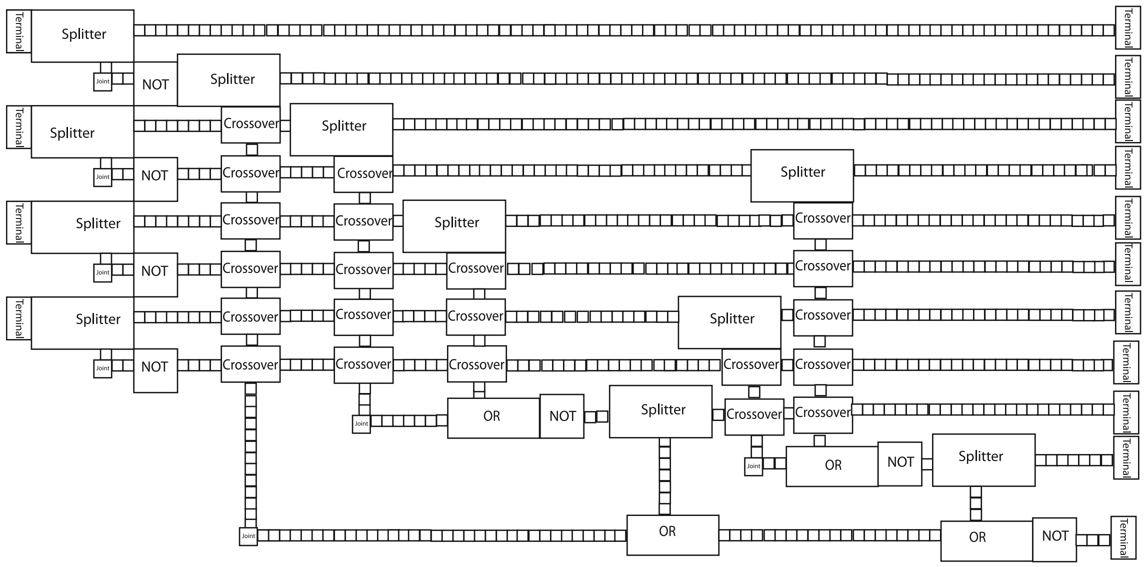


Figure 7. A sketch of the Minesweeper board for formula F' from Figure 6.

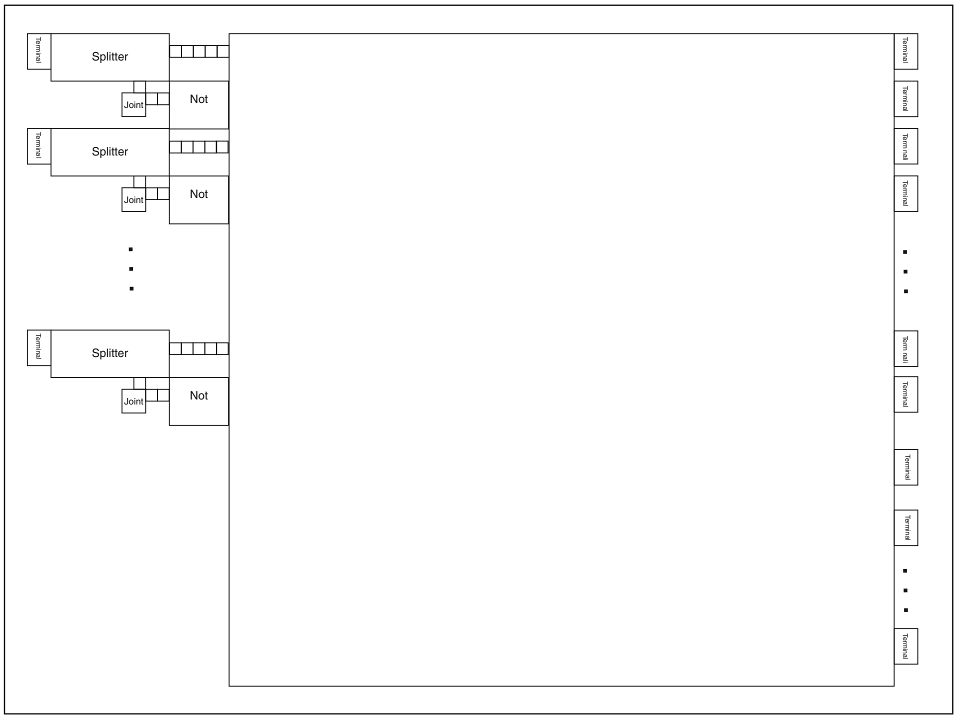


Figure 8. General Setup.

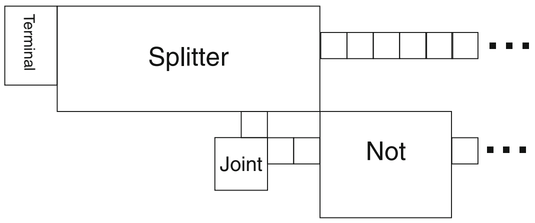


Figure 9. Wires for a variable and its negation.

an *output* square (with the exception of the tiles representing inputs or outputs, called "terminals," which have either only exactly one output square or only exactly one input square).

2. Each tile is constructed such that its input square(s) and output square(s) align with the output square(s) of the directly preceding and the input square(s) of the directly following tiles.
3. Each tile is constructed such that its input square contains a mine if and only if the corresponding output square of the preceding tile does not contain a mine.
4. Each tile is constructed such that it is impossible to derive the content of any of its covered squares without using knowledge from other tiles.
5. Each tile is constructed such that, if all its numbers are revealed and the input/output values of its neighboring tiles are unknown, then exactly the mines that are indicated by solid squares in the neighboring tiles are derivable.
6. Every tile is constructed such that every possible assignment of mines that is consistent with the tile's information uses the same number of mines (this provision is needed to ensure a well-defined number of hidden mines, *k*, for the constructed Minesweeper board).
7. Each tile is constructed such that its width and length is divisible by three (to make it easier to plan the board layout).

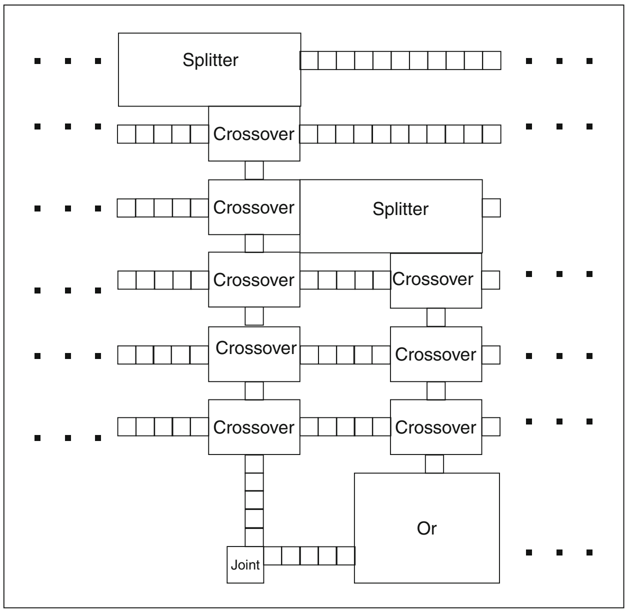


Figure 10. Connecting two variables using an OR-gate; the information is sent vertically down, using one splitter and crossovers.

We distinguish between three classes of tiles: *terminals*, *connectors*, and *logical operators*. Figures 11–16, are

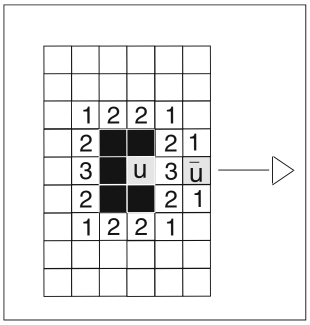


Figure 11. An input terminal. Output terminals are constructed symmetrically. Grey squares denote covered squares, white squares denote squares with content "0." For clarity, flagged squares are represented by black squares instead of flags.

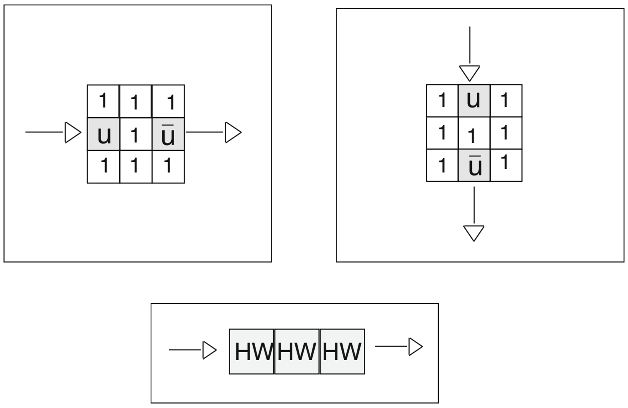


Figure 12. Top left: a horizontal wire-tile (HW). Top right: a vertical wire-tile (VW). Bottom left: A horizontal wire built out of three horizontal wire-tiles.

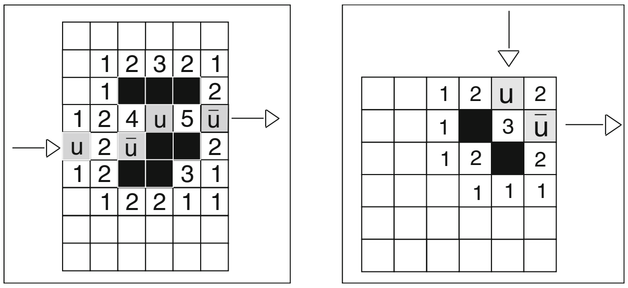


Figure 13. Left: An Up-shifter moves horizontal wire tiles upwards by one row. Down-shifters, left-shifters, and right-shifters are constructed accordingly. Right: A joint connects a vertical and a horizontal wire-tile.

illustrations of tiles in these classes. In these figures, covered tile squares are labeled to make it easier for the reader to assess correctness of the constructed tiles. Two squares s_1 and s_2 are labeled u when: s_1 contains a mine if and only if s_2 contains a mine. Further, two squares s_1 and s_2 are labeled u and $\bar{u}$, respectively, when: s_1 does not contain a mine if and only if s_2 contains a mine. Below we discuss the properties of each different type of tile.

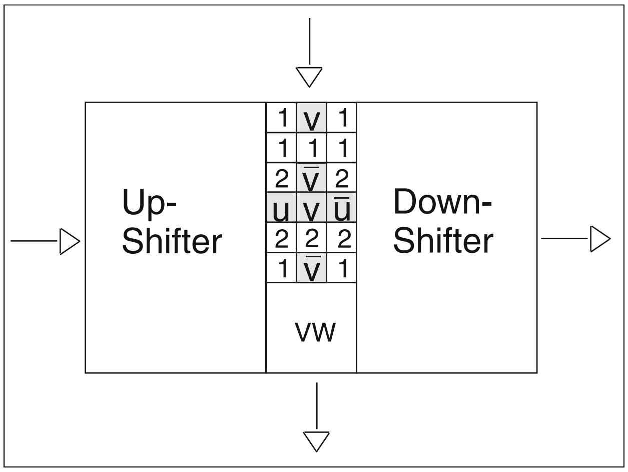
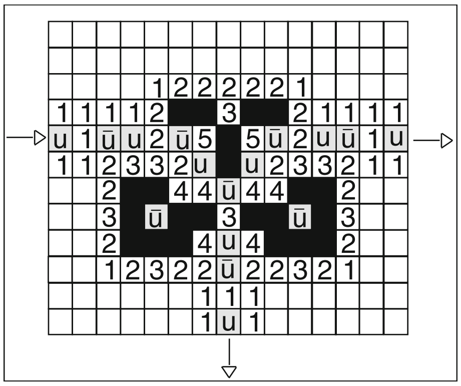


Figure 14. A crossover transmits information from left to right and from top to bottom.



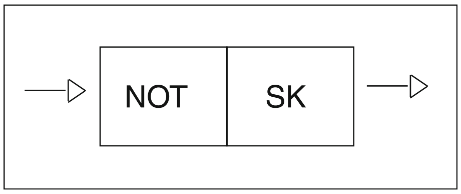
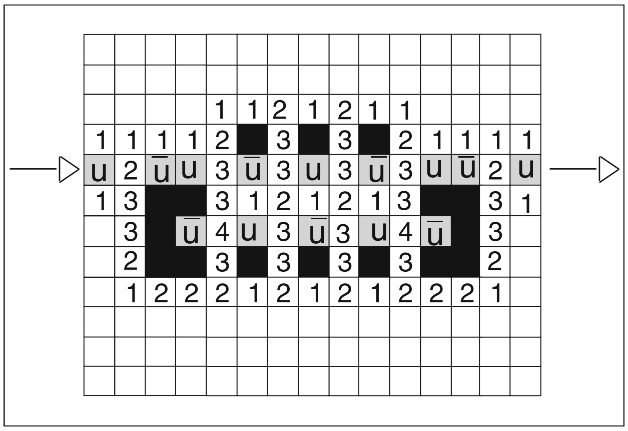


Figure 15. Realization of a splitter with one input and two outputs. The input of the splitter-kernel (SK, top) is inverted.

1. Terminals

Input terminals: An *input terminal* is a tile of size 6 × 9 squares containing exactly 6 mines. Five of them are derivable solely by the digits (derivable mines are depicted by black squares), one needs as additional information the input value of the neighboring tile. The right covered square (labeled with $\bar{u}$) is the output square.

For each variable u in formula F', we place on the left-end side of the board an input terminal. If there is a mine assignment for the Minesweeper board where the output square of an input terminal does not contain a mine, we interpret this as corresponding to a truth assignment of formula F' where variable u is set to TRUE, otherwise as u is set to FALSE.



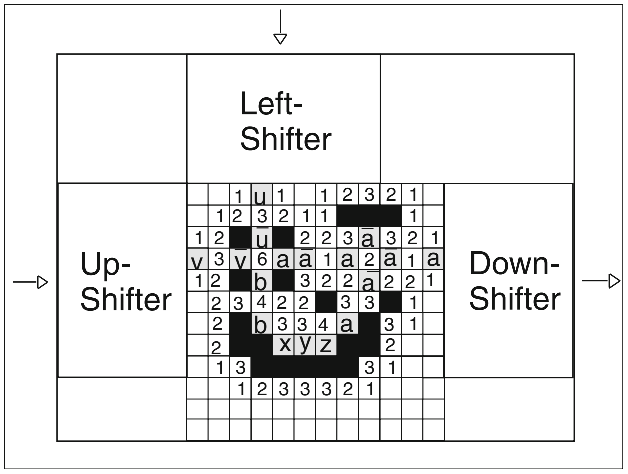


Figure 16. Logical Operators: A NOT-tile (top) and an OR-tile (bottom).

Output terminal: Exactly one *output terminal* is placed on the right-bottom end of the constructed Minesweeper board. The output terminal is the mirror image of an input terminal (reflected across the vertical axis), with the left covered square being its input square.

If there is a mine assignment where the input square of the output terminal contains a mine, then there is a truth assignment that satisfies formula F'.

2. Connectors

Wire tiles: We simulate the wires of our Boolean circuit with *wire tiles*. A wire tile is of size 3×3 containing exactly one mine. Its location depends on its neighboring tiles. Wire tiles come in two kinds: those that transmit information from left to right and from top to bottom. The left/top covered square in a wire tile is its input square (labeled u), the right/bottom covered square is its output square (labeled $\bar{u}$).

By construction, the output square of a wire tile contains a mine if and only if the input square does not contain a mine; thus, the labeling is consistent with the truth values. In other words, if the input square of a wire tile contains a mine then this can be interpreted as $u = \text{TRUE}$ being transmitted to the next tile. If, on the other hand, the input square does not contain a mine then this can be interpreted as $u = \text{FALSE}$ being transmitted. Two or more wire tiles that are placed in sequence are also called a *wire*.

Shifters: We sometimes need *shifters* to ensure that input or output squares are properly aligned with neighboring tiles. Shifters come in two kinds, a horizontal shifter is a tile of size 6×9 and a vertical shifter is a tile of size 9×6. A shifter contains exactly 9 mines, 7 of which are derivable solely by the tile's digits. Horizontal shifters can be either up-shifters or down-shifters, and vertical shifters can be either left-shifters or right-shifters.

Joints: We use *joints* to simulate "bent" wires in the circuit. A joint is a tile of size 6×6 containing exactly 3 mines, two of which are derivable solely by the tile's digits.

The output square of a joint (either its top, right, or bottom covered square) contains a mine if and only if the input square (left, bottom, or top covered square) does not contain a mine.

Crossovers: To allow wires to cross each other we use *crossovers*. A crossover is a tile of size 15×9 containing exactly 21 mines, 14 of which are derivable solely by the tile's digits. A crossover has two input squares—the top-most and left-most covered squares—and two output squares—the bottom-most and right-most.

Output square $\bar{u}$ of a crossover tile contains a mine if and only if input square u does not contain a mine, and the other output square $\bar{v}$ contains a mine if and only if its corresponding input square v does not contain a mine.

Splitters: A *splitter* takes one input and produces two output lines with the same value. A splitter is formed from a *splitter kernel* and a NOT-tile (detailed below) because, as we will see, the kernel's outputs are actually the negation of its input. Thus, we use the NOT-tile to negate the input to the splitter kernel. A splitter kernel is a tile of size 15×12 containing 30 mines, 22 of which are derivable solely by the digits of the splitter kernel. It has one input square (the left covered square) and two output squares (the right-most and bottom-most covered squares).

The splitter kernel is constructed such that its output squares contain mines if and only if its input square also contains a mine.

3. Logical Operators

NOT-tile: We use *NOT-tiles* to simulate not-gates. This tile is of size 15×12 and contains exactly 23 mines, 16 of which are derivable solely from the tile's digits. The left-most covered square is the input square and the right-most covered square is the output square.

By construction, the output square of a NOT-tile contains a mine if and only if the input square also contains a mine.

This means that, compared to wire tiles, a NOT-tile flips the truth value fed into it.

OR-tiles: To simulate or-gates, we use *OR-tiles.* An OR-tile is a tile of size 24×18 containing exactly 58 mines, 43 of which are derivable by the tile's digits. An OR-tile has two inputs: (1) the top-most covered square (labeled by u) and (2) the left-most covered square (labeled v). Its output square is the right-most covered square (labeled $\bar{a}$).

By construction, the output square of an OR-tile does not contain a mine (corresponding to the truth value $a = (u$ OR $v) =$ TRUE being transmitted) if and only if neither of its input squares contains a mine (corresponding to u or v being TRUE).

A sketch of the Minesweeper board constructed from the described tiles to encode a Boolean formula is given in Figure 7. Correctness of the tiles for terminals, connectors, and NOT-tile and the OR-tile can be verified by checking that the labeling of the tiles is consistent with the information in the uncovered squares (i.e., whether squares with label u indeed all either contain a mine or all do not contain a mine, and that all squares labeled $\bar{u}$ contain a mine in the latter case, but not in the former case) and checking that these tiles posses the general tile properties (listed in the previous subsection "Representation of Board Tiles"), when properly aligned.

Of these properties, the hard one to verify is Property 4, namely that it is impossible to derive the content of any of a tile's covered squares without using knowledge from other tiles. For this, recall our previous example with Anna and Vince. If we treat the gadget as a self-contained board, we can show that the covered squares cannot be derived as Anna did—by showing that for every covered square there is at least one possible assignment (of mines to the covered squares) that puts a mine in that square, and another that leaves the square empty. To do this, consider all the possible truth assignments for the wires running through the given gadget. In all the gadgets involving just one variable—which we call u—all the covered squares correspond to u or $\overline{u}$. The squares corresponding to u contain a mine when u is true, and are empty when u is false (vice-versa for $\overline{u}$). Since every covered square corresponds to either u or $\overline{u}$, considering both possible assignments (u is either true or false) immediately shows that for every covered square there is a possible mine assignment that puts a mine in that square (u true puts mines in the squares corresponding to u, u false puts mines in the squares corresponding to $\overline{u}$), and one that leaves that square empty. This gives us Property 4 for single-variable gadgets. The property can be shown for the remaining two-variable gadgets using a similar—though more involved—technique over all four possible input combinations.

Correctness of the Reduction

To prove correctness of our reduction we need to show that formula F is unsatisfiable if and only if board $B_{F'}$ is a Yes-instance for Minesweeper Inference. We know from our previous discussion that formula F is unsatisfiable if and only if formula F' is unsatisfiable if and only if Boolean circuit $C_{F'}$ is unsatisfiable. It remains to show that $C_{F'}$ is unsatisfiable if and only if we can infer the content of at least one covered square of the minesweeper board $B_{F'}$ (viz., for a covered square in the output terminal). (It is not required that that inference be made in polynomial time.) We prove this in two steps: we show (1) $C_{F'}$ is unsatisfiable $\Rightarrow B_{F'}$ is a Yes-instance for Minesweeper Inference, and (2) $C_{F'}$ is satisfiable $\Rightarrow B_{F'}$ is a No-instance for Minesweeper Inference.

1. If $C_{F'}$ is unsatisfiable, then for each possible truth-assignment to the variables inputted into $C_{F'}$—simulated by all possible mine placements in the input terminals of $B_{F'}$—$C_{F'}$ outputs FALSE. This corresponds to a mine placement in the right covered square of $B_{F'}$'s output terminal. But this means that the right covered square in the output terminal of $B_{F'}$ can be inferred to contain a mine.
2. If $C_{F'}$ is satisfiable, then there exists a truth-assignment to the variables—simulated by corresponding mine placements in the input terminal—such that $C_{F'}$ outputs TRUE. This corresponds to a mine placement in the *left* covered square in the output terminal of $B_{F'}$. We know on the other hand that for every formula F' there exists a truth-assignment as input to $C_{F'}$ that results in output FALSE. (This fact comes as the payoff for the sly dodge of giving F' above an extra variable not needed in F.) This corresponds to a mine placement for $B_{F'}$ that has a mine in the right square of the output terminal. This, however, implies that no mine in the output terminal tile can be inferred. Given Property 4 discussed previously and the manner in which the circuit and therefore the board simulates the conjunction of clauses, for no other covered square in the Minesweeper board can the content be inferred.

With this proof we complete the reduction from the Unsatisfiability problem to the Minesweeper Inference problem. To verify that the reduction is a *polynomial-time* reduction, observe that the size of the board is proportional to the number of literals in formula F.

Because Unsatisfiability is known to be a co-NP-complete problem, the presented polynomial-time reduction proves that Minesweeper Inference is co-NP-hard. Taken together with the co-NP-membership proof in the previous section, this proves that Minesweeper Inference is co-NP-complete. This means that solving the Minesweeper Inference problem—which is necessary for playing the Minesweeper game optimally—is as hard as solving any co-NP-complete problem. It also means that Minesweeper Inference itself is not NP-complete unless NP = co-NP.

Conclusion

As noted by Richard Kaye, the Minesweeper game can be used to illustrate fundamental questions in the theory of computational complexity. In this article, we extended Kaye's work and showed how Minesweeper illustrates not only the famous "P = NP?" question, but also the important "NP = co-NP?" question. In the process, we revealed several interesting properties of Minesweeper. To explain these properties, we consider in the following text several scenarios that may arise in playing the game of Minesweeper.

Say you are playing Minesweeper and, at some point, you find yourself stuck. You are unsure if you can proceed in deterministic mode or if you are forced to guess. You could, of course, consider every possible placement of mines consistent with the visible information and verify whether there exists a covered square whose content is fixed across all possible placements. In that case you could infer its content. But such an exhaustive method looks daunting, as there are already billions of possible ways of placing $k = 8$ mines on a 10×10 board. You wonder if you could perhaps devise a faster method to determine if you are forced to guess, say, one that takes at most polynomial time. Our co-NP-completeness result shows that, no matter how hard you try, you will not be able to come up with an efficient (polynomial-time) method for solving your inference problem—or at least that if you do, you will have proved P = NP.

Now, suppose that a friend is looking over your shoulder at the configuration on the screen and claims to know the answer to your inference problem. There are two possibilities. Either he claims you *can* infer the content of another covered square, or he claims that you *cannot*. Of course, in both cases you would want to be convinced. What do you think will be more difficult for your friend, to convince you that you *can* infer something or to convince you that you *cannot*? Again our co-NP-completeness is relevant, this time with (what you might find to be) a counterintuitive twist. It shows that if you *cannot* infer anything, then your friend should easily be able to convince you. That is, he can give you a simple argument that you can verify in polynomial time. However, if you *can* infer something, then your friend will generally *not* have a simple argument that you can verify in polynomial time—at least, not unless he proves NP = co-NP.

Assume that your friend tells you that you *can* infer something, but at the same time claims that he cannot give an argument for this fact that you can verify in polynomial time. You are willing to concede he is correct, but would at least like to hear an argument for why he cannot give you an easily verifiable argument. Again you will be disappointed, because your friend cannot give you such an argument either—at least, not without proving that NP $\neq$ co-NP and therefore also P $\neq$ NP.

Finally, our result is relevant to another paper by Kaye on Minesweeper[2], where he considers playing Minesweeper on boards of infinite size. In this more recent paper, Kaye shows that playing infinite Minesweeper is co-RE complete and finds this curious, as NP-complete finite problems usually correspond to RE-complete problems in the infinite setting. In light of our result, we can now explain this discrepancy: In the finite setting Kaye's Minesweeper Consistency problem looks for a consistent placement of mines for an arbitrary board configuration, whereas in the infinite setting his generalized consistency problem asks for a consistent mine placement that also extends the input configuration. This change means that Kaye's co-RE complete infinite version of Minesweeper is not a generalization of his original NP-complete Minesweeper Consistency problem but of the co-NP-complete Minesweeper Inference problem that we have presented in this article.

ACKNOWLEDGMENTS

We thank Hausi, Curtis, and Brian Muller, and Denise Broekman for fruitful discussions on the Minesweeper game. Allan Scott and Ulrike Stege are supported by an NSERC discovery grant awarded to Ulrike. Allan Scott furthermore acknowledges the financial support of the University of Victoria in the form of a student fellowship. Parts of this article were written while Iris van Rooij was supported by the Department of Human-Technology Interaction and the J. F. Schouten school at TU Eindhoven. We further are grateful to our anonymous referees for helpful suggestions.

[2]Richard Kaye, *Infinite versions of Minesweeper are Turing complete*. http://www.mat.bham.ac.uk/R.W.Kaye