RICHARD KAYE

Minesweeper is NP-complete

NP-completeness

Many programming problems require the design of an algorithm which has a "yes" or "no" output for each input. For example, the problem of testing a whole number for primality requires an algorithm which answers "yes" if the input number x is prime, and "no" otherwise.

In trying to devise an algorithm to solve a given problem, one aspect of obvious practical importance is the time it takes to run. Since a typical algorithm may take more time on some inputs than others, the running time of an algorithm is usually regarded as a function of the input. For technical reasons, it is convenient to consider the way this function varies with the *number of symbols* required to write the input. (This number of symbols is usually denoted by n.) For example, for the input 17, our algorithm may require this number to be written in binary (as 10001), so here $n = 5$.

Different algorithms for the same problem may run in different amounts of time, due perhaps to the different coding methods used or to different theoretical bases for the algorithms. However, it may be that for a particular problem, all valid algorithms can be shown to take at least a certain amount of time, due to the inherent difficulties in the problem being solved. *Complexity theory* aims to study the *inherent* difficulties of problems, rather than the time or memory resources used by any particular algorithm or program.

It is certainly possible to find problems that can only be solved on a computer using a huge amount of time. It is also possible to find sensible-sounding problems that cannot be solved on a computer at all! However, there are two classes of problems that are of greatest interest for complexity-theorists.

The first of these classes is the collection, P, of *Polynomial-time computable* problems. These are the problems that can be solved on a normal computer and within an amount of time of order n, or n^2, or n^3, or n^4, ... (As before, n is the number of symbols required to write down the input to the problem. Note in particular that the running time of such a program is bounded by a polynomial in the *length of the input,* not the input itself.)

Of course, for a rigorous treatment of the subject, a precise definition of the mathematical model of computer we are using and what constitutes the running time of the computer, must be given. For the purposes of this article I will be less precise, but give here the two main points. Firstly, our computers will have an unlimited amount of memory—that is to say that they always have enough memory to complete the computation in hand. This does not seem particularly restrictive, as any terminating computation can only use a finite amount of memory anyway, and for most algorithms considered here, the amount of memory required for any particular computation can be estimated fairly accurately in advance. Secondly, the time taken by the computer is the number of steps required, where a single step can only process a single character's worth of information and a "character" comes from a fixed alphabet. (Characters could be single bits, or bytes, or 32-bit words, or symbols from some other finite set, provided this finite set is specified in advance.) To give an illustrative example, observe that arbitrary natural numbers can be represented on such computers (as sequences of binary digits, for example) and two such numbers can be multiplied together, but the time taken to multiply these numbers will not be a single step—it will instead be a function of the length of the numbers, for the computer can only process the numbers character-by-character.

A large amount of heuristic evidence exists supporting the thesis that the notion of a polynomial-time computable problem is independent of the particular computer model used. That is, if a problem is solved in polynomial time on one computer then the algorithm used can be transferred to a different kind of computer and will also run in polynomial time there. There is also strong evidence that suggests that the complexity class P consists of precisely those problems that are soluble *in practice* on an ordinary computer. Problems not in P may be theoretically soluble, but only with impractical running times even on the very fastest computer.

The second class of problems of interest is the class of *Nondeterministic Polynomial-time computable* problems, NP. These are problems that can be solved in polynomial time as before, but on a special "enhanced" computer able to perform "nondeterministic" algorithms. The reason for the interest in NP is that this class contains a great many problems of significant practical importance that are not known to be soluble by an ordinary polynomial-time algorithm, including some very well-known problems such as that of the "travelling salesman."

To define NP, we just need to explain the idea of a *nondeterministic* algorithm. These algorithms are like ordinary ("deterministic") ones except that there is an extra kind of instruction allowed which instructs the computer to guess a number. The computer performing this instruction is assumed to have the very special ability always to make a correct guess if one is available, and it is this aspect of nondeterminism that is difficult to implement in practice! Having made a guess, the nondeterministic algorithm is required to verify that the guess was indeed a correct one, because only by doing this can it determine whether a correct guess was possible at all.

For example, it is easy to use nondeterminism to tell if a whole number input x is composite (i.e., not prime). The machine should guess two whole numbers y, $z > 1$ and compute their product, yz. If $yz = x$ then the machine has verified that the guess was correct, so may answer yes, the number x is composite. If $yz \neq x$ then the machine may safely answer no, as in this case it is allowed to assume that no better guess was available, i.e., that x really is prime. Since a single multiplication can be carried out rather quickly, this nondeterministic machine will decide if a number is composite very rapidly without any lengthy search over all the possible factors.

A nondeterministic machine is not allowed to guess the answer ("yes" or "no") to the problem and output that, because the machine would not have verified this guess. The special power of these machines lies in the fact that it is not necessary to verify that any particular guess was *incorrect* (because only correct guesses are chosen if they are available). It is only required to verify that a guess is *correct*. Because of the different nature of these "yes" and "no" answers, it is not always true that the complement of a problem solvable using nondeterminism is as easy to solve nondeterministically. In the case of composite and prime numbers, for example, it is not immediately obvious how one might show that the set of primes (the complement of the set of composites) is recognizable in polynomial time by a nondeterministic algorithm. The problem here is to guess something that shows the input x is prime, and then to verify our guess quickly, but what should we guess? In fact, there is just such a "certificate of primality," as was first observed by Pratt[1] (see Figure 1).

This algorithm is based on the property that a number $x > 2$ is prime if and only if there is y such that $y^{x-1} \equiv 1 \bmod x$ and $y^q \not\equiv 1 \bmod x$ for all $q < x - 1$. It is recursive in the sense that it calls itself with smaller values.

1. On input x, if $x = 2$ answer "yes," and if $x = 1$ answer "no." Otherwise go to the next step.
2. Guess y and verify that $y^{x-1} \equiv 1 \bmod x$. (If this fails, answer "no" and stop.)
3. Guess a prime factorisation $a_1 a_2 \ldots a_n$ of $x - 1$ and run the algorithm recursively to check that each a_i is prime.
4. Verify that $2^{(x-1)/a_i} \not\equiv 1 \bmod x$ for each prime factor a_i of $x - 1$. If any of these fail, answer "no;" otherwise answer "yes."

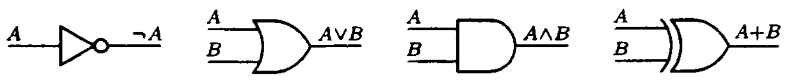
Figure 1. Pratt's nondeterministic algorithm for primality.

Needless to say, no "nondeterminism chip" has been developed to use in real computers (though some believe that quantum mechanics implies that something rather like nondeterminism might be built into a usable device).

As already mentioned, the class NP of *Nondeterministic Polynomial-time* problems is the class of problems that can be solved in polynomial time on a nondeterministic machine. It is generally believed that nondeterminism really does introduce problems that were not already in P, and also that there are NP problems whose complement does not lie in NP, but here lies the main problem. To date, no one has managed to find an NP problem and *prove* it is not in P. The famous "P = NP" question is whether there is such a problem. This is one of the most important open problems in mathematics—perhaps even *the most* important open problem. It has the same status as Fermat's last theorem before Wiles's solution, with a long history (going back well before computers). The majority of mathematicians believe that P and NP really are different (though several well-respected mathematicians consider it quite plausible that P = NP), but no one has a proof. Every mathematician dreams of solving a problem like this, and a huge number have tried, but no one has succeeded.

The difficulty of proving that P $\neq$ NP is not due to lack of examples of interesting problems in NP. In fact, mathematicians now have a huge list of problems—including the travelling salesman and many others of practical interest—

[1]V.R. Pratt, "Every prime has a succinct certificate," *SIAM J. Comput.* 4 (1975), 214–220.



A	$\neg A$
T	F
F	T

A	B	$A \vee B$
T	T	T
T	F	T
F	T	T
F	F	F

A	B	$A \wedge B$
T	T	T
T	F	F
F	T	F
F	F	F

A	B	$A + B$
T	T	F
T	F	T
F	T	T
F	F	F

A *boolean circuit* is a circuit built of the familiar logic gates such as AND ($\bigwedge$), OR ($\bigvee$), XOR ($+$), and NOT ($\neg$), each with inputs that may be true (T) or false (F). A circuit will have several inputs labelled $p_1, p_2, \ldots, p_n$ and an output q. The problem SAT is

> Given a boolean circuit C, is there some combination of true/false values for the inputs of C so that the output of C is true?

There are algorithms to answer this question, but none running in polynomial time is known. The obvious algorithm (to check all possible combinations of the inputs of C) takes too long, as there are 2^n combinations for n inputs. SAT is NP-complete.

Figure 2. The NP-complete problem SAT.

which are in NP and for which we have a proof that if P $\neq$ NP, then the problem is *not* in P. A problem, A, is typically shown to be of this type by proving that it is NP-*complete*, i.e., that every other NP problem, B, can be solved by a deterministic polynomial-time program which converts its input, x, for the problem B to an input, $f(x)$, for the problem A, with the property that the answer to problem B for input x is the *same* as the answer to problem A for input $f(x)$. If there is a polynomial-time computable function $f(x)$ with these properties, we say the problem B *reduces to* the problem A. Loosely speaking, a problem B reduces to a problem A, if A "includes" all instances of B as special cases, and the NP-problem A is NP-complete if it "includes" (in this sense) *all* other NP-problems.

To see the importance of this, consider a problem B in NP, and suppose also that we are given an NP-complete problem, A. Then there is a polynomial-time computer program that converts each instance, x, of the problem B to an instance, $f(x)$, of the problem A. But if our NP-complete problem A is actually in P, the problem A for $f(x)$ can be solved in polynomial time by a deterministic algorithm, hence B also can be solved in deterministic polynomial time, because the answers for A on input $f(x)$ and B on input x are the same.[2] This also applies to any other C in NP (with a different function $f(x)$ of course), so if A is in P, then every problem in NP will be in P, i.e., P = NP.

Cook[3] and, independently, Levin[4] first showed that NP-complete problems exist. In particular, the problem SAT of logical satisfiability (see Figure 2) is NP-complete. Although there are a great many NP-complete problems of practical importance, no one has found one which may be solved by a polynomial-time algorithm, and it is widely believed that no such exist. Turning a necessity into a virtue, many people have attempted to design cryptosystems so that a potential codebreaker would have to solve an NP-complete problem in order to break the code—taking too much time even on the fastest computer. Either way, an answer to the P = NP question would have significant practical importance.

The Minesweeper Game

Many of the ideas mentioned above may be illustrated effectively with a game many readers will be familiar with. *Minesweeper* comes with Microsoft's Windows operating system.[5] In it, the player is presented with an initially blank grid. Underneath each square there may be a mine, and the object of the game is to locate all these mines without being blown up. You select a square to be revealed; if it is a mine you are blown up (and the game is over), but with luck, perhaps it isn't. In this second case, when the square is revealed you see a number from 0 to 8, which is the number of mines in the eight immediately neighbouring squares. Figure 3 shows a typical position in such a game. The numbered squares are the squares that have been revealed, and no others have been uncovered yet. Two of the unrevealed squares are marked with a *, and these squares have already been identified as having mines in them. The others have been labelled with letters for identification.

[2]There is an important technical consideration omitted from the argument here: if A is in P, then the running time for the algorithm for A on input $f(x)$ is bounded by a polynomial in the length of $f(x)$, not the length of x itself. However, $f(x)$ itself is computed by a polynomial-time algorithm, and it is straightforward to deduce from this that the length of $f(x)$ is itself bounded by a polynomial in the length of x, so the algorithm just outlined for B is really polynomial time in the input x.

[3]S.A. Cook, "The complexity of theorem proving procedures," *Proc. Third Annual ACM Symposium on the Theory of Computing* (1971), 151–158.

[4]L. Levin, "Universal search problems," *Problems of Information Transmission* 9 (1973), 265–266.

[5]"Windows" is a trademark of Microsoft. The author has no connections with Microsoft, and nothing here should be regarded as comment on any of Microsoft's products.

F	D	2	1	2	1
A	A	3	*	4	B
2	2	3	*	5	B
0	0	1	1	4	B
0	1	1	1	2	B
0	1	C	E	E	E

Figure 3. An example position in *Minesweeper*.

	2	2	2	2	
	2	0	0	2	
	2	0	0	2	
	2	2	2	2	

Figure 4. Determine the location of all mines.

Faced with such a position in a game, there are several things one can deduce about the position of the mines, and which squares can be revealed safely. First, the squares marked A have mines, because of the 2s just below them. Next, the squares marked B also have mines because of the 4s and the 5 to their left. (These numbers include the two previously identified mines marked with stars.) Similarly, the square C has a mine. It follows that the squares marked D and E are clear since the mines at A, B and C account for the numbers neighbouring these squares. At this stage, it is not possible to determine if square F has a mine or not. However, the player may mark the identified mines A, B, C and uncover the safe squares D and E, and from the number revealed at square D (a 2 or a 3) determine if square F is safe or has a mine, thereby clearing the whole board.

Now that the rules of the game have been explained, the reader may like to consider the configuration in Figure 4. This particular game is played on a 6×6 board, and sixteen squares are revealed as shown. It is possible to deduce the location of all the mines from the information given.

The general *Minesweeper problem* is: Given a rectangular grid partially marked with numbers and/or mines, some squares being left blank, to determine if there is *some* pattern of mines in the blank squares that give rise to the numbers seen. In other words, to determine if the data given are *consistent*. This is a typical yes/no problem, as discussed above, and if we could solve this problem efficiently on a computer, we would have an excellent method for playing the game. To determine if a square is safe, we could write down the configuration we currently see with a single change made by marking the square in question with a mine, and feed this into the computer; if the computer says this pattern is inconsistent, then there is no mine at the square in question and it is safe to reveal it, otherwise there may be a mine. Similarly, by changing the description of the square in question to one containing a "0", then a "1", and so on up to "8", we may determine if it is correct to identify a mine at that square.

The Minesweeper problem is in NP, for to determine if an incomplete description is consistent, it suffices to guess the positions of the mines and then verify that these mines produce the numbers seen. It is not at all clear whether the complementary problem—whether some input configuration is inconsistent—is in NP, for what might we guess to show inconsistency? It is also reasonably straightforward to see that the Minesweeper problem can be reduced to SAT, for the rules of the game and any particular configuration can be described by a boolean circuit (see Figure 5).

In fact, the Minesweeper problem is NP-complete. This means it is just as difficult as any of the other NP-complete problems (such as SAT, the travelling salesman, and so on) and it is highly unlikely that there is an efficient algorithm

Consider a three-by-three block of squares labelled as shown.

a	b	c
d	e	f
g	h	i

Let a_{m} denote "there is a mine at a," and for $0 \leq j \leq 8$ let a_j denote "there is no mine at a and precisely j mines in the neighbouring squares around a"; and similarly for $b, c, d, \ldots, i$. Then the rules of Minesweeper for the centre square e can be described by the following statements:

1. precisely one of $e_{\mathrm{m}}, e_0, e_1, \ldots, e_8$ is true;
2. for $k = 0, 1, \ldots, 8$, if e_k is true then precisely k of $a_{\mathrm{m}}, b_{\mathrm{m}}, c_{\mathrm{m}}, d_{\mathrm{m}}, f_{\mathrm{m}}, g_{\mathrm{m}}, h_{\mathrm{m}}, i_{\mathrm{m}}$ are true;

and these can all be expressed (in a rather cumbersome fashion) by boolean circuits in the 90 inputs $a_{\mathrm{m}}, a_0, \ldots, i_8$. If we let C be the circuit consisting of all of these circuits for all points in the rectangular grid in place of e, the outputs of all these being combined into a single AND gate, then the Minesweeper problem becomes equivalent to an instance of SAT: given certain inputs for C being true or false, are there truth values for the other inputs that makes the output of the whole circuit C true?

Figure 5. Reduction of the Minesweeper problem to SAT.

for solving it. One way to prove Minesweeper is NP-complete is to show how to build "computers" using Minesweeper configurations. Since computers can be thought of as being made out of wires and logic circuits, that is what we will try to imitate in Minesweeper. In fact, as SAT is NP-complete, this suffices, because we will have shown how SAT reduces to Minesweeper, and Cook's result shows that any NP problem can be reduced to SAT.

Boolean Circuits in Minesweeper

Examine the configuration in Figure 6. (Here, again, the letters x and x' label unrevealed squares which may or may not contain mines.) A moment's thought will show that there are just two possible configurations: either all of the squares marked x contain a mine and those marked x' do not, or else the other way round.

We shall regard this as a wire carrying a value which is true or false depending on whether the xs or the x's have the mines. To define the value true or false carried in the wire precisely, we arbitrarily choose a direction for the wire—here going from left to right—and say that the value is *true* if the xs are mines, and *false* otherwise. In other words, if the squares just behind the centre 1s are mines ("behind" meaning in the sense of the chosen direction of the wire) then the value carried is true, and it is false otherwise. Note in particular that the truth of the signal in the wire is defined relative to its direction and the position of the centre 1s, not in terms of any absolute position on the grid.

We will need to be able to bend wires, and to split them. Figures 7 and 8 show how to do this. Figure 7(a) is a simple 90° bend in the wire. Figure 7(b) shows how a wire can be terminated. In these two diagrams, the squares marked * have mines in them. Such configurations can be given by explicitly marking the square as having a mine, but in all of the configurations here it is not necessary to do so. In these and all the following diagrams, the areas outside the bounding lines are assumed known to contain 0s, and in particular do not contain mines, and all the positions of the mines indicated by *s on the diagrams can be deduced from the numbers given. For (a), the mines are located by the 1221 to the top and to the right and the 3 between them, and in (b) the mines are located by the 12321 to the left. Figure 8 shows a way of "splitting" a wire. Notice that the outputs are two signals X and an inverted signal X'. Any of these wires can be terminated by a piece as in Figure 7(b) and the splitter can be combined with bends and further splitters, to make splitters with any number of outputs.

Figure 9 shows how to construct a NOT gate (similar to part of the splitter in Figure 8). This is obviously an important device for logic circuits, but it is useful here in one other important respect: since we defined truth/falsity in a wire relative to the position of the centre 1s in the wire, we may find a problem when we want to combine two or more signals if they are not aligned correctly. Figure 10 gives a configuration made from two NOT gates and provides one possible solution: this device enables the alignment of the 1s in three-by-three blocks to change so that the wire in question can be used as the input to some other device placed just about anywhere on the grid. (It is also possible to make a phase-changer out of three bent pieces of wire of the correct length.)

So far the configurations have been comparatively simple, but in order to mimic arbitrary boolean circuits we will need to have AND, OR, XOR gates, and so on. At first sight, it would seem that just the AND gate will suffice, because (as is well known from digital circuits) any gate or boolean circuit can be made from a combination of AND and NOT gates. For example, we can make up an OR gate from AND and NOT gates by the familiar formula $A \vee B = \neg(\neg A \wedge \neg B)$. But in principle there could still be a difficulty, in that we have not

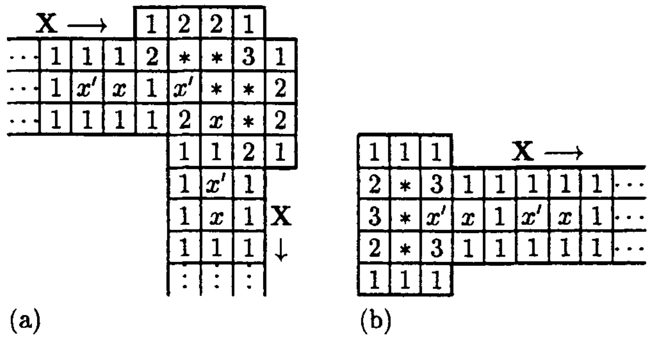


Figure 7. (a) A bent wire. (b) A terminated wire..

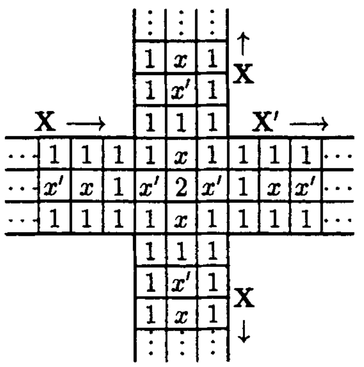


Figure 8. A three-way splitter.

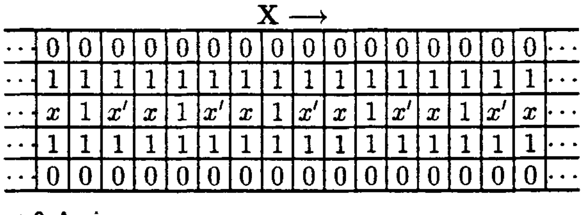


Figure 6. A wire.

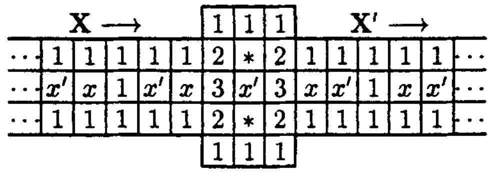


Figure 9. A NOT gate.

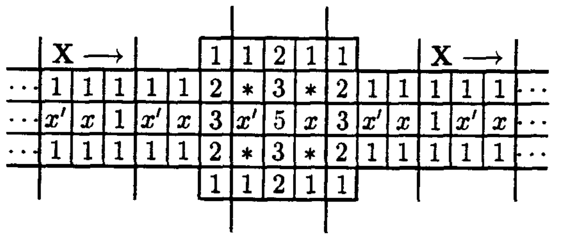


Figure 10. A phase-changer made from two NOT gates.

yet provided any method by which two signals can cross over each other.

In fact, this turns out not to be a problem after all. Crossing two wires over is clearly not going to be possible in the plane, but Figure 11 shows that a crossing of two wires can be simulated in the plane by using three splitters and three XOR gates. An XOR gate can in turn be made out of AND and NOT gates, as Figure 12 shows. (Planar circuits like these were discovered by Goldschlager.[6])

Figure 13 shows how an AND gate can be constructed. This is rather more complicated than previous ones. It takes two input wires U and V, has one output wire, labelled T, and has a central square at the heart of the gate (containing a 4) which is where the signals are combined.

The AND gate has two internal wires, R,S, which are aligned and looped back to a splitter at the output T via a pair of devices labelled a_1, a_2, a_3 and b_1, b_2, b_3 To analyse what happens here, we first see what happens if the output T is true, i.e., if the ts are mines and the t's are clear. In this case, from the 3 above and below a_3, we have that a_2 and a_3 must be mines, so a_1 is clear, and s is a mine. Similarly r is a mine. Thus the central 4 already sees four mines—s, t, r, and the *—so u', v' are both clear and the inputs U, V are both true. This shows that if t is a mine, all the other unknown squares are determined, and it is straightforward to check that these values are consistent with the data given.

Now suppose one or both of u, v is clear of mines, i.e., at least one of the inputs is false. Then, as we have just seen, t must be clear, and t' must be a mine. The central 4 sees 2 or 3 mines out of the u', v', and the *. So either one or both of r, s must be mines. We need to check both cases are possible. But if s is a mine, it is easy to check that a_1 and a_3 being mines and a_2 clear is consistent with the data given. Likewise, if s is clear, then a_1 and a_2 being mines and a_3 clear is consistent. The argument is identical for r, so if one or both of the inputs U,V is false then the output T is false, and each case is consistent with the data given. Therefore the whole configuration represents an AND gate, as required.

With these building blocks, we now have enough information on how to convert boolean circuits to Minesweeper configurations. Figure 14 illustrates the idea for the formula $(P \vee Q) \wedge (R \vee \neg Q)$. We write a program which, given as input a boolean formula, constructs a Minesweeper configuration such as that in the figure. The crossed lines are cross-overs, the filled-in circles are splitters, boxes denote gates, and the lines are wires. Square brackets denote terminators, as in Figure 7(b), except that the terminator marked T forces this wire to have the value 'true'. (This can be done by simply cutting a wire going from left-to-right just after a vertical row of three 1s.) It is clear from the diagram how to devise an algorithm that converts an arbitrary boolean for-

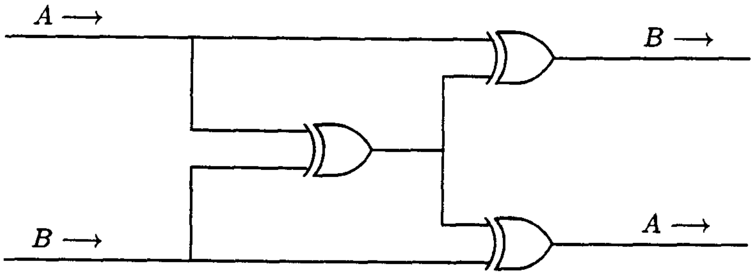


Figure 11. Crossing two wires with three XOR gates.

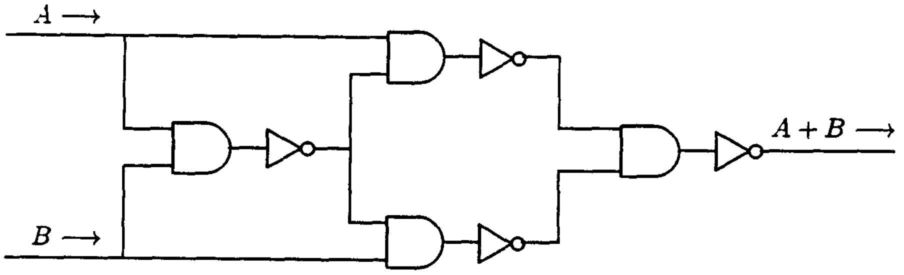


Figure 12. Making an XOR gate with AND and NOT gates.

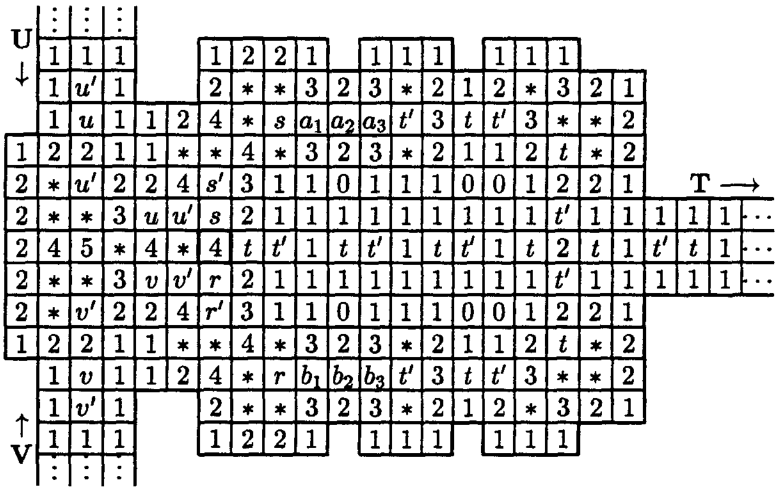


Figure 13. An AND gate.

[6]L.M. Goldschlager, "The monotone and planar circuit value problems are log space complete for P," *SIGACT News* 9(2) (Summer 1977), 25–29.

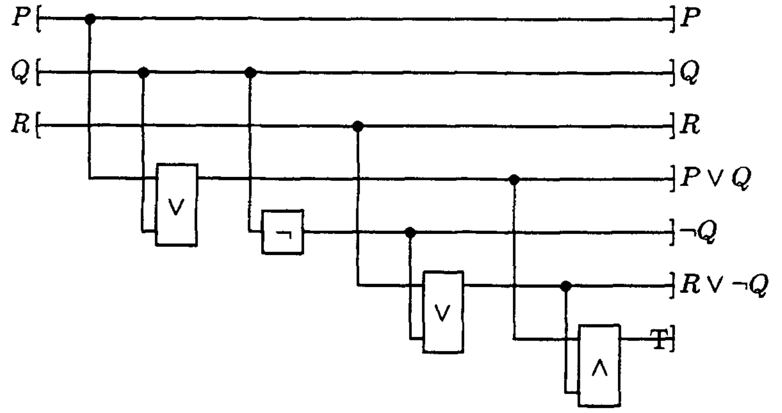


Figure 14. A Minesweeper circuit for $(P \vee Q) \wedge (R \vee \neg Q)$

mula to a Minesweeper configuration that is consistent if and only if the boolean formula is satisfiable. Each gate, terminator, and crossover can be put inside an $N \times N$ box (for some fixed value of N that can be predetermined), and the overall size of the configuration is therefore of the order of N^2n^2, where n is the number of symbols in the boolean formula. It follows that our program runs in polynomial time, and hence that SAT is reducible to the Minesweeper problem. But SAT itself is NP-complete, hence so is the Minesweeper problem.

It is worth pointing out that the NP-completeness proof just given is slightly stronger than originally stated. First, as has been pointed out, no $*$s need be given in the Minesweeper configurations used to test the satisfiability of boolean formulas. This is because the position of these mines can rapidly be deduced from the numbers in neighbouring squares. More interestingly perhaps—as the referee of this article has pointed out to me—the configurations may be taken to satisfy the condition that all squares neighbouring one marked 0 are uncovered. (Certainly all the gadgets in the figures satisfy this restriction.) This means that the action of the Microsoft Minesweeper program to automatically clear all such squares does not give any significant help for solving the Minesweeper problem.

Of course the configurations you can get in an actual game (where the mines are set at random by a computer) are unlikely to be like any of these boolean circuit configurations, so there remains a considerable art to playing the game, and there are many nuances and different kinds of deductions that one can make other than those used here. So it may even be that some polynomial-time algorithm is "good enough" at solving the sort of Minesweeper problems that occur in practice, even though (assuming P ≠ NP) it cannot actually solve all theoretically possible configurations. Many of the other NP-complete problems known are studied in the same way, with a view to finding algorithms that are not guaranteed to work, but do seem to work in most cases of interest. Finally, it is nice to know that to current knowledge, there may still be an efficient algorithm for Minesweeper, and finding it could solve one of mathematics's most important open problems.

Acknowledgment

The author would like to thank the anonymous referee for particularly helpful remarks on the first draft of this paper, which have proved invaluable in its revision.

AUTHOR

RICHARD KAYE

School of Mathematics and Statistics
The University of Birmingham
Birmingham, B15 2TT
England
e-mail: R.W.Kaye@bham.ac.uk

Richard Kaye studied first at Cambridge and then at Manchester, obtaining a PhD for work on models of arithmetic under the supervision of Jeff Paris. After six years of postdoctoral work at Oxford he went to his present position as senior lecturer at Birmingham. He has written a monograph on models of arithmetic and a textbook on linear algebra. He is a keen amateur trombonist, and he currently leads the trombone section of the Oxford Symphony Orchestra.