The Complexity of Puzzles: NP-Completeness Results for Nurikabe and Minesweeper

---

A Thesis
Presented to
The Division of Mathematics and Natural Sciences
Reed College

In Partial Fulfillment
of the Requirements for the Degree
Bachelor of Arts

Brandon P. McPhail

December 2003

Approved for the Division
(Mathematics)

_____

James D. Fix

Acknowledgments

I would like thank my family and close friends for their constant love and support.

I also give special thanks to Jim Fix, who has been so enthusiastically generous with his time and thoughts, and to members of my orals committee for the stimulating discussions and helpful suggestions.

Table of Contents

List of Figures

Abstract

In this thesis we provide a formal treatment of puzzles. We develop the mathematical notion of a solution to a puzzle and attempt to precisely characterize when a family of puzzles is "hard". We characterize our ability to solve a family of puzzles as depending on whether or not a feasible algorithm can be devised to find solutions, in general, to puzzles in that family. Our notion of "hardness" is based on the number of operations that the solving algorithm requires to solve puzzles in that family as a function of the size of the puzzle. Thus, we study puzzles by looking at the inherent computational complexity of solving puzzles, borrowing concepts from computability and complexity theory. For certain families of puzzles, we show that determining whether they are "hard" is simply a matter of answering the famous open question in computational complexity: "Does P=NP?" Finally, NP-completeness results are given for the puzzles *Nurikabe* and *Minesweeper*.

Chapter 1

Computational Preliminaries

You said it ... one of those Recognizers comes after me, gonna hafta jump clear out of the data stream.

– Tron, 1981

In this thesis we provide a formal treatment of puzzles. We develop the mathematical notion of a solution to a puzzle and attempt to precisely characterize when a family of puzzles is "hard". We characterize our ability to solve a family of puzzles as depending on whether or not a feasible algorithm can be devised to find solutions, in general, to puzzles in that family. Our notion of "hardness" is based on the number of operations that algorithm requires to solve puzzles as a function of the size of the puzzle. Thus, we study puzzles by looking at the inherent computational complexity of solving puzzles, borrowing concepts from computability and complexity theory. For certain families of puzzles, we show that determining whether they are "hard" is simply a matter of answering the famous open question in computational complexity: "Does P=NP?" Finally, NP-completeness results are given for the puzzles *Nurikabe* and *Minesweeper*. In the first chapters we introduce the formal concepts upon which our proofs will rely.

1.1 Strings, languages, and alphabets

In some sense, computation can be viewed as a process of manipulating some set of symbols, whether we are alphabetizing a list of words, computing the sum of two integers, or increasing the brightness of red values in a digital image. Each symbol carries no intrinsic meaning; they serve only as distinct placeholders. The concept of a bit, for example, could be conveyed just as effectively if we spoke of **a**s and **b**s in place of **0**s and **1**s. Formally, any non-empty finite set forms an alphabet from which we can draw our symbols.

Definition 1.1. *An **alphabet** is a non-empty finite set.*

Consistent with our notion of letters in an English or Russian alphabet, symbols can be strung together to form words or *strings* from our alphabet. For our discussion,

we denote an alphabet by Σ.

Definition 1.2. *A **string over** an alphabet Σ is a finite sequence of symbols from Σ. We use $|x|$ to denote the **length** of a string x over Σ. The symbol ϵ is reserved for the string of length zero called the **empty string**. Σ^* denotes the set of all strings over Σ.*

For example, `010110` is a string over the alphabet $\{\mathtt{0}, \mathtt{1}\}$ just as `bubba` and `bobby` are strings of length 5 over $\{\mathtt{a}, \mathtt{b}, \mathtt{o}, \mathtt{u}, \mathtt{y}\}$.

Definition 1.3. *A **language** is a subset of Σ^*.*

The set of strings over $\{\mathtt{0}, \mathtt{1}\}$ containing an even number of `0`s is an example of a language. Similarly, the set of three-letter English words such as `cat` or `bog` forms a language over the alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}, \ldots, \mathtt{z}\}$ and is a proper subset of Σ^*.

1.2 Turing Machines

By adopting the machinery of set theory, our formalisms now allow us to ask some very interesting questions. Some of the most well-known problems in mathematics can be phrased as tests for inclusion of a string in a language. Around the early twentieth century, as physicists and engineers began to construct machines capable of computing answers to complicated problems faster than could a human, mathematicians were already making remarkable discoveries about the fundamental limits of a computer. Central to these results was the development of an abstract model for computation, developed by Alan Turing[17] in 1936. Simple yet surprisingly powerful, Alan Turing's *Turing machine* is computationally equivalent to a modern computer and can perform any computation of today's PC. Perhaps more importantly, a modern computer is incapable of all computations that a Turing machine is incapable of. For our purposes, we will use Turing machines to define more precisely our notion of computation, algorithms, and the time they take to answer our questions about puzzles.

Informally, a Turing machine, depicted in Fig. 1.1, is a computer which has a finite control that directs the reading and writing of symbols on an infinitely long tape. That control, which serves as the program for the Turing machine, can be in one of a finite set of states. Initially, the tape contains only a string representing the input followed by blanks ($\sqcup$). The machine moves its tape head along the tape, with each step reading a symbol, writing a symbol, and then changing its state accordingly. At any time the machine may enter a final state and output Accept or Reject. Also like a modern computer, a Turing machine might never halt and instead run forever.

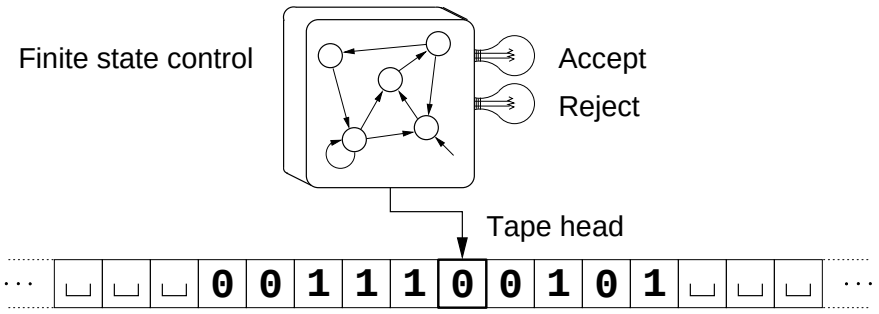


Figure 1.1: The deterministic Turing machine

1.2.1 Determinism

Definition 1.4. *Formally, a deterministic* ***Turing machine*** *(DTM) is a 7-tuple* $(Q, \Sigma, \Gamma, q_0, q_A, q_R, \delta)$ *where*

1. Q *is the set of states,*
2. *the input is a string over the alphabet* Σ $(\sqcup \notin \Sigma)$*,*
3. *the tape symbols form a string over the alphabet* Γ*, where* $\sqcup \in \Gamma$ *and* $\Sigma \subseteq \Gamma$*,*
4. $q_0 \in Q$ *is the* ***start state****,*
5. $q_A \in Q$ *is the* ***accept state****,*
6. $q_R \in Q$ *is the* ***reject state*** $(q_R \neq q_A)$*, and*
7. $\delta \colon (Q \setminus \{q_A, q_R\}) \times \Gamma \to Q \times \Gamma \times \{L, R\}$ *is the transition function.*

Computation on a DTM proceeds as follows. Initially, the tape contains only an input string $x = x_1 x_2 \ldots x_n$ where $x_i \in \Sigma$. The rest of the tape is blank. The finite control is set to state q_0, and the tape head reads in the first symbol x_1 of the input string. For each step of computation, the DTM, given the current state and symbol under the tape head, writes a symbol to the tape, moves the tape head left or right, and changes its state. This action is encoded by the transition function δ. Namely, in state q reading w, δ maps the pair (q, w) to a triple (q', w', d) such that

- the finite control transitions from q to the state $q' \in Q$,
- the symbol $w' \in \Gamma$ overwrites w on the tape, and
- the tape head moves either one cell to the left, if $d = L$, or one cell to the right, if $d = R$.

Successive applications of the transition function on states and symbols from the tape yield a process of computation for a given input string. We wish to describe this process of computation by identifying each stage of the computation. Formally, each step of the DTM on an input string is determined by a configuration consisting of the current state, the contents of the tape, and the position of the tape head.

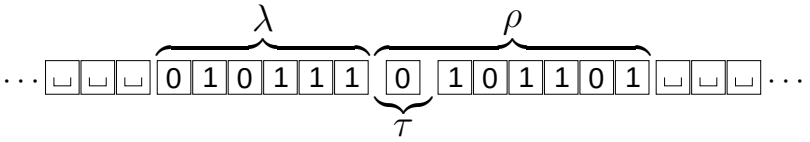


Figure 1.2: A configuration (λ, q, ρ)

Definition 1.5. *A **configuration** is a triple (λ, q, ρ) where*

1. *λ is the string of tape symbols from the first non-blank symbol on the tape up to but not including the symbol τ under the tape head,*

2. *q is the current state, and*

3. *ρ is the string of tape symbols starting with the symbol τ under the tape head up to and including the last non-blank symbol in the tape.*

*If x is the input string for our DTM, then $C_0 = (\epsilon, q_0, x)$ is the **start configuration**. In our definitions we treat the configuration (λ, q, ϵ) as the configuration $(\lambda, q, \sqcup)$ in order to have a symbol under the tape head. We consider these configurations equivalent. In addition, any configuration containing the state q_A is an **accepting configuration**. Similarly, we call any configuration containing q_R a **rejecting configuration**.*

The running of a DTM on some input can now be described by the exact sequence of configurations it assumes. Since δ is a function, every configuration leads to a unique successive configuration. If a DTM ever reaches the same configuration twice, it will loop forever; otherwise, the DTM will either halt or produce infinitely many configurations.

Definition 1.6. *A configuration $(\lambda\sigma, q, \tau\rho)$ **yields** a configuration $(\lambda\alpha, q', \beta\rho)$ if either*

1. *$\delta(q, \tau) = (q', \tau', R)$, $\alpha = \sigma\tau'$, and $\beta = \epsilon$, or*

2. *$\delta(q, \tau) = (q', \tau', L)$, $\alpha = \epsilon$, and*

$$\beta = \begin{cases} \sqcup\tau' & : \quad \lambda\sigma = \epsilon \\ \sigma\tau' & : \quad \text{otherwise.} \end{cases}$$

We write $(\lambda\sigma, q, \tau\rho) \vdash (\lambda\alpha, q', \beta\rho)$ to denote this relation. If there exists a sequence $C \vdash C_1 \vdash C_2 \vdash \cdots \vdash C_{k-1} \vdash C'$, we write $C \vdash^k C'$. Similarly, if there exists some $k \geq 0$ such that $C \vdash^k C'$, we write $C \vdash^* C'$.

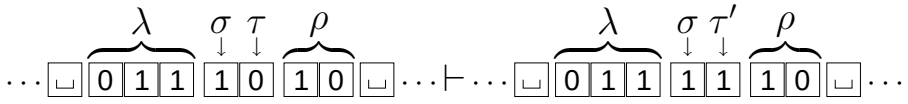


Figure 1.3: Configuration (λ, q, ρ) yields another configuration

Definition 1.7. *A **computation** $\mathcal{C}$ is a sequence of configurations $C, C_1, C_2, \ldots, C_k$ where*

$$C \vdash C_1, C_1 \vdash C_2, \ldots, C_{k-1} \vdash C_k.$$

*If C_k is an accepting configuration, then we call $\mathcal{C}$ an **accepting computation**. Otherwise, $\mathcal{C}$ is a **non-accepting computation**, and if C_k is a rejecting configuration, then $\mathcal{C}$ is also a **rejecting computation**.*

The set of input strings accepted by a DTM is a language over the input alphabet. Since the question of membership in a language is so central to our discussion of computability, we should make clear our notion of a language accepted by some DTM.

Definition 1.8. *M **accepts** $x \in \Sigma^*$ if there exist some λ and ρ in Γ^* such that*

$$(\epsilon, q_0, x) \vdash^* (\lambda, q_A, \rho).$$

*Likewise, M **rejects** x if there exist some λ and ρ in Γ^* such that*

$$(\epsilon, q_0, x) \vdash^* (\lambda, q_R, \rho).$$

*In either case, we write $M(x) = \lambda\rho$ to denote that M **outputs** the string $\lambda\rho$ when **run on** input x.*

Some strings are accepted by a DTM while others are rejected. Our DTM might also never halt. We would like to describe the sets of strings associated with each of these actions. Most importantly, we define the language of a DTM M as the set of all strings accepted by M.

Definition 1.9. *The **language of** M is the set*

$$L(M) = \{x \mid \exists \lambda, \rho \in \Gamma^* \colon (\epsilon, q_0, x) \vdash^* (\lambda, q_A, \rho)\}.$$

*We say that M **recognizes** a language L if $L = L(M)$. A language is **Turing-recognizable** if a Turing machine exists that recognizes it.*

Let us consider those DTMs that halt on any input. If a DTM enters a halting state for all inputs $x \in \Sigma^*$, we call it a decider.

Definition 1.10. *A DTM M is a **decider** if and only if for all $x \in \Sigma^*$ there exist $\lambda, \rho \in \Gamma^*$ such that*

$$(\epsilon, q_0, x) \vdash^* (\lambda, q_f, \rho) \text{ for some } q_f \in \{q_A, q_R\}.$$

*We say that M **decides** a language L if $L = L(M)$ and M is a decider.*

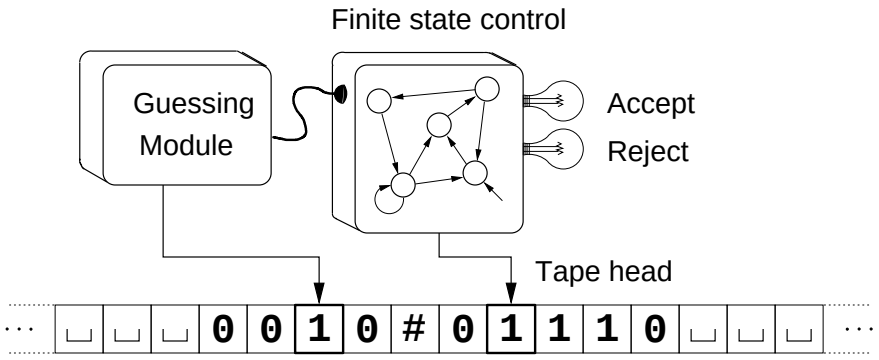


Figure 1.4: The nondeterministic Turing machine

1.2.2 Nondeterminism

Let us augment our Turing machine to produce a rather unrealistic, yet useful, model of computation. We model *nondeterministic* computation by outfitting our familiar deterministic Turing machine with the ability to "guess" any string from Γ^* and write it to the tape before computation begins. A **nondeterministic Turing machine** (NTM) is a DTM augmented with a symbol $\# \in \Gamma$ and a guessing module with its own write-only tape head.

Initially, the tape contains only the input string x. The tape head of the finite control is positioned over the first symbol of x. The write-only head is positioned over the first blank left of x. Computation of an NTM on an input string x then proceeds in two stages:

- Guessing stage:
  - Prior to entering state q_0, the finite control is inactive, and the NTM enters a guessing stage. First, the guessing head writes the symbol # to the tape and moves one cell to the left.
  - For each proceeding step of the guessing stage, either the write-only head writes some symbol in Γ and moves one cell to the left, or the guessing module is deactivated, the finite control is activated, and the Turing machine enters state q_0. The choice of whether to write a symbol or to leave the guessing stage is made completely arbitrarily.
- Checking stage:
  - If the guessing module halts, computation then proceeds deterministically on the contents of the tape according to the transition rules of our original Turing machine. The guessing module and write-only tape head are no longer involved.

The arbitrary nature of the decision to write a symbol or leave the guessing stage allows the guessing module to write any string from Γ^* to the tape before it halts – if it ever halts. The nondeterministic nature of our augmented Turing machine arises directly from the arbitrary fashion of this decision.

After the guessing stage is complete, the Turing machine has an initial configuration $(\sigma\#, q_0, x)$ where σ is the guessed string or **hint** written by the guessing module. The Turing machine may read symbols from σ and write to that part of the tape. Indeed, it is this ability to read any string from Γ^* as a hint for an input x that gives our NTM its rather unrealistic properties.

The notions of a **configuration** and a **computation** of an NTM are defined analogously to those of a DTM. However, since different possible computations may arise from a single input string depending on the hint chosen by the guessing module, we should carefully develop the notion of acceptance by an NTM. Notice that for a given input string x, there is a single computation for each hint in Γ^*. Thus, the number of possible computations for each x is infinite.

Definition 1.11. *An NTM M **accepts** $x \in \Sigma^*$ if there exists a $\sigma \in \Gamma^*$ such that the computation*

$$C_0, C_1, C_2, \ldots, C'$$

is an accepting computation, where $C_0 = (\sigma\#, q_0, x)$.

Definition 1.12. *The **language of** an NTM M is the set of all strings accepted by M. That is,*

$$L(M) = \{x \mid \exists\sigma \in \Gamma^* \exists\rho, \lambda \in \Gamma^* \colon (\sigma\#, q_0, x) \vdash^* (\lambda, q_A, \rho)\}.$$

*We say that M **recognizes** a language L if $L = L(M)$.*

1.3 Time complexity

1.3.1 Time complexity for DTMs

If we want to describe the efficiency of a Turing machine, we first need to decide on a measure of time. We define the running time of a DTM on a given input string as the number of computation steps taken to reach a halt state.

Definition 1.13. *The **running time** of a DTM M on input x is*

$$t_M(x) = \begin{cases} k & : \quad C_0 \vdash^k (\lambda, q_f, \rho) \text{ where } q_f \in \{q_A, q_R\} \\ \infty & : \quad \text{otherwise.} \end{cases}$$

Note that M may not halt on all inputs. If, on the other hand, M is a decider, then we can define the time complexity of a computation as the maximum number of steps required to halt for an input of a given length.

Definition 1.14. *The **time complexity** of a decider M for inputs $x \in \Sigma^*$ of length n is given by*

$$T_M(n) = \max\{t_M(x) \mid |x| = n\}.$$

1.3.2 Time complexity for NTMs

When defining the running time for an NTM, we want to consider the time taken for computations on all hints, even ones that don't halt, regardless of whether the input is accepted or not. The easiest way to do this is to consider the set of all computation lengths over all hints. The running time is the maximum element in that set, if one exists.

Definition 1.15. *The **running time** of an NTM M on input x is*

$$t_M(x) = \begin{cases} \max H_x & : \quad H_x \text{ is finite} \\ \infty & : \quad \text{otherwise} \end{cases}$$

where

$$H_x = \{k \mid (\sigma\#, q_0, x) \vdash^k (\lambda, q, \rho) \text{ for } \sigma \in \Gamma^*\}.$$

We define the time complexity function for M on inputs of a given length n as the maximum running time for any input $x \in \Sigma^*$ of length n.

Definition 1.16. *The **time complexity** of an NTM M for inputs of length n is*

$$T_M(n) = \max\{t_M(x) \mid |x| = n\}.$$

1.3.3 Time complexity bounds

When considering the time complexity of Turing machine, we wish to describe the "efficiency" of a Turing machine. The following definitions are essential to that discussion.

Definition 1.17. *We say that a function f is **in the order of** or **bounded asymptotically by** g if and only if there exists a fixed positive real number $c \geq 0$ and a fixed natural number k such that for all $n \in \mathbf{N}$ where $n \geq k$*

$$|f(n)| \leq c|g(n)|.$$

We write $f(n) = O(g(n))$ to denote this relation.

Effectively, this notation allows us, when specifying bounds for a function, to disregard coefficients and all but those terms of the highest order. So, for example,

$$f(n) = \frac{n^4}{10000} + 1000n^3 + 100000 = O(n^4).$$

Notice also that if $f(n) = n$, then not only does $f(n) = O(n)$, but $f(n) = O(n^2)$ and $f(n) = O(n \log n)$ as well.

The existence of a polynomial time bound turns out to be a good measure of what is "reasonable" for a Turing machine to compute. In this sense, we will use polynomial time bounds as evidence that a puzzle can be easily solved.

Definition 1.18. *If the time complexity of a Turing machine M (deterministic or nondeterministic) is bounded by a polynomial p, that is,*

$$T_M(n) = O(p(n))$$

*then we say that M is a **polynomial-time** Turing machine.*

Chapter 2

Introduction

Practical application is found by not looking for it, and one can say that the whole progress of civilization rests on that principle.

– Howard W. Eves, *Mathematical Circles Squared*

If history has taught us anything, it's the usefulness of puzzles.

Scoff if you like, but games and puzzles have been occupying and influencing the course of human thought for a very long time. Puzzles have entertained and inspired for millennia. The fascination of the ancients in simple puzzles planted a seed of inquiry that has grown to encompass a vast collection of modern mathematical and philosophical thought. Some early puzzles, such as *Magic Squares* or *ruler and compass constructions*, originated in ancient Egypt, Greece, and China and still challenge modern algebraists and combinatorists today.

Almost four thousand years ago an Egyptian puzzler scrawled the following on a sheet of papyrus:

Seven houses contain seven cats. Each cat kills seven mice. Each mouse had eaten seven ears of grain. Each ear of grain would have produced seven hekats of wheat. What is the total of all of these?

Preserved today, the Rhind papyrus of 1850 BC provides an example of early brain-teasers and mathematical games. The intriguingly simple question persuades us to engage in a mathematical, almost algorithmic style of thought – and it's fun. The puzzle may seem unimportant (actually it's a fine example of ancient Egyptian algebra), but to the scribe it was perhaps as papyrus-worthy as any of the other 83 puzzles on that same sheet. When calculating volumes for cylindrical granaries, for example, the scribe first calculates the area of the base and relies on a ratio of a circle's circumference to its diameter that is suggested to be 256 to 81. This was the best approximation of π (off by less than 1%) before Archimedes.

Today, puzzles can be found everywhere, and a mathematician should be pleased. Standard textbooks in mathematics and physics are often essentially collections of puzzles with complex descriptions. Unfortunately, what puzzles a quantum physicist or a logician may not be as interpretable or fun to an average reader of the Sunday newspaper, and so a market for "recreational" puzzles exists. Countless types of

puzzles (such as the crossword) have been invented to meet this demand, and often thousands of puzzle designs have been published for a single type of puzzle.[1]

New types of puzzles frequently attract the attention of mathematicians since they are not only fun but often express some rather interesting or subtle properties. Two-player games such as Chess, Go, and Hex and one-player games such as solitaire, crosswords, jigsaw puzzles, Tetris, and Rubik's cube have inspired thousands of papers.

For the rest of this thesis, we adopt the perspective of a puzzle designer. This should be fun. Our goal is to maximize the quality of our puzzles while minimizing the time it takes us to design them. We imagine a fictitious puzzler Bob, avidly seeking new challenging mind-benders to solve. The demand for puzzles is constant, and, with other professional "puzzle consultants" like Will Shortz out there, the competition is fairly stiff. Ideally, we would like a detailed method or recipe for generating high quality puzzles as quickly as we can.

What then constitutes a "good" puzzle?

Well, surely the puzzle mustn't be too easy. While a simple puzzle may be acceptable for the children's section of the local newspaper, we run the risk of boring sharp-minded Bob. We want to ensure that instances of our puzzles aren't easily solved, eliminating the possibility that some "trick," shortcut, or other efficient procedure might quickly yield a solution. If such a simple rule could be devised to easily solve our puzzles, we should assume that Bob will eventually discover the trick and become bored. In short, if we want "hard" puzzles, no procedure should solve our puzzles too efficiently.

But what does this mean? Before we go any further, we will need a clear understanding of how procedures might solve puzzles and how we should measure efficiency.

2.1 Algorithms

The notion of an algorithm is fairly intuitive. Informally, an algorithm is a sequence of instructions to perform a specific task. A cooking recipe provides an algorithm for the preparation of a specific meal. As any chef who has ever prepared a strawberry shortcake with sour cream might understand, each step of an algorithm must be well-defined and easily understood. Surprisingly, the term *algorithm*, despite its intuitive nature, escaped formal definition until only the last century when Alan Turing[17] introduced his theoretical computing machines. We will restrict our attention to a precise notion of an algorithm. Formally, our notion of an algorithm is any sequence of operations that can be performed by a deterministic Turing machine – it is a computer program.

Definition 2.1. *An **algorithm** is a DTM.*

[1] Since 1944, the New York Times alone has printed over 2500 Sunday crossword puzzles.

2.1.1 Encoding problems

Intuitively, a *problem* is a question for which we seek a specific answer. A description of a problem consists of a characterization of the question, and a set of parameters whose values are left unspecified. Each specification for the parameters yields an *instance* of the problem.

A **problem description** consists of

1. a complete description of the problem's parameters and
2. a statement of what properties the solution is required to satisfy.

Problems with a *Yes* or *No* answer are called **decision problems**.

For an example of a problem, we might ask "Given a natural number, is the number even?" A characterization of the problem EVENNESS includes a set of unspecified digits and the property of evenness. Any natural number, such as `38` or `421`, is an instance of this problem. The problem is *solvable* if an algorithm exists that can decide it. The answer to any instance is either Yes or No, so this is an example of a decision problem.

Since an algorithm is described by the computation of a DTM on some inputs, we require some formal machinery to interface these informal notions of problems with our definition of a Turing machine. Somewhat predictably, if a Turing machine is to perform a computation on an instance of a problem, we must *encode* each instance as an input string. Since the same Turing machine must solve each instance of a problem, the procedure for encoding instances of a particular problem as strings for a Turing machine must be consistent. In short, we require an *encoding scheme*.

If, for example, we choose to encode each instance of EVENNESS using a binary representation over $\Sigma = \{\mathtt{0}, \mathtt{1}\}$, then

$$\text{encode}(\mathtt{421}) = \underbrace{\mathtt{110100101}}_{9}.$$

The length of our input string is 9. Consider instead the following encoding of `421` over $\Sigma = \{\mathtt{0}, \mathtt{1}, \mathtt{2}, \mathtt{3}, \mathtt{4}\}$ (base 5):

$$\text{encode}(\mathtt{421}) = \underbrace{\mathtt{3141}}_{4}.$$

Now the length of our input string has been cut to 4. Since the running time of a Turing machine is often dependent on the length of its input, it appears that the particular encoding scheme we choose can heavily impact the time it takes to solve a problem.

While it may seem from our example that the encoding will drastically affect our running time, we will be considering only reasonable encodings. In practice, "reasonable" encoding schemes always vary at most polynomially from each other, and one would be hard pressed to imagine a sensible scheme that produces exponentially longer strings. As Garey and Johnson[7] point out, choosing an encoding scheme is a somewhat arbitrary procedure.

The Turing machine provides the essential underpinnings for a theory of computation and complexity. The question of what can be proved and solved becomes in many ways a question of what problems can be solved by a Turing machine. Quite remarkably, not all problems can be solved. As a classic example, the *halting problem* asks the question: "Given an encoding of a Turing machine M and an input string x, will M halt when run on x?" It turns out that the problem cannot be solved by a Turing machine.

Definition 2.2. *A* ***problem*** Π *is a language under some encoding scheme over an alphabet* Σ*. Each string in* Σ^* *is an* ***instance*** *of* Π*. A Turing machine* M *is a* ***solver*** *for* Π *if and only if it decides* Π*.*

2.2 Measuring complexity

Complexity theory deals with the resources required during computation to solve a certain problem. Typically, resources are space and time.

Space complexity is a measure of the number of tape cells visited by the Turing machine's tape head. Analogously, we could measure the amount of memory accessed by a computer program during execution or the number of beads used during a calculation on an abacus.

Time complexity is a measure of the number of steps taken by a Turing machine before it halts. We could describe the time complexity of a computer by measuring the number of CPU cycles or checking the elapsed time on the processor clock.

We could, however, pick some other resource as a measure of complexity.

Imagine, for a moment, that a large Turing machine were constructed of metal and grease. With each step of a computation, our 2-ton tape head pounds a symbol into a tape cell as it makes its way along the tape, made perhaps of a long strip of tin. Every time the tape head changes direction, however, a complicated whirling of gears slowly brings the hunk of metal to a halt and propels it in the other direction. Due to the overwhelming resources (the burning of fossil fuels?) required to accelerate and decelerate the tape head, we might choose as our measure of complexity the number of times the tape head changes direction during a computation.

At the end of every year at some liberal arts college, the math secretary begins the laborious chore of typing up completed math theses on her archaic typewriter. Unfortunately, her typewriter only includes room for a single character. If a different character is required, the secretary must remove the piece of metal type and retrieve the newly required character on a new piece. As would be her misfortune, the ruling communist regime keeps all metal type locked up securely in a town 300 miles away and limits secretaries' access to a single piece of type per calendar month. Every time the thesis student chooses the luxury of introducing a new character, the poor math secretary begins her 600 mile trek to the People's Type Repository and back before typing the next character. If we wished to speak of the computational complexity of a typing math secretary, an obvious measure of complexity would present itself.

For our purposes, we will use the running time of a Turing machine as our measure of complexity. We will say that a problem is tractable if a polynomial-time

DTM exists that solves it.

Problems encountered in subsequent chapters will be decision problems unless otherwise stated.

2.3 Complexity classes

If we put this machinery to use, we see that efficiently solving a puzzle requires defining the problem, choosing an encoding scheme, and finding a polynomial-time algorithm (DTM) that solves each instance. Defining the problem and choosing an encoding scheme seems easy enough, but how are we to know whether a polynomial-time solver even exists? We cannot simply test all possible algorithms, since there are infinitely many Turing machines to choose from.

Complexity theorists seek an understanding of the inherent difficulties of problems, not the running times of particular algorithms. A system of classification of problems allows us to differentiate between these inherent difficulties. Two collections of decidable problems form the most important complexity classes, P and NP.

We denote with P the set of decision problems solvable by a deterministic Turing machine in polynomial time. In a sense, P consists of the "easy" problems. This can be somewhat misleading, since, for the sake of generality, the time complexity function for a DTM completely ignores constants and considers only worst-case running times for membership in P and asymptotically long running times for lack membership in P.

Consider that a problem Π that has an algorithm that uses at most n^{1000} steps is in P, while a problem Π' requiring roughly $2^{\frac{n}{1000}}$ steps is not in P. For inputs of length greater than 1000, Π can indeed be solved more quickly than Π'; however, if we were to use current computing technology to solve the "easy" problem Π for an input of length $n = 2$, civilizations may rise and fall and the Earth may plunge into the sun before our computation would ever halt.

In practice, however, most polynomial-time algorithms tend to have reasonable bounds.

Definition 2.3. *A problem* Π *is in* P *if there exists a polynomial-time DTM that decides* Π*.*

NP denotes the set of decision problems decidable by a nondeterministic Turing machine in polynomial time.

Definition 2.4. *A problem* Π *is in* NP *if there exists a polynomial-time NTM that decides* Π*.*

Computation on an NTM is, in a sense, a parallelized form of "guess and check". Remarkably, an NTM always guesses a hint that results in an accepting configuration if such a hint exists. After guessing, computation proceeds deterministically. In order to confirm that the guess indeed leads to an accept state, the DTM associated with the latter part of this computation acts as a *verifier* for the guessing stage.

Definition 2.5. *A problem* Π *is* ***verifiable*** *if an NTM exists that solves it.*

Framed in the context of solving puzzles encoded as decision problems, an NTM must be able to verify the existence of a solution in polynomial time for the problem to be in NP.

Definition 2.6. *Let* Π *be a problem associated with some encoding scheme that encodes all possible solutions to* Π *as strings in* Γ^**. Given any instance of* Π*, an NTM* M *will accept if any string in* Γ^* *is a solution for* Π*. If the time complexity of* M *has a polynomial bound, then we say that* Π *is* ***polynomial-time verifiable****.*

Equivalently, the complexity class NP consists of those problems for which a solution can be *verified* in polynomial time.

2.3.1 P vs. NP

Since every deterministic computation is a possible nondeterministic computation, the class P is a subset of NP.

Theorem 2.1. $P \subseteq NP$.

Proof. Any DTM can be modeled by an NTM whose guessing module writes no symbols to the tape. Thus, given any problem Π, if a polynomial-time DTM M exists that solves Π, then an NTM M' can write no hint to the tape and perform the same computations as M. Thus, for all $\Pi \in P$, it follows that $\Pi \in NP$. □

The reader may instinctively ask if P is a proper subset, that is, if some problem Π is known to exist in NP that is not in P. While the dominant hunch among computer scientists seems to suggest that $P \subset NP$, the question of whether $P = NP$ or $P \neq NP$ remains the most important unsolved problem in theoretical computer science. As perhaps a testament to its importance, the Clay Institute has placed a $1,000,000 bounty on any solution. To claim this prize, one has three choices:

1. Find a problem $\Pi \in NP$ and prove that $\Pi \notin P$.

2. Prove that $P \stackrel{?}{=} NP$ is undecidable.

3. Find a polynomial-time algorithm for all problems in NP.

Supposing that the question can be decided, one strategy for proving either 1) or 3) is to examine relations between problems in NP. If we seek a problem not contained in P to prove 1), we should look for the particularly "harder" problems in NP. If a polynomial-time algorithm could be found to solve the provably "hardest" problem in NP, then we would have a proof for 3).

2.3.2 Reductions

Q: *How many mathematicians does it take to screw in a light bulb?*

A: *One. The mathematician gives the light bulb to a Republican president, thereby reducing the joke to one already known to have a funny punchline.*

A reduction is a method of converting one problem into another problem such that a solution to the second problem yields a solution to the first. A reduction preserves the existence of a solution. If A reduces to B and B has a solution, then so does A; however, the existence of a reduction does not imply the existence of a solution.

If the reduction can be implemented in polynomial time, then if A reduces to B and solutions to B can be found ($B \in P$) or verified ($B \in NP$) in polynomial time, then so can solutions to A.

Definition 2.7. *A function $f\colon \Sigma^* \to \Sigma^*$ is a **polynomial-time computable function** if there exists a polynomial-time DTM M such that*

$$\forall x \in \Sigma^*\colon M(x) = f(x).$$

Note that since M can only write one symbol per step,

$$\forall x \in \Sigma^*\colon |f(x)| \leq t_M(x) + |x|.$$

Definition 2.8. *Let Π_1 and Π_2 be two languages. We say that Π_1 is **polynomial-time reducible** to Π_2, written $\Pi_1 \leq_P \Pi_2$, if there exists a polynomial-time computable function $f\colon \Sigma^* \to \Sigma^*$ such that*

$$\forall x \in \Sigma^*\colon x \in \Pi_1 \iff f(x) \in \Pi_2.$$

*The function f is called a **polynomial-time reduction** of Π_1 to Π_2.*

If we can construct a polynomial-time reduction from a problem A to a problem B, we can use the time-complexity of B to make determinations about the time-complexity of A. More specifically, reductions provide us a tool to prove that a problem is in P.

Lemma 2.2. *If Π_1 is polynomial-time reducible to Π_2, then*

$$\Pi_2 \in P \implies \Pi_1 \in P.$$

Proof. Let f denote the reduction of Π_1 to Π_2. Assume that $f(x)$ can be computed in time bounded by a polynomial p_1, and that Π_2 can be recognized by a DTM in time bounded by a polynomial p_2.

Given a string $x \in \Pi_1$ of length n, we can compute the string $f(x)$ in time bounded by $p_1(n)$. If $f(x) \in \Pi_1$, then $x \in \Pi_2$. Since $|f(x)| \leq p_1(n)$, a DTM can test if $f(x) \in \Pi_1$ in time $p_2(|f(x)|) \leq p_2(p_1(n))$. Since the total time to test if $x \in \Pi_2$ for any x is bounded by a polynomial $p_1(n) + p_2(p_1(n))$, it follows that $\Pi_1 \in P$. □

We introduce one more complexity class of problems. A problem $\Pi \in NP$ is NP-complete if every problem in NP has a polynomial-time reduction to Π.

Definition 2.9. *A problem $\Pi \in NP$ is **complete for NP** or **NP-complete** if*

$$\forall \Pi' \in NP \colon \Pi' \leq_P \Pi.$$

Theorem 2.3. *If a problem Π is NP-complete and $\Pi \in P$, then $P = NP$.*

Proof. Let Π' be a problem in NP. Then by the definition of NP-completeness, Π' is reducible to Π. Since $\Pi \in P$, it follows immediately from Lemma 2.2 that $\Pi' \in P$ and $P = NP$. □

Lemma 2.4. *If $\Pi_3 \leq_P \Pi_2$ and $\Pi_2 \leq_P \Pi_1$, then $\Pi_3 \leq_P \Pi_1$.*

Proof. Let f be the polynomial-time computable function that yields the reduction $\Pi_3 \leq_P \Pi_2$. Let g be the polynomial-time computable function that yields the reduction $\Pi_2 \leq_P \Pi_1$. Note that the composition $g \circ f$ can also be computed in polynomial time since $|g(f(x))|$ is polynomial in $|f(x)|$ and $|f(x)|$ is polynomial in $|x|$. Since

$$x \in \Pi_3 \iff f(x) \in \Pi_2 \iff g(f(x)) \in \Pi_1,$$

it follows that $\Pi_3 \leq_P \Pi_1$. □

Theorem 2.5. *A problem Π_1 is NP-complete if*

1. *$\Pi_1 \in NP$ and*
2. *there exists an NP-complete problem Π_2 such that $\Pi_2 \leq_P \Pi_1$.*

Proof. Let Π_3 be a problem in NP. Then $\Pi_3 \leq_P \Pi_2$. Since $\Pi_2 \leq_P \Pi_1$, by Lemma 2.4 we can compose the two reductions such that $\Pi_3 \leq_P \Pi_1$. Since Π_3 could be any problem in NP, it follows that all problems in NP are reducible to Π_1 and that, by definition, Π_1 is complete for NP. □

Chapter 3

Boolean Logic

3.1 All roads lead to SAT

3.1.1 Boolean algebra

Boolean logic is a foundation. Upon it rests, or teeters, an empire of academia and corporate wealth.

For the logician, Boolean algebra allows the construction and deduction of *valid arguments* — the inferred logical result of a set of axioms and *well-formed formulas* that is collectively called the *propositional calculus*.

For the mathematician, the formalization of mathematical statements relies on Boolean algebra as the cornerstone of the *predicate calculus* of first-order logic.

For the computer scientist, Boolean logic provides the nuts and bolts of digital computation and circuit design. The music of modern computing is a symphony of zillions of Boolean calculations.

For the complexity theorist interested in proving something interesting about puzzles, Boolean logic is a means to an end. Before we can construct a reduction from one NP-complete problem to another problem, we need a starting point. We need at least one NP-complete problem. Elementary questions in Boolean logic quickly lead us to a problem called SAT — the starting point from (and to) which virtually all other NP-complete problems are reduced.

Each Boolean variable can be assigned only one of the two Boolean values **true** and **false**. We write T for *true*, and F for *false*. True/false, on/off, up/down, Boolean logic captures the essence of anything with two mutually exclusive states. When thinking of sets of Boolean variables, we might imagine a very long panel of two-position light switches (no dimmers, please).

Similar to the way we build arithmetic expressions such as

$$3 \div (5 - 2) \times 4,$$

we use combinations of Boolean variables and **Boolean connectives** such as $\neg$ (NOT), $\wedge$ (AND), and $\vee$ (OR) to build Boolean expressions. Expressions such as

$$x_1 \wedge \neg(x_2 \vee \neg x_1)$$

can be constructed using a recursive definition of what we consider valid Boolean expressions.

Definition 3.1. *A **Boolean expression** over $X \cup \{\neg, \vee, \wedge, (,)\}$ of variables in $X = \{x_1, x_2, \ldots\}$ is*

- *a constant T or F,*
- *a variable $x \in X$,*
- *an expression of the form $\neg\phi$,*
- *an expression of the form $\phi \wedge \phi'$,*
- *an expression of the form $\phi \vee \phi'$, or*
- *an expression of the form (ϕ)*

*where ϕ and ϕ' are Boolean expressions. A Boolean expression of the form $\neg x$ or x, where x is a Boolean variable, is called a **literal**. In addition, if ϕ and ϕ' are Boolean expressions,*

- *$\phi \wedge \phi'$ is called the **conjunction** of ϕ and ϕ',*
- *$\phi \vee \phi'$ is called the **disjunction** of ϕ and ϕ', and*
- *$\neg\phi$ is called the **negation** of ϕ.*

*The disjunction or conjunction of literals is called a **clause** over X. Let us denote the set of Boolean expressions with Φ.*

We can manipulate sets of Boolean variables by applying **Boolean operations** to Boolean expressions. We define the semantics of each form of Boolean expression as follows.

Definition 3.2. *A **truth assignment** τ is a mapping from X to the set of **truth values** $\{T, F\}$.*

Definition 3.3. ***Negation** or **logical NOT** is a unary Boolean operation*

$$\neg\colon \{T, F\} \to \{T, F\}$$

where

$$\neg(x) = \begin{cases} T & : \quad x = F \\ F & : \quad \text{otherwise.} \end{cases}$$

Definition 3.4. ***Conjunction** or **logical AND** is a binary operation*

$$\wedge\colon \{T, F\} \times \{T, F\} \to \{T, F\}$$

where

$$\wedge(x, y) = \begin{cases} T & : \quad x = T, y = T \\ F & : \quad \text{otherwise.} \end{cases}$$

Definition 3.5. ***Disjunction*** *or* ***logical OR*** *is a binary operation*

$$\vee\colon \{T,F\}\times\{T,F\}\to\{T,F\}$$

where

$$\vee(x,y)=\begin{cases} F & : \quad x=F, y=F \\ T & : \quad \textit{otherwise.}\end{cases}$$

We can now determine the truth value of compositions of Boolean operations such as

$$\vee(T,F), \neg(F), \text{ and } \wedge(F,\vee(\neg F,T)),$$

but we need some additional concepts to describe all of the possible truth values associated with a statement like

$$\neg x_3\wedge(x_1\vee(x_2\wedge\neg x_3)).$$

We associate a truth value with a Boolean expression under some truth assignment according to the following definition of a *Boolean value* of an expression.

Definition 3.6. *Given a truth assignment τ, the* ***Boolean value*** *of a Boolean expression ϕ is defined by a mapping*

$$\nu_\tau\colon \Phi\to\{T,F\}$$

such that, if ϕ is a Boolean expression, then

$$\nu_\tau(\phi)=\begin{cases} \phi & : \quad \phi\in\{T,F\} \\ \tau(\phi) & : \quad \phi\in X \\ \nu_\tau(\phi') & : \quad \phi=(\phi') \\ \neg(\nu_\tau(\phi')) & : \quad \phi=\neg\phi' \\ \vee(\nu_\tau(\phi_1),\nu_\tau(\phi_2)) & : \quad \phi=\phi_1\vee\phi_2 \\ \wedge(\nu_\tau(\phi_1),\nu_\tau(\phi_2)) & : \quad \phi=\phi_1\wedge\phi_2\end{cases}$$

Definition 3.7. *A Boolean expression $\phi\in\Phi$ is* ***satisfiable*** *if there exists a truth assignment τ such that*

$$\nu_\tau(\phi)=T.$$

We say then that τ ***satisfies*** *ϕ.*

We now pose the problem of satisfiability for a Boolean expression.

Definition 3.8. (SAT)
Given a Boolean expression $\phi\in\Phi$, is ϕ satisfiable?

Consider as an example the Boolean expression

$$x_1\wedge\neg(x_2\vee\neg x_1).$$

Can we assign truth values for x_1 and x_2 such that the expression is satisfied? If we assign T to x_1 and F to x_2 (we'll call this assignment τ), then

$$\begin{aligned}
\nu_\tau(x_1 \wedge \neg(x_2 \vee \neg x_1)) &= \wedge(\nu_\tau(x_1), \nu_\tau(\neg(x_2 \vee \neg x_1))) \\
&= \wedge(\tau(x_1), \neg(\nu_\tau(x_2 \vee \neg x_1))) \\
&= \wedge(T, \neg(\vee(\nu_\tau(x_2), \nu_\tau(\neg x_1)))) \\
&= \wedge(T, \neg(\vee(\tau(x_2), \neg(\nu_\tau(x_1))))) \\
&= \wedge(T, \neg(\vee(F, \neg(\tau(x_1))))) \\
&= \wedge(T, \neg(\vee(F, \neg(T)))) \\
&= \wedge(T, \neg(\vee(F, F))) \\
&= \wedge(T, \neg(F)) \\
&= \wedge(T, T) \\
&= T.
\end{aligned}$$

We could compile a **truth table** of truth values for $\nu_\tau(x_1 \wedge \neg(x_2 \vee \neg x_1))$ under every possible truth assignment τ. If we were to check every value of ν_τ for our Boolean expression, we would get the following table:

x_1	x_2	$\nu_\tau(x_1 \wedge \neg(x_2 \vee \neg x_1))$
T	T	F
T	F	T
F	T	F
F	F	F

Note that only one truth assignment satisfies our Boolean expression. That is to say, our Boolean expression is **uniquely satisfiable**.

Finding a satisfying truth assignment for our Boolean expression wasn't very difficult, but then again, we only had to check truth values for two Boolean values. What if our Boolean expression had, say, 10,000 variables and 30,000 literals? Obviously, a Boolean expression of that size would take us a while longer; however, a computer can perform 30,000 operations in less than a second. Before we begin typing up an algorithm, the following theorem should give us pause for thought.

Lemma 3.1. *$SAT \in NP$.*

Proof. Given a Boolean expression $\phi \in \Phi$ of length n, nondeterministically guess the satisfying truth assignment τ and write it as a hint on the tape. Since $\nu_\tau(\phi) = O(p(|\phi|))$, we can verify the solution in polynomial time by evaluating $\nu_\tau(\phi)$. □

Theorem 3.2. (Cook's Theorem)
SAT is NP-complete.

As SAT truly is a means to our end, the interested reader is directed elsewhere for a more rigorous treatment of Cook's theorem. A proof can be found in Cook's original paper on complexity[3] or in [9]. A much more thorough investigation of problems in symbolic logic and an interesting alternate proof of Cook's theorem can be found in [14].

3.2 Boolean circuits

Cook's theorem allows us to use Theorem 2.5 to prove new NP-completeness results. In practice, virtually all reduction proofs of NP-completeness involve a chain of reductions that lead back to SAT. SAT provides us with an algebraic notion of Boolean logic; however, most of the puzzles we're likely interested in involve some discretized plane, like a grid or some other tiling. Our reductions are simplified by introducing a graph-theoretic representation of SAT using Boolean circuits. First, we'll need some new notation.

3.2.1 Some tools from graph theory

Definition 3.9. *A **directed graph** or **digraph** G is a pair (V, E) of a non-empty alphabet V of **nodes** or **vertices** and a finite family $E \subseteq V \times V$ of **directed edges**. If $e = (v_1, v_2)$ is an ordered pair in the **edge set** E, then the edge e is said to **join** two vertices v_1 and v_2 in the **vertex set** V. We call the first and second elements of the ordered pair the **tail** and **head**, respectively, of the directed edge. Thus, e is a directed edge **from** v_1 **to** v_2.*

Instead of using pairs to denote edges, we can also write edges as strings of length 2 over V such as $e = v_1v_2$.

Definition 3.10. *A digraph is called **simple** if*

$$\forall (v_1, v_2) \in E \colon v_1 \neq v_2.$$

Let us consider some examples. Fig. 3.1 represents the simple directed graph $G = (V, E)$ consisting of a vertex set

$$V = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}\}$$

and an edge set

$$E = \{(\mathtt{a}, \mathtt{d}), (\mathtt{e}, \mathtt{d}), (\mathtt{c}, \mathtt{e}), (\mathtt{c}, \mathtt{d}), (\mathtt{b}, \mathtt{c}), (\mathtt{d}, \mathtt{b})\} = \{\mathtt{ad}, \mathtt{ed}, \mathtt{ce}, \mathtt{cd}, \mathtt{bc}, \mathtt{db}\}.$$

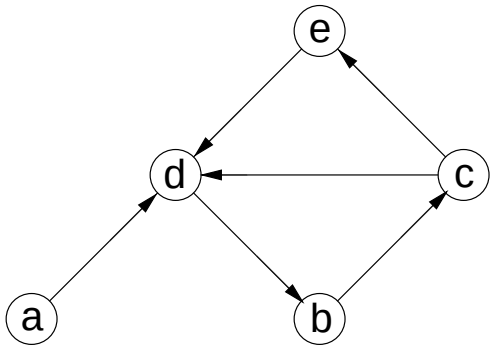


Figure 3.1: An example of a simple directed graph

Definition 3.11. *If G is a digraph, then a **directed path** on G is any sequence $P = (v_0, v_1, \ldots, v_n)$ where the following holds:*

1. *With the possible exception of $v_0 = v_n$, each $v_i \in P$ is unique.*
2. *Every ordered pair (v_i, v_{i+1}) of vertices in P is a directed edge in E.*

*If $v_0 = v_n$, then the directed path is called a **directed cycle**. A digraph is **acyclic** if it contains no directed cycles.*

Consider a map of downtown Portland. Quite frustratingly, virtually every street is one-way, allowing traffic to flow in only one direction. If we represent intersections as vertices and every stretch of one-way street with a directed edge, we have a digraph. A directed path in our digraph corresponds to a (legal) driving route downtown. If our driving route corresponds to a directed cycle in our digraph, then we'll find ourselves driving in circles!

Definition 3.12. *Given a digraph $G = (V, E)$, the **in-degree** of a vertex $v \in V$ is the number of edges in E with v as a head. That is to say,*

$$\text{indegree}(v) = |\{v' \in V \mid (v', v) \in E\}|.$$

*Similarly, we define the **out-degree** of v as the number of edges in E with v as a tail, that is*

$$\text{outdegree}(v) = |\{v' \in V \mid (v, v') \in E\}|.$$

We can now introduce the definition of a Boolean circuit.

Definition 3.13. *A **Boolean circuit** is an acyclic directed graph (V, E) where*

$$V = \{v_{\mathcal{O}}\} \cup \textsc{Input} \cup \textsc{Or} \cup \textsc{And} \cup \textsc{Not} \cup \textsc{Branch}$$

defined as follows:

1. $\textsc{Input} \neq \emptyset$ *is a collection of **input vertices**, labeled with variables. For every $v_{\mathcal{I}} \in \textsc{Input}$,* $\text{indegree}(v_{\mathcal{I}}) = 0$ *and* $\text{outdegree}(v_{\mathcal{I}}) = 1$
2. *$v_{\mathcal{O}}$ is the **output vertex**, labeled with an **O**, where*
   (a) $\text{indegree}(v_{\mathcal{O}}) = 1$,
   (b) $\text{outdegree}(v_{\mathcal{O}}) = 0$, *and*
   (c) $\forall v \in V$: *there exists a path from v to $v_{\mathcal{O}}$.*
3. $\textsc{And}$ *is a collection of vertices, labeled with $\wedge$, called **AND gates** such that for every $v_\wedge \in \textsc{And}$,* $\text{indegree}(v_\wedge) = 2$ *and* $\text{outdegree}(v_\wedge) = 1$.
4. $\textsc{Or}$ *is a collection of vertices, labeled with $\vee$, called **OR gates** such that for every $v_\vee \in \textsc{Or}$,* $\text{indegree}(v_\vee) = 2$ *and* $\text{outdegree}(v_\vee) = 1$.

5. NOT *is a collection of vertices, labeled with* $\neg$*, called* ***NOT gates*** *or* ***inverters*** *such that for every* $v_\neg \in$ NOT, $\text{indegree}(v_\neg) = 1$ *and* $\text{outdegree}(v_\neg) = 1$.

6. BRANCH *is a collection of unlabeled vertices called* ***branch gates*** *or* ***splitters*** *such that for every* $v \in$ BRANCH, $\text{indegree}(v) = 1$ *and* $\text{outdegree}(v) = 2$.

7. $\big\{\{v_{\mathcal{O}}\}, \text{INPUT}, \text{OR}, \text{AND}, \text{NOT}, \text{BRANCH}\big\}$ *is a partition of* V.

We say that the edges are the ***wires*** *and the vertices in* OR$\cup$AND$\cup$NOT$\cup$BRANCH *are the* ***gates*** *of our circuit* G.

We say that a Boolean circuit is ***satisfied*** *if a truth assignment exists that satisfies the corresponding Boolean expression for* G.

Given any Boolean expression ϕ, we can convert it into a Boolean circuit. For example, if $\phi = x_1 \wedge \neg(x_2 \vee \neg x_1)$, then we can construct the corresponding Boolean circuit in Fig. 3.2.

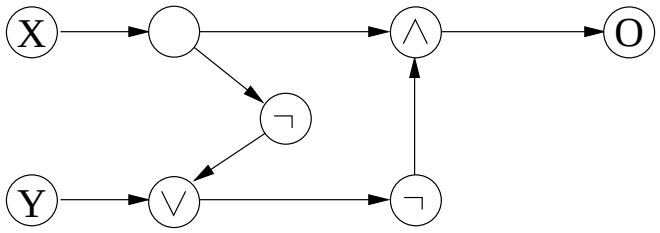


Figure 3.2: A Boolean circuit corresponding to $x_1 \wedge \neg(x_2 \vee \neg x_1)$

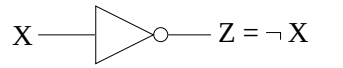
3.2.2 Circuit construction

Our Boolean circuits closely adhere to the notation of graph theory; however, the standard notation for circuit design deviates somewhat from our current set of symbols. To conform with the circuit designers of the world, we make the following modifications to our notation:

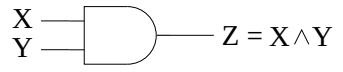
- Wires are represented as sequences of straight lines instead of arrows. Each gate has an **incident** side associated with its symbol that preserves the orientation of the wire.
- Branch gates are represented by black dots.
- The remaining gates are represented in Fig. 3.3.

Fig. 3.4 illustrates the Boolean circuit for $x_1 \wedge \neg(x_2 \vee \neg x_1)$ using the notation of digital circuit design.

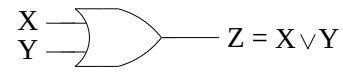
We can now create quite varied and complicated circuits using only the basic gates AND, OR, NOT, and BRANCH. Indeed, if we allow input vertices to represent digits of binaries numbers, we can even answer Yes or No questions about



X	$Z = \neg X$
T	F
F	T



X	Y	$Z = X \wedge Y$
T	T	T
T	F	F
F	T	F
F	F	F



X	Y	$Z = X \vee Y$
T	T	T
T	F	T
F	T	T
F	F	F

Figure 3.3: Graph symbol and truth table for a NOT gate, an AND gate, and an OR gate.

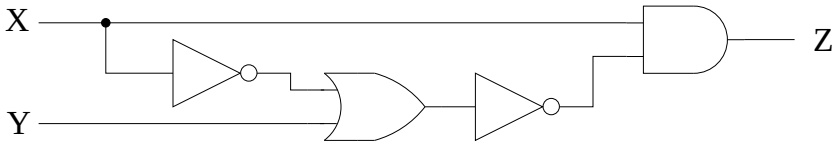
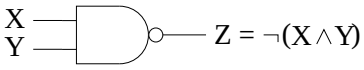


Figure 3.4: Conventional notation for a Boolean circuit corresponding to the expression $x_1 \wedge \neg(x_2 \vee \neg x_1)$

integer arithmetic. If we were to modify our definition of a Boolean circuit to allow more than one output vertex, we could actually perform addition, subtraction, multiplication, and other operations of integers.

We will restrict ourselves to a few simpler constructions, starting with a component called a NAND gate, where NAND stands for "NOT-AND". As the name suggests, a NAND gate is simply an AND gate followed by an inverter and corresponds to the Boolean expression $\neg(x_1 \wedge x_2)$.



X	Y	$Z = \neg(X \wedge Y)$
T	T	F
T	F	T
F	T	T
F	F	T

Figure 3.5: Graph symbol and truth table for a NAND gate

A NAND gate is somewhat special. Any circuit that can be constructed using NOT, AND, and OR gates can also be modeled using only NAND gates instead. We need only to consider the constructions of AND, OR, and NOT gates using only NAND gates in Fig. 3.6 to see that this can be done.

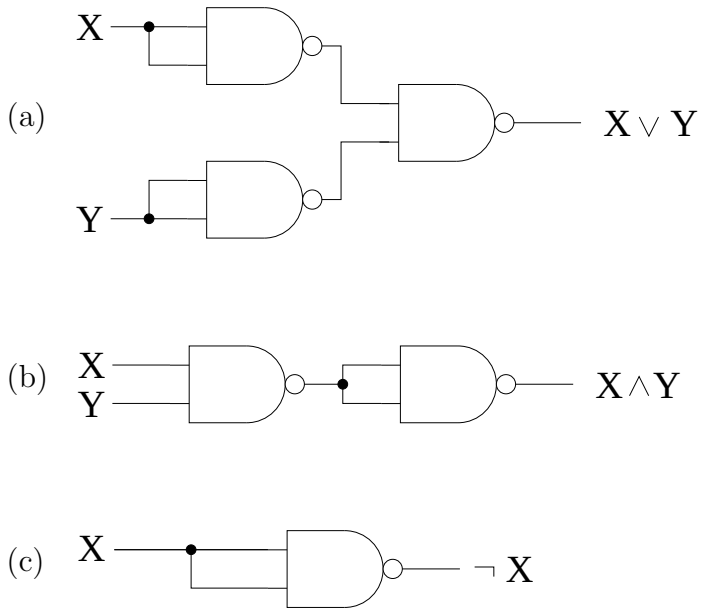
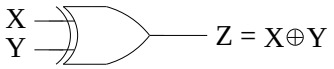


Figure 3.6: Constructions with NAND gates to model (a) an OR gate, (b) an AND gate, and (c) a NOT gate

We introduce another component that will prove useful for reductions in the following chapters. Disjunction, implemented with an OR gate, captures a notion of **inclusive OR**: $x_1 \vee x_2 = T$ even when both $x_1 = T$ and $x_2 = T$. With **Exclusive OR**, represented with symbol $\oplus$, we assert that $x_1 \oplus x_2 = T$ if and only if $x_1 = T$ and $x_2 = F$ or $x_1 = F$ and $x_2 = T$. That is to say

$$x_1 \oplus x_2 \equiv (x_1 \vee x_2) \wedge \neg(x_1 \wedge x_2) \equiv (x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2).$$

Fig. 3.7 defines a new gate, called an **XOR gate**, that corresponds to the Boolean operation $\oplus$.



X	Y	$Z = X \oplus Y$
T	T	F
T	F	T
F	T	T
F	F	F

Figure 3.7: Graph symbol and truth table for an XOR gate

Like our other gates, an XOR gate can be constructed using only NAND gates. Fig. 3.8 presents the construction.

Definition 3.14. (Circuit SAT)
Given a Boolean circuit $G = (V, E)$, is G satisfiable?

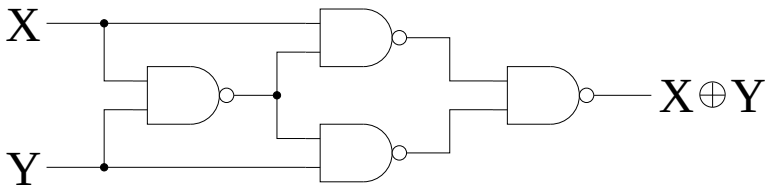


Figure 3.8: Construction of an XOR gate from four NAND gates

3.2.3 Planar circuits

As a last component, we consider the Boolean circuits restricted to a plane. Occasionally, a wire may need to cross the path of another wire to reach a gate. Until now, we have no explicit method or gate for allowing wires to cross without interference. As it turns out, we can construct a wire crossing using three XOR gates, as in Fig. 3.9.

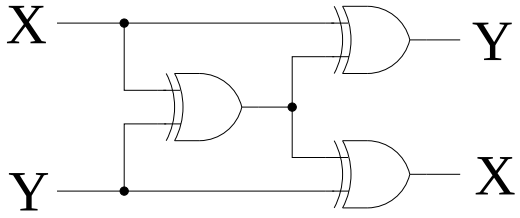


Figure 3.9: Construction of a wire crossing from three XOR gates

Definition 3.15. *A digraph is* ***planar*** *if it can be embedded in a plane without crossing edges.*

Fig. 3.10 presents an example of a non-planar graph. No matter how we draw the edges between the vertices, every embedding in the plane requires at least one wire crossing.

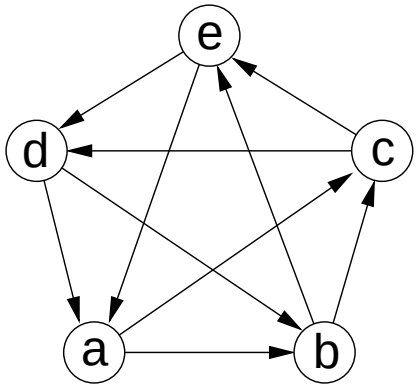


Figure 3.10: An example of a non-planar digraph

We now have all of the tools necessary to begin constructing reductions of SAT. In the chapters that follow, we consider two NP-complete puzzles called Nurikabe and

Minesweeper. Kaye[11] proved that Minesweeper is NP-complete by demonstrating the existence of a polynomial-time reduction of Circuit SAT to Minesweeper. We pursue the same approach to prove that Nurikabe is NP-complete.

Chapter 4

Nurikabe is NP-complete

Nurikabe is one of many popular *pencil-and-paper* puzzles originating in Japan. Most, if not all, paper-and-pencil puzzles consist of a grid of cells, a list of rules, and some initially specified cells. The objective is to complete the puzzle by shading in sections of the grid in compliance with the rules. Yato[22] provides the following list of popular paper-and-pencil puzzles:

- Nurikabe
- Nonogram[18] (or Paint-by-Numbers) *
- Slither Link[21] *
- Cross Sum[16] (or *Kakkuro*) *
- Number Place[16] (or *Sudoku*) *
- Heyawake

Those marked with a * are known to be NP-complete. According to Ueda and Nagao[18], Nonogram was the first paper-and-pencil puzzle to be proved NP-complete, although various NP-completeness results have been found for paper-and-pencil puzzles since then[22]. By constructing a polynomial-time reduction from Circuit-SAT to Nurikabe, we present here a new NP-completeness result for paper-and-pencil puzzles.

4.1 Introduction to Nurikabe

The goal of a Nurikabe puzzle is to shade in cells in the grid until all of the following conditions are met:

1. A cell containing a number cannot be shaded in.
2. Two cells are **touching** if they are directly above, below, to the left, or to the right of each other. Two cells are **contiguous** if they are connected by a sequence of touching cells.

3. Each numbered cell describes the number of contiguous white cells in which it is contained. Each area of white cells must contain exactly one number.

4. There cannot be any 2-by-2 blocks of shaded cells.

5. All of the shaded cells must be contiguous.

Deducing which cells must be shaded often requires determining which cells must be white. Fig. 4.1 presents an instance of Nurikabe and its solution. Typically, a good Nurikabe puzzle, like all good pencil-and-paper puzzles, has a unique solution; however, we do not include this property in our proof of NP-completeness. While the rules for Nurikabe may sound at first a bit complicated, there are some immediate strategies that result:

- If a cell contains a `1`, shade in its four neighbors.
- If a numbered cell is diagonally adjacent to another numbered cell, shade in the two cells that touch both numbered cells.
- If two numbered cells lie in the same row or column and are separated by only a single white cell, shade in that cell.

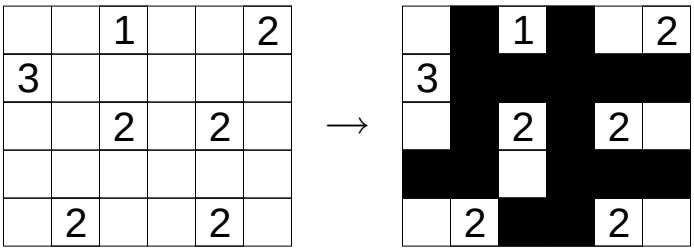


Figure 4.1: An instance of Nurikabe and its corresponding solution

Additional Nurikabe puzzles are provided in the appendix (without solutions). At this point, the interested reader is encouraged to practice solving a puzzle or two before proceeding. An intuitive understanding of the constraints of the puzzle will greatly assist an appreciation and understanding of the remainder of this chapter — and they're fun.

4.2 Formal definition

Now that we're somewhat familiar with the puzzle Nurikabe, let's introduce some formal notation before we define the decision problem associated with the puzzle. We will revisit some of this notation again in the next chapter when we introduce the puzzle Minesweeper.

We are interested in *grid-based* puzzles, so it seems appropriate to begin with the definition of a grid.

Definition 4.1. *A **grid** G is an $n \times m$ matrix M of symbols over some alphabet Σ. We say that each element of M is a **cell** in G and write x_{ij} to denote the cell in the i-th row and j-th column.*

We want to describe the connectivity of our grid of cells. In Nurikabe, for example, we define *contiguity* in terms of those cells directly above, below, to the right, and to the left of a given cell. In other puzzles, such as Minesweeper, a relation is established between each cell in the grid and all 8 of its neighbors.

Definition 4.2. *Given an $n \times m$ grid G, we say that two cells x_{ij} and x_{hk} in G are **directly adjacent** if*

$$|i - h| + |j - k| = 1.$$

*Likewise, we say that x_{ij} and x_{hk} are **diagonally adjacent** if*

$$|i - h| = |j - k| = 1.$$

*Two cells in G are simply **adjacent** if they are directly or diagonally adjacent.*

Definition 4.3. *If x is a cell in G, we define the **4-neighborhood** of x as the set of cells directly adjacent to x and write*

$$\aleph_4(x) = \{x' |\ x \text{ and } x' \text{ are directly adjacent}\}.$$

*We define the **8-neighborhood**[1] of x as the set of cells adjacent to x and write*

$$\aleph_8(x) = \{x' |\ x \text{ and } x' \text{ are adjacent}\}.$$

We can use our definitions of adjacency to precisely define our notions of contiguous cells and 2-by-2 blocks of cells.

Definition 4.4. *If S is a subset of cells in a grid G, then we say that two cells x and x' in S are **connected** if there exists a set $S' \subseteq S$ where*

$$S' = \{x =: x_0, x_1, x_2, \ldots, x_{k-1}, x_k := x'\}$$

such that

$$x_i \text{ and } x_{i+1} \text{ are directly adjacent for } 0 \leq i \leq k - 1.$$

*We say that S is **contiguous** if for every $x, x' \in S$*

$$x \neq x' \implies x \text{ and } x' \text{ are connected.}$$

Definition 4.5. *A subset S of cells in a grid G is a **2-by-2 block** if $|S| = 4$ and*

$$\forall x, x' \in S\colon\ x \text{ and } x' \text{ are adjacent}\ .$$

[1] This is also known as the **Moore's neighborhood** in the study of cellular automata.

Definition 4.6. *An **instance of Nurikabe** is an $n \times m$ grid G of cells from the set*

$$\left\{\boxed{}, \boxed{1}, \boxed{2}, \ldots, \boxed{\text{k}}\right\}$$

with at least two cells $x \in G$ and $x' \in G \setminus \{x\}$ such that $x \neq \boxed{}$ and $x' \neq \boxed{}$.

Definition 4.7. (NURIKABE)
Given an $n \times m$ instance I of Nurikabe, does there exist a partition

$$\mathcal{P} = \{B, W_0, W_1, \ldots, W_p\}$$

of cells in I such that the following conditions are true?

1. *For every $X \in \mathcal{P}$, X is contiguous.*
2. *For every $x \in B$, $x = \boxed{}$.*
3. *B contains no 2-by-2 blocks.*
4. *For every $W_i \in \mathcal{P}$, let $k = |W_i|$. The following conditions hold:*
   - *There exists exactly one $x \in W_i$ such that $x \neq \boxed{}$.*
   - *If $x \in W_i$ and $x \neq \boxed{}$, then $x = \boxed{\text{k}}$.*
   - *If $x \in W_i$, then*
   $$\forall x' \in \aleph_4(x) \colon x' \in W_i \text{ or } x \in B.$$

In our definition of the decision problem, we partition cells into p sets, of which B corresponds to our shaded cells and each W_i corresponds to a contiguous set of white cells of which only one cell is numbered. Condition 1 ensures that our partitioned sets are contiguous. Condition 2 ensures that we shade no numbered cells, and condition 3 allows no 2-by-2 blocks of shaded cells in our solution. Condition 4 demands (i) that each contiguous block of white contain exactly one numbered cell, (ii) that the numbered cell equal the number of white cells connected to it, and (iii) that contiguous blocks of white be bounded on all sides by shaded cells.

Note that if a solution contains no shaded cells, then $B \in \mathcal{P}$ is an empty set. However, a partition of sets can contain no empty sets, so we have a problem with our problem. Instances that correspond to these types of solutions are, however, very simple. The must contain only a single numbered cell whose value equals the product $n \cdot m$. Avoiding these instances does not affect the complexity of the problem.

Lemma 4.1. *Nurikabe $\in$ NP.*

Proof. Let G be the $n \times m$ grid of an instance I of Nurikabe, and let $N := n \cdot m$ be length of the input.

Nondeterministically guess a partition $\mathcal{P} = \{B, W_0, W_1, \ldots, W_k\}$ of the cells in G. We can verify that $\mathcal{P}$ is a solution to I in polynomial time using the following deterministic algorithm:

First we define the following algorithm IsContiguous which checks in time $O(N^2)$ whether a set C is contiguous:

On input C:
    $A \leftarrow$ some $x \in C$
    for each x in A
        for each x' in C
            if $x' \in \aleph_4(x)$ then
                $A \leftarrow \{x'\} \cup A$
    if $|A| \neq |C|$ then
        REJECT
    ACCEPT

We now describe our algorithm IsSolution which verifies whether a partition $\mathcal{P}$ is a solution to a grid G.

On input G and $\mathcal{P} = \{B, W_0, W_1, \ldots, W_k\}$:
    for each X in $\mathcal{P}$
        if IsContiguous(X) = REJECT then
            REJECT
    for each x in B
        if $x \neq \square$ then
            REJECT
        for each x_1 in $\aleph_4(x)$
            for each x_2 in $\aleph_4(x)$
                for each x_3 in $\aleph_8(x)$
                    if $x_3 \in \aleph_4(x_1)$ and $x_3 \in \aleph_4(x_2)$ then
                        REJECT
    for each W_i in $\mathcal{P}$
        $A \leftarrow \emptyset$
        for each x in W_i
            if $x \neq \square$ then
                $A \leftarrow \{x\} \cup A$
        if $|A| \neq 1$ then
            REJECT
        $y \leftarrow$ the value of the numbered cell in A
        if $|W_i| \neq y$ then
            REJECT
        for each x in W_i
            for each x' in $\aleph_4(x)$
                if $x' \notin B$ and $x' \notin W_i$ then
                    REJECT
    ACCEPT

IsSolution checks each of the conditions in Def. 4.7 in order. Condition 1 runs

IsContiguous k times, so this step of the verification runs in $O(N^3)$. Condition 2 can be checked in linear time. Condition 3 can also be checked in at most linear time and requires k times a constant $4 \cdot 4 \cdot 8 \cdot (4+3) = 896$ steps to determine whether 2-by-2 blocks exist. Lastly, condition 4 is checked for each contiguous white block W_i in at most $O(k \cdot (N + 4 \cdot N)) = O(N^2)$ steps.

Hence, the solution is verifiable in time $O(N^3)$. □

4.3 Circuit-SAT reduction

Lemma 4.2. *There exists a polynomial-time reduction of Circuit-SAT to Nurikabe.*

We present a proof by construction, similar to the techniques used by Kaye[11], Friedman[6], and Moore and Robson[13]. Our goal is to model the properties of a Boolean circuit using only the rules of Nurikabe. Given a Boolean circuit $\mathcal{C}$, we will construct a corresponding circuit on a Nurikabe board in time polynomial in the number of inputs.

4.3.1 Initial setup

First, we construct a grid large enough to contain our circuit. As a sort of silicon and substrate, we tile our grid with 1s in a fashion such that each 1 is a knight's move away from neighboring 1s. Notice that this tiling immediately forces a solution with almost maximally connected shaded cells and no 2-by-2 blocks. We will place our circuit components on the tiled board in such a manner to avoid introducing 2-by-2 blocks while preserving the connectivity of the shaded cells. Fig. 4.2 illustrates an initial tiling for a 15-by-15 Nurikabe grid.

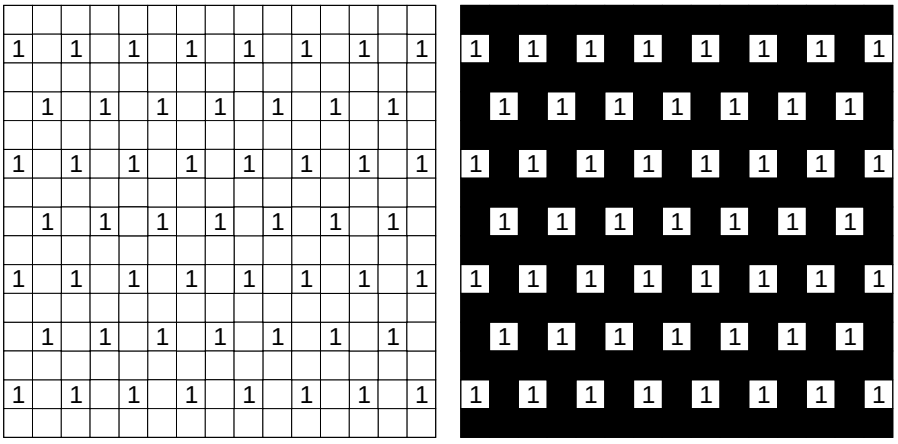


Figure 4.2: Initial tiling of a 15-by-15 Nurikabe grid with 1s (*left*) and its uniquely determined solution (*right*)

4.3.2 Wires

Our first task is to create a wire that has the ability to carry a signal from one gate to the next. Fig. 4.3 presents a construction of a wire in Nurikabe. The figure presents only a portion of the wire; the periodic tiling is intended to extend in both directions until it reaches a component of our circuit. Our wires will always extend diagonally across our Nurikabe board. As with all of the circuit components we will be introducing, the area away from the wire extends into our tiling of 1s. In our figures, we will give partially shaded Nurikabe grids. We use ■ and ○ to denote cells that must be black and white, respectively, in any solution of the instance depicted. The reader may find it an interesting exercise to verify that each ■ and ○ cell can be deduced from only the numbered cells.

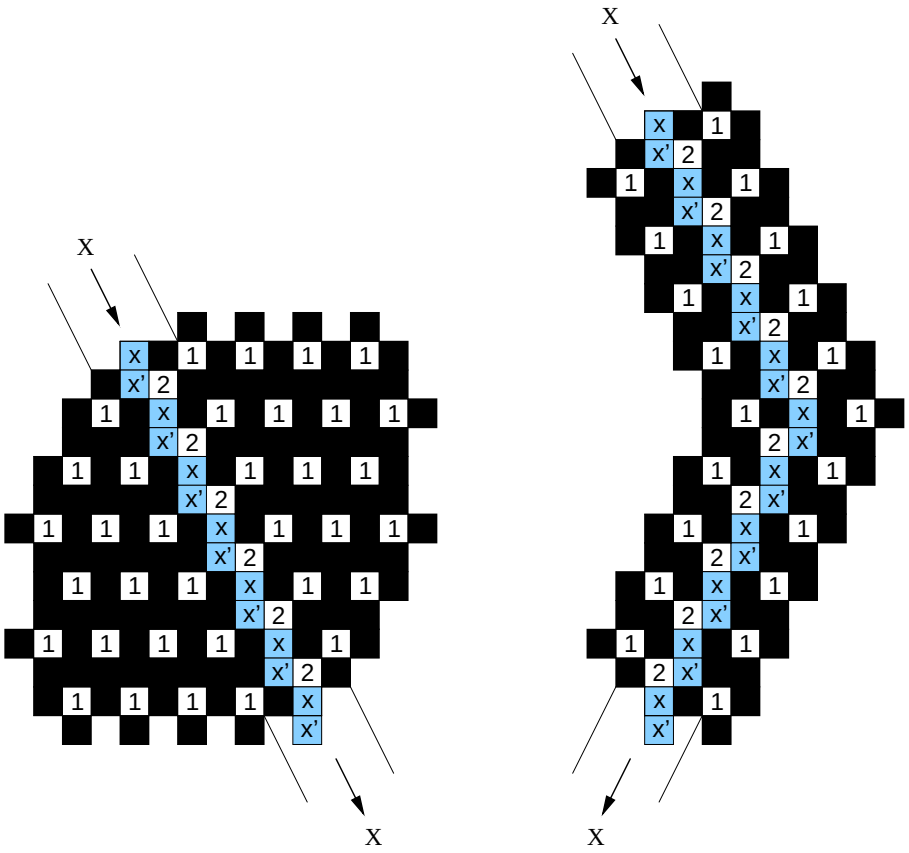


Figure 4.3: A Boolean wire in Nurikabe (*left*) and a component to allow bends in the wire (*right*)

In our wire tile in Fig. 4.3, we number the x's and x's starting with 1 at the top. If we shade in x_1, we can deduce the following:

$$x_1 = \blacksquare \implies x'_1 = \square \implies x_2 = \blacksquare \implies x'_2 = \square \ldots$$

Conversely, if we shade in x'_1, we can deduce that

$$x'_1 = \blacksquare \implies x_2 = \square \implies x'_2 = \blacksquare \implies x_3 = \square \ldots$$

Fig. 4.4 illustrates the two possible shadings for our wire tile. In order to distinguish **false** from **true**, we will associate a truth assignment with a shading of our wires. Notice that if [x] = ■, then the black-shaded cells on either side of the wire are connected by each [x]; however, if [x'] = ■, then the wire effectively becomes a barrier between the black-shaded areas on either side of the wire[2]. We define as **true** the state of a wire that connects shaded cells on either side and **false** the state of a wire that barricades these cells on each side of the wire.

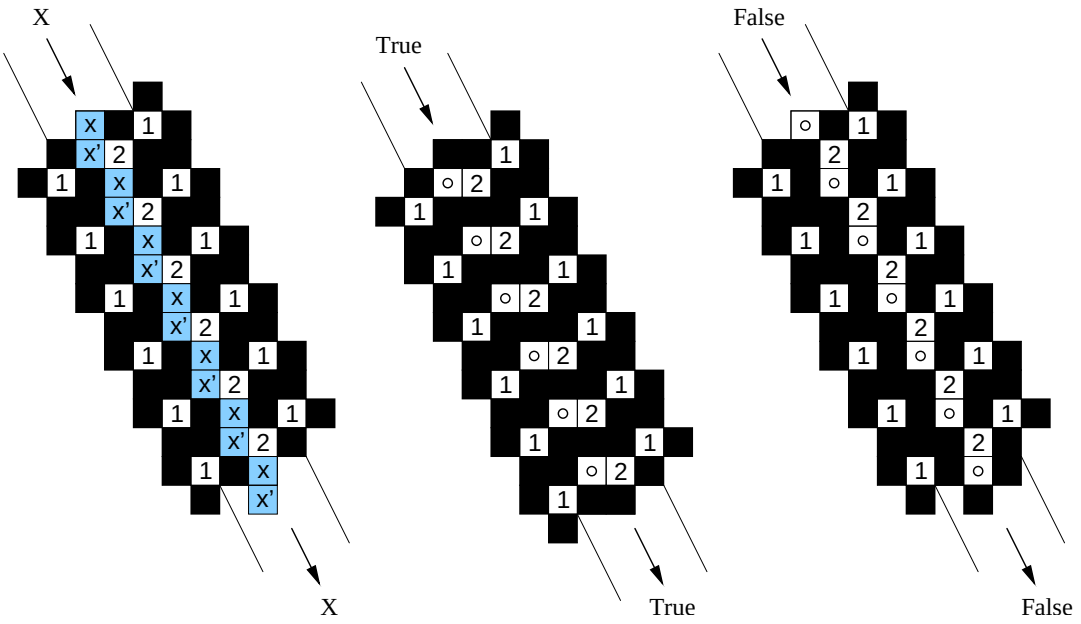


Figure 4.4: A Boolean wire (*left*) can be set to **true** (*middle*) or **false** (*right*)

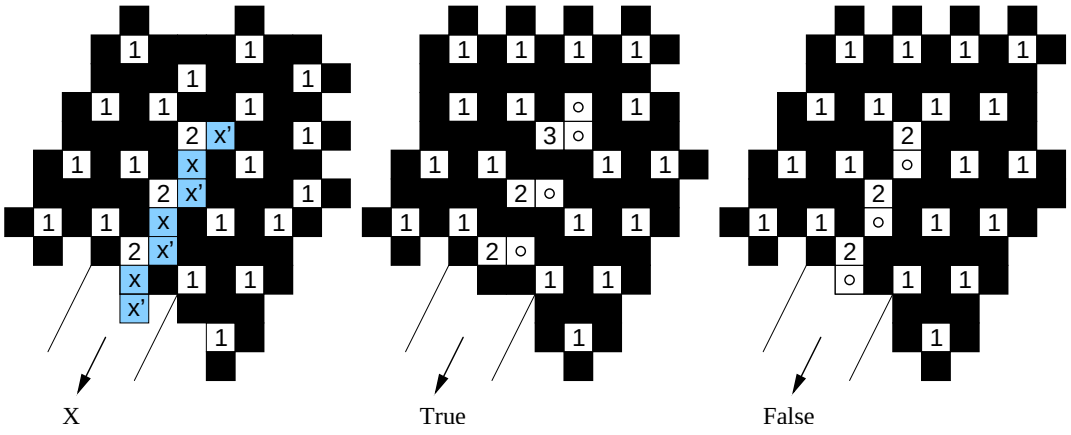


Figure 4.5: A variable terminal (*left*), a constant terminal that fixes a wire to **true** (*middle*), and one that fixes a wire to **false** (*right*)

In order for signals to propagate along wires we need to have starting points for each signal. We construct **variable terminals** for each input vertex in $\mathcal{C}$ as

[2]We will revisit this problem of connectivity across our circuit later.

illustrated in Fig. 4.5. The state of the wire that extends from the terminal is uniquely determined by whether x = ■ or x = □.

Conversely, a terminal can also be an **output** vertex, and we can observe the result of evaluating the Boolean expression that corresponds to $\mathcal{C}$ by examining the state of x in Fig. 4.5.

For our reduction, however, we are only interested in determining whether we can find truth assignments for our input vertices that result in a Boolean value of **true** for our output vertex. Thus, we must provide a way to fix the value of a wire to a constant truth value, in this case, **true**. Fig. 4.5 also presents two **constant terminals** that allow us to fix a value.

We should provide our wires with the ability to change direction, and Fig. 4.3 also presents such a component. While compositions of this component will not allow our wire to, in a sense, head back in the direction it came from, we should note that since a Boolean circuit can contain no cycles, such wire behavior is not necessary[3].

Fig. 4.6 presents us with our first logic gate — a NOT gate. If the incident wire is **true** (x = ■), then the outgoing wire carries the value **false** (x = □).

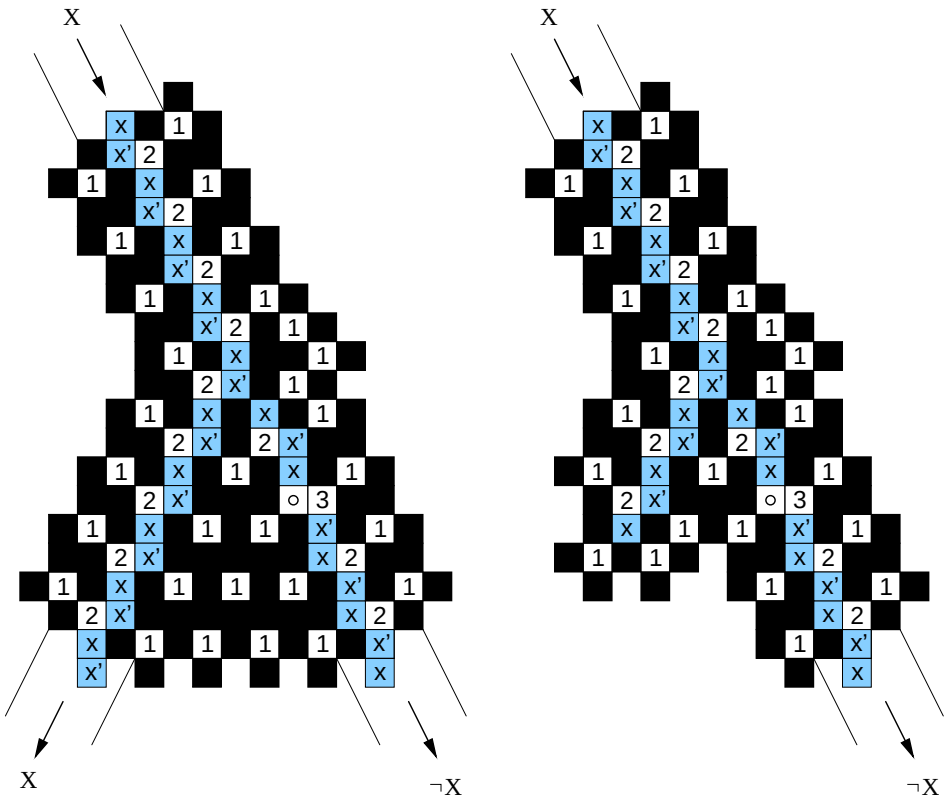


Figure 4.6: A branch gate/splitter (*left*) and a NOT gate/inverter (*right*)

The branch gate, also illustrated in Fig. 4.6, allows us to send a signal to more than one gate. Notice that the right outgoing wire of the branch gate inverts the

[3]Additional components that allow more freedom in the construction of Boolean circuits, at the cost of a somewhat more complicated proof, can be found in the appendix.

signal of the input wire. The NOT gate of Fig. 4.6 is a modified splitter where the left outgoing wire has been terminated with a variable terminal. If we add a NOT gate to the right outgoing wire in Fig. 4.6, we can ensure that the splitter outputs two signals corresponding to the signal carried by the input wire.

Fig. 4.7 presents our first fairly complicated gate.

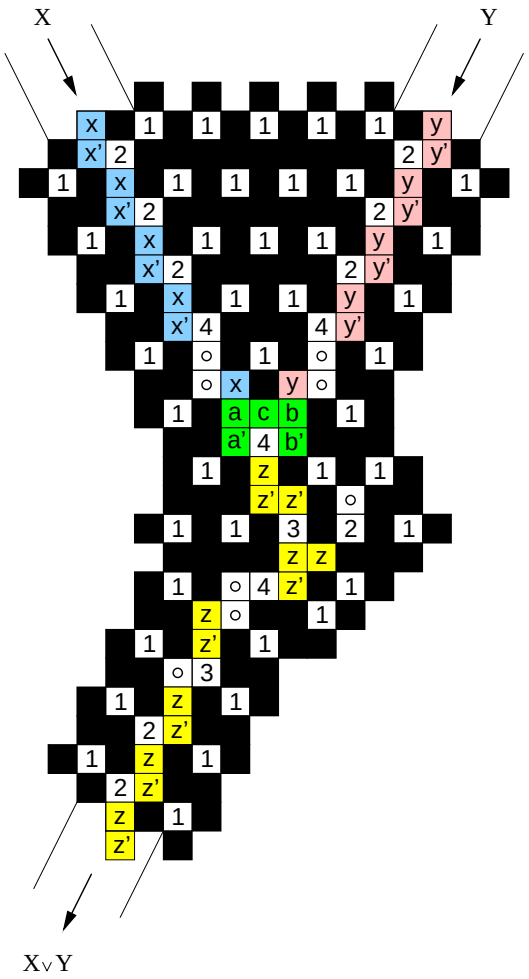


Figure 4.7: An OR gate

Claim 4.3. *Fig. 4.7 depicts an OR gate.*

Proof. Let X, Y, Z be Boolean variables corresponding to the cells of x, y, z, respectively. We will construct a truth table for X, Y, and Z by specifying ■ or □ values for x, y, and z.

- Suppose that $X = T$ and $Y = T$. The reader can verify that x = ■ and y = ■, hence x' = □ and y' = □ for all x, x', y, y' in our gate.

  Suppose now that c = ■. Then it follows that both a = □ and b = □. a and b must be connected by white cells to a numbered cell; otherwise, a 2-by-2 block will be formed by each. The 4 is the only numbered cell that

presents itself. It follows then that b' = □ and a' = □ in order to connect a and b to the 4. However, now the 4 resides in block of contiguous white containing at least five squares, which cannot be. It follows that c = □.

One of {a, a'} and one of {b, b'} must be white to prevent the appearance of 2-by-2 blocks in our puzzle. Since the 4 has already "committed" a white cell to c (and one to itself), only two white cells remain free. In any case, 4 contiguous white cells contain the 4, so we shade in all cells bordering that area, including the cell z directly under the 4. It follows that z = ■, and thus $Z = T$. The reader may verify that every z' = □ and every z = ■.

- Suppose now that $X = F$ and $Y = F$. As before, the reader can verify that x = □ and y = □, hence x' = ■ and y' = ■ for all x, x', y, y' in our gate.

  It immediately follows that a = ■ and b = ■ after we shade in the cells neighboring the two contiguous sets of 4 white cells lining the input wires. To prevent the appearance of 2-by-2 blocks, we can also deduce that a' = □ and b' = □. At this point, since both y = □ and x = □, the ■ cell above c will only be connected to the other black cells if c = ■.

  The central 4 still requires one more white cell as part of its contiguous cells. The only undetermined cell is z, so it follows that z = □ and $Z = F$. Again, the reader may wish to verify these details.

- Suppose now that $X = F$ and $Y = T$. Then x = □ and y = ■, hence x' = ■ and y' = □ for all x, x', y, y' in our gate. In addition, we know that a = ■ after shading in the bordering cells of the contiguous block of 4 cells in the left input wire. To avoid a 2-by-2 block, it follows that a' = □.

  Suppose now that c = ■. Then b = □ to prevent the appearance of a 2-by-2 block. b can only be contained in the contiguous block of the central 4 and for that to happen, it must also be the case that b' = □. The central 4 now resides in a contiguous block of 4 cells, so we can shade all cells directly adjacent to that block, including z. Hence, z = ■.

  Suppose instead that c = □. One of {b, b'} must be white to prevent the appearance of a 2-by-2 block. The central 4 only has one white cell left to commit, since both c = □ and a' = □. Either b = □ or b' = □, but in either case, shading the directly adjacent cells of the resulting contiguous set of 4 white cells will force z = ■.

  So in either case, z = ■ and $Z = T$. The case for $X = T$ and $Y = F$ is analogous.

Our argument by cases produces the following truth table:

X	Y	$Z = X \vee Y$
T	T	T
T	F	T
F	T	T
F	F	F

□

We can add NOT gates to the inputs or output of our OR gate to construct an AND gate and a NAND gate. For explicit constructions of these gates, please see the appendix. As we did in Chapter 3, we can then use our NAND gates to construct XOR gates and wire crossings. Those constructions, while quite straightforward, would be overly large to include as figures in the space here.

During our discussion of constructing Boolean components in Nurikabe, we skipped over a few details that should now be addressed.

Firstly, it is very important that all of our components actually line up. Imagine if we paint every other row in our Nurikabe grid pink. If an x cell lies in a pink row, then all x cells in a wire will lie in a pink row, since wires have an even period. We see that our wires may be out of **phase** if an x cell of one wire lies in a pink row and a y cell of another does not. As it turns out, all of our components have a period of 2, so lining things up isn't that difficult as long as we restrict ourselves to always place variable terminals such that the first x of each lies in the same color row as the others. Nevertheless, we can construct phase shifters to fix the problem should it ever occur. The components of Fig. 4.8 allow us to shift the phase of wires to ensure that everything lines up[4].

Secondly, another detail which requires attention is that of connectivity. The straightforward circuit representation of $\neg(x_1 \vee x_1) = T$, for example, would lead to two wires carrying the signal **false** to an OR gate. Unless we do something to avoid it, the black cells in the area contained by the two wires, the splitter, and the OR gate would not be connected to the rest of the black cells in the grid. As a remedy, we add the requirement that two NOT gates be placed along every piece of wire, allowing some spacing between the NOT gates and the two ends of the wire as well as between the two NOT gates themselves. If the wire is **false**, then the strip of wire between the two NOT gates is true and all black cells on the board remain connected. Likewise, all black cells are connected if the wire is **true**, since the strip between the two NOT gates does not occupy the entire length of the wire.

Lastly, we need to show that our circuit components can be integrated into the initial tiling of 1s and s without introducing any 2-by-2 blocks while still ensuring that all black cells are connected. In a sense, this can be expressed as an easy tiling problem. I give a sketch of a proof to convince the reader that this is not difficult to ensure.

Pick a bounding box B for the largest gate that we considered, the OR gate. Fill in the unspecified areas in the bounding box such that when two OR gates are placed next to each other or above one another, the tiling of 1s and s is

[4]However, if we decide to introduce some of the additional Nurikabe components from the appendix, then we will need these phase shifters.

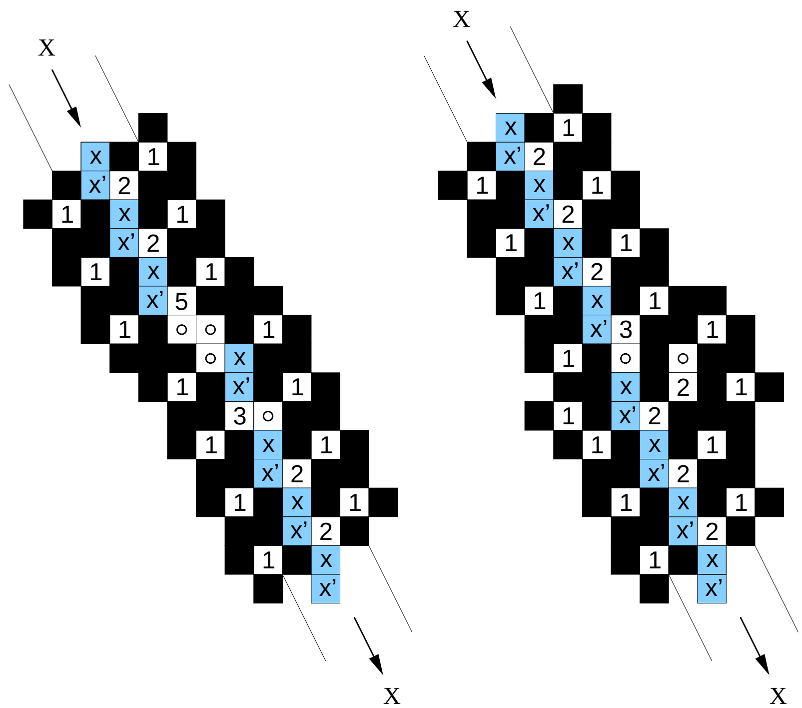


Figure 4.8: Two variants of a phase shifter

uninterrupted across the border of the adjacent bounding boxes. If we place all of our other gates in bounding boxes of compatible sizes, we can design our circuits by tiling rectangles. We should additionally ensure when we do this that wires line up across the borders. We can find large enough bounding boxes such that the wire tiles between gates, using phase shifters and bends, always line up correctly with the inputs and outputs of gates. Fig. 4.9 gives an example of a possible bounding box for an OR gate. Fig. 4.10 presents a circuit constructed with these Nurikabe tiles.

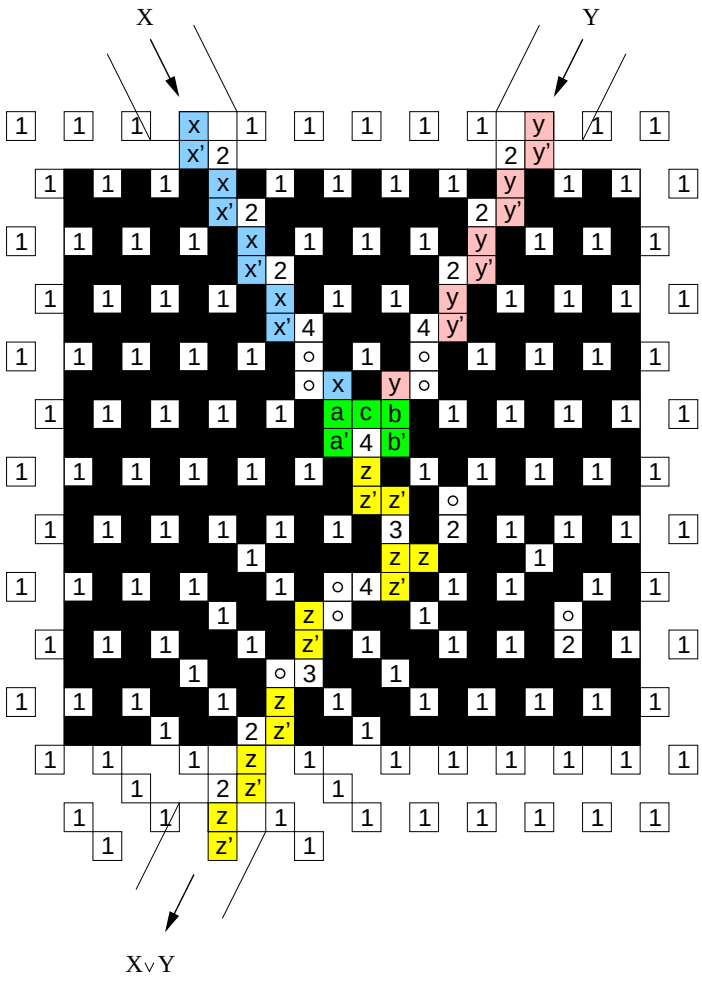


Figure 4.9: One possible bounding box for an OR gate

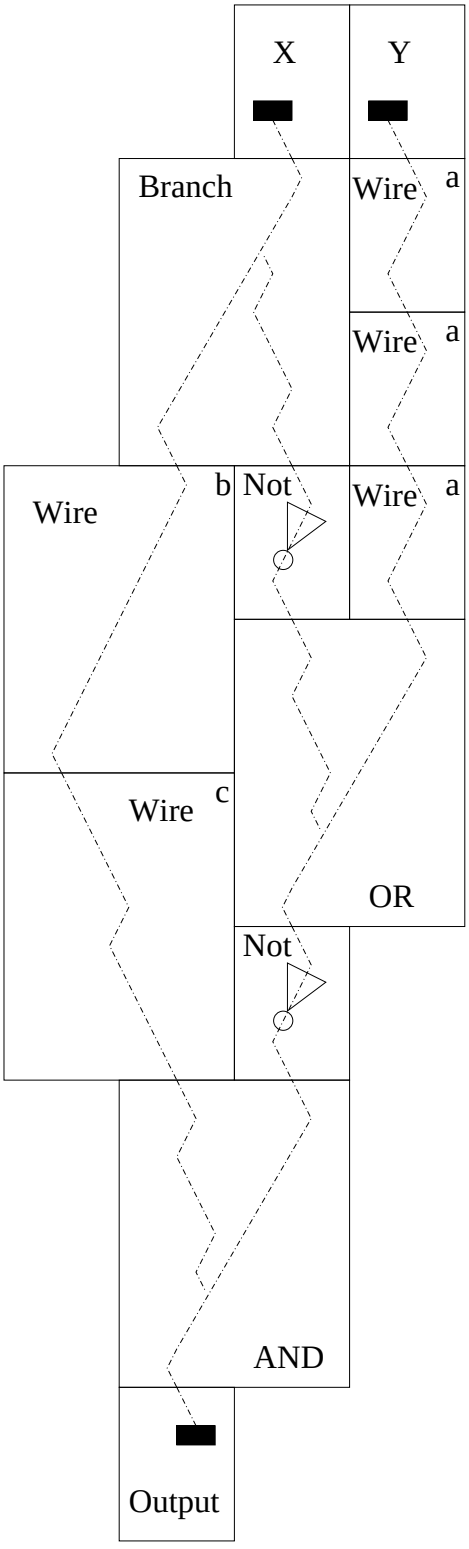


Figure 4.10: A Boolean circuit using Nurikabe tiles corresponding to $x_1 \wedge \neg(x_2 \vee \neg x_1)$

4.4 Polynomial-time reducibility

Before our proof is complete, we need to verify that any given Boolean circuit $\mathcal{C}$ can be converted to a corresponding Nurikabe grid in time polynomial in the number of inputs for $\mathcal{C}$.

Lemma 4.4. *The mapping of Circuit-SAT to Nurikabe can be realized in polynomial time.*

Proof. Since each circuit component can be placed in a bounding box of size at most k^2 for some fixed k and the number of Nurikabe tiles is polynomial in the number of edges and nodes in $\mathcal{C}$, it follows that the circuit is constructible in polynomial time. □

Theorem 4.5. *Nurikabe is NP-complete.*

Proof. Using Theorem 2.5, the proof of NP-completeness follows directly from Lemmas 4.1, 4.2, and 4.4. □

Chapter 5

Minesweeper

Minesweeper was originally released by Microsoft in 1992 as part of Windows 3.1. The computer game, written by programmers Robert Donner and Curt Johnson, has since remained a standard on all subsequent releases of Microsoft operating systems. One could reasonably conjecture that no other computer game has seen as many players — or consumed as many misspent corporate man-hours.

Briefly described, each game of Minesweeper in the computer version consists of a grid of cells. Initially, all cells are *covered*. Before the game begins, the computer hides *mines* in some of the covered cells and tells the player how many mines are hidden among the cells. The player clicks on a covered cell to reveal it. If the covered cell contains a mine, the game is over. Otherwise, the cell is labeled with a number representing the total number of cells in its 8-neighborhood that contain mines. Every time a cell is successfully uncovered, the computer rewards the user with some additional information about surrounding cells. In order to win a game of Minesweeper, the player must carefully use the information provided by the numbered cells to deduce the location of mines in neighboring covered cells. When the number of covered cells remaining equals the number of mines, the player wins and the game is over.

We are more interested in a paper-and-pencil variant of the computer game of Minesweeper. If at any point in a game of Minesweeper we press the "Print Screen" button on the keyboard and walk to the printer, what comes out on paper is a paper-and-pencil or "offline" version of the game. Instead of searching for one cell that cannot contain a mine and revealing it to receive new information, our goal is simply to find a placement of mines on the sheet that is consistent with the numbered cells. Fig. 5.1 provides an example of a game of Minesweeper after some number of clicks and a possible placement of mines that is consistent with the numbering. In this sense, the computer game of Minesweeper is a succession of "offline" games, differing only in one cell.

Considerable effort has been expended on the part of Minesweeper enthusiasts to create the ultimate program to play Minesweeper. Currently, the only programs that can always find a solution (if one exists) require an exhaustive search of all possible locations for mines. The question of the existence of an efficient Minesweeper solver was reduced to $P \stackrel{?}{=} NP$ when Richard Kaye proved NP-completeness for the prob-

lem of determining the existence of a solution to a given Minesweeper grid. In his proof, Kaye constructs a reduction of Circuit-SAT to Minesweeper. The proof that Nurikabe is NP-complete was inspired largely by his article in *The Mathematical Intelligencer*[11].

5.1 Formalizing Minesweeper

A formalization of the Minesweeper problem will be helpful. Intuitively, an instance of Minesweeper is a grid of:

1. *Numbered cells*: This cell is known not to contain a mine. The number provides us with a useful piece of information about the number of mines among its 8 neighbors.

2. *Covered cells*: It is initially unknown whether this cell contains a mine.

The computer game also includes a count of the total number of mines for the grid. We do not include this additional constraint in our formal description of an instance.

Definition 5.1. *An **instance of Minesweeper** is an $n \times m$ grid of cells from the alphabet*

$$\Sigma = \Big\{\boxed{0}, \boxed{1}, \boxed{2}, \boxed{3}, \boxed{4}, \boxed{5}, \boxed{6}, \boxed{7}, \boxed{8}, \boxed{?}\Big\}.$$

The game of Minesweeper is played by deducing which of the covered cells actually hide mines and which do not. That is to say, we *assign* a mine to a subset of the covered cells. If our assignment of mines to covered cells is consistent with the numbered cells, then we have found a solution.

To improve the readability of figures and conform to the computer game's representation of this puzzle, we will use $\square$ in place of $\boxed{0}$ in all of our figures.

Definition 5.2. *Let $\Gamma := \Big\{\square, \blacksquare\Big\}$. An **assignment** is a mapping $\alpha \colon \mathbf{N} \times \mathbf{N} \longrightarrow \Gamma$ such that for each x_{ij} of an instance I of Minesweeper*

$$\alpha(x_{ij}) := \alpha(i, j) = y \text{ for some } y \in \Gamma.$$

Definition 5.3. *An assignment α **satisfies** or **solves** an instance I of Minesweeper if and only if the following two conditions hold:*

1. *For all $x \in I$, if $\alpha(x) = \blacksquare$, then $x = \boxed{?}$.*

2. *Let $\delta \colon \square \mapsto 0, \blacksquare \mapsto 1$ be a mapping from Γ to $\{0, 1\}$. Then for all $x \in I$, if $x = \boxed{\text{k}}$ for some k, then*

$$\sum_{y \in \aleph_8(\alpha(x))} \delta(y) = k.$$

*We call α a **solution** to I.*

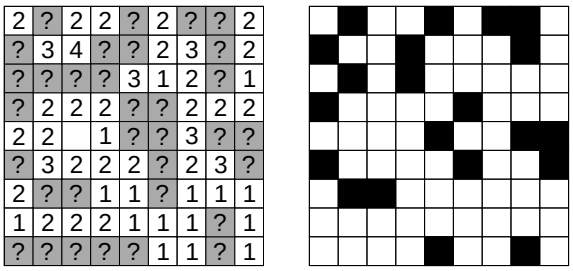


Figure 5.1: An instance of Minesweeper (*left*) and one possible solution (*right*)

Definition 5.4. *An instance of Minesweeper is* ***satisfiable*** *if and only if there exists an assignment that solves it.*

Fig. 5.1 provides an example of an instance of Minesweeper with a solution. With this tool chest of definitions, we are now in the position to reason about the problem of deciding whether a solution exists for a given Minesweeper grid.

Definition 5.5. (MINESWEEPER)
Given an instance I of Minesweeper, is I satisfiable?

5.2 Minesweeper is NP-complete

We present here a sketch of Kaye's proof of NP-completeness for Minesweeper.

Lemma 5.1. *Minesweeper* $\in NP$.

Proof. Let I be an $n \times m$ instance of Minesweeper.

Nondeterministically guess some subset A of cells in I, and define an assignment α that maps some cells in A to ■ and all others to □. We can verify that α is a solution to I in polynomial time using the following deterministic algorithm:

On input I and α:
    for each x in I
        if $\alpha(x) =$ ■ then
            if $x \neq$ [?] then
                REJECT
    for each x in I
        if $\alpha(x) =$ ■ then
            for each y in $\aleph_8(x)$
                if $y =$ [0] then
                    REJECT
                otherwise if $y \neq$ [?] then
                    $y \leftarrow y - 1$
    for each x in I
        if $x \neq$ [0] and $x \neq$ [?] then
            REJECT
    ACCEPT

Our algorithm consists of two parts corresponding to the two conditions in Def. 5.3. The first part searches for mappings of α that assign a mine to a cell that is not covered. The second part searches for cells in I that map to mines under α and decrements each neighboring numbered cell. If a cell cannot be decremented (i.e. it is zero), then too many mines were assigned to neighboring cells. The third part confirms that each numbered cell is now zero. If any numbered cell is greater than zero, then not enough mines were assigned to neighboring cells. Each step runs in linear time, hence the solution is polynomial-time verifiable. □

Lemma 5.2. *There exists a polynomial-time reduction of Circuit-SAT to Minesweeper.*

Proof. We present here the components of Kaye's reduction. As before, our construction of Boolean circuits begins with the definition of a signal propagated along a wire. Fig. 5.2 presents Kaye's construction of a wire.

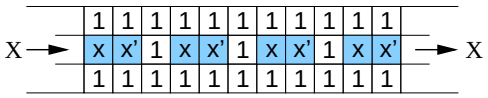


Figure 5.2: A Minesweeper wire

The cells labeled with variables are covered cells whose values cannot be determined without additional information. If the leftmost x of the this portion of a wire contains a mine, it follows that the neighboring x' does not, which in turn implies that the next x also contains a mine, and so forth and so on. We write x = ■ to symbolize the assignment of a mine to the cell x. There is a sense of orientation here in that the Boolean value for the wire propagates from left to right. We define the state of the wire as **true** if the first cell of each pair of cells (in this case x) contains a mine. Conversely, the state is **false** if x = □ instead[1]. Note that we could have just as easily used the same piece of wire in the figure to represent a wire propagating a signal from right to left instead. When constructing the instance, we can fix a value for the wire by replacing one of the covered cells in the wire with a 1. In these figures, assume that no covered cells border the edges of our wires in the constructed Boolean circuit.

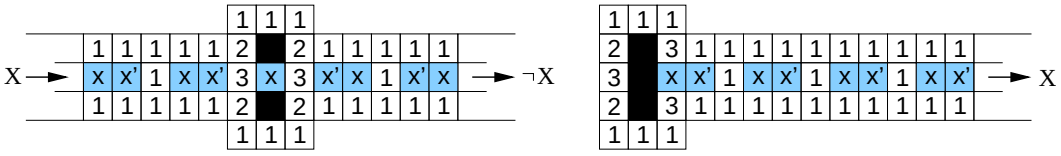


Figure 5.3: A NOT gate/inverter (*left*) and a variable terminal (*right*)

Fig. 5.3 depicts a variable terminal and an inverter or NOT gate. The black cells in these figures denote those covered cells that are deducible to be mines without

[1]This is different from Kaye's original proof, where he assigns a mine in the first cell to denote **false** and **true** otherwise. This change is not so important for our reduction; however, it means that our AND gate is an OR gate under Kaye's definitions of true and false.

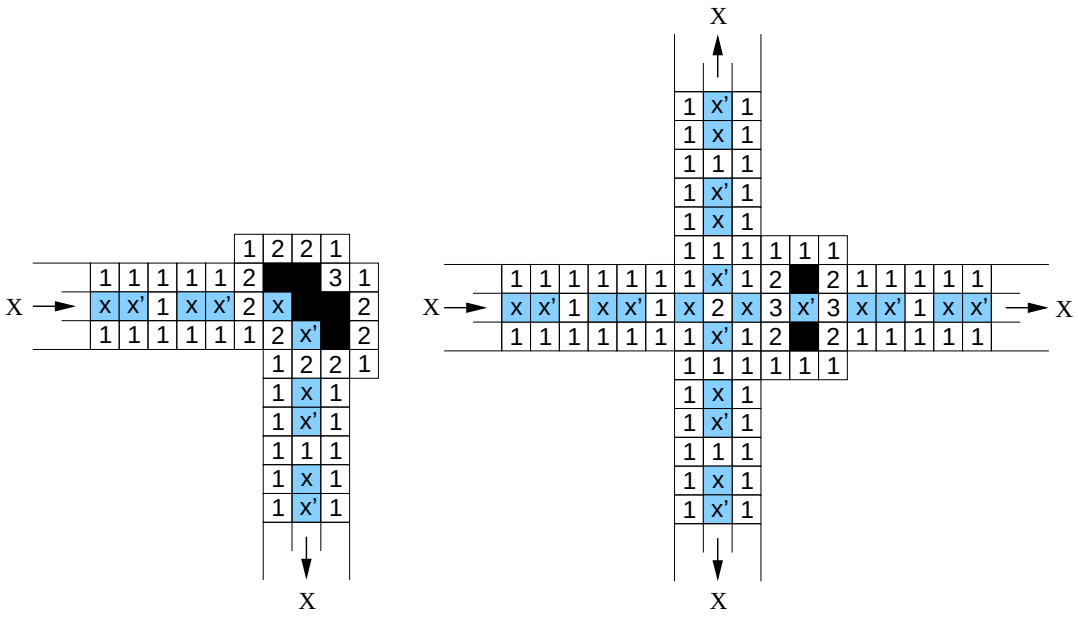


Figure 5.4: A bend in a wire (*left*) and a branch gate/splitter (*right*)

additional information about mines elsewhere in the grid. We can bend wires and split them using components of Fig. 5.4. Note the inclusion of a NOT gate in the splitter.

Claim 5.3. *Fig. 5.5 depicts an AND gate.*

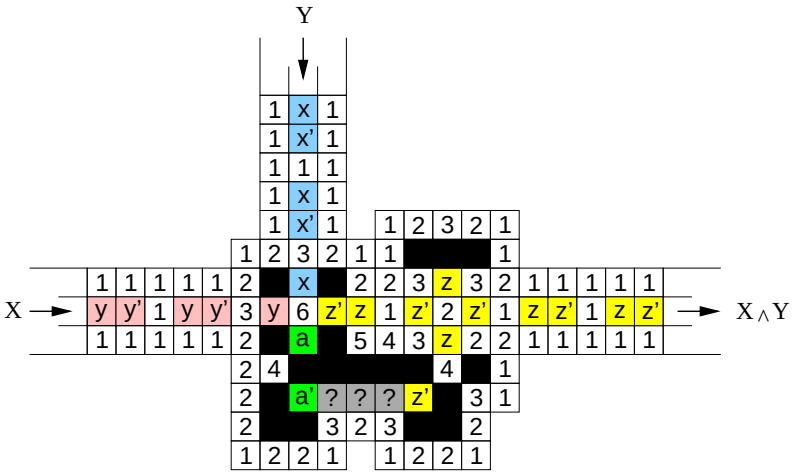


Figure 5.5: An AND gate

Proof. As we did in Chapter 4 for Nurikabe, we will construct a truth table for X, Y, and Z by specifying ■ or □ values for x, y, and z.

- Suppose now that $X = T$ and $Y = F$. Then x = ■ and y = □. Since the 6 requires one more mine within its 8-neighborhood, either a = ■ or z' = ■.

Suppose that a = ■. Then the 6 has only six neighboring mines if z' = □, which implies z = ■. Now consider the row of ?s. The 2 demands that exactly two of the three ?s contain mines. Consider now the 3 under the rightmost ?. Since z' = □, the 3 has three covered neighbors, and it follows that the row ? ? ? must be assigned values □■■. Now the 3 under the leftmost ? only has three mines in its 8-neighborhood if a' = ■. However, if a' = ■ then a = □ and we have a contradiction to our original assumption. It follows that a = □.

Now that a = □, it follows that z' = ■ and z = □. The reader may verify that the only assignment of mines to the ? cells that satisfies all surrounding numbered cells is ? ? ? = ■□■ .

Since z = □, we conclude that $Z = F$. The case for $X = F$ and $Y = T$ is analogous.

- Suppose that $X = T$ and $Y = T$. It follows that x = ■ and y = ■. With the addition of x and y, the 6 now has all 6 mines in its 8-neighborhood, so it follows that a = □ and z' = □. Thus z = ■ and $Z = T$. The reader may verify that only the multiple assignment ? ? ? = □■■ satisfies all surrounding numbered cells.

- Suppose now that $X = F$ and $Y = F$. Then x = □ and y = □. Since the 6 requires two more mines within its 8-neighborhood, both a = ■ and z' = ■. It follows that z = □ and $Z = F$. The reader may verify that only the multiple assignment ? ? ? = ■■□ satisfies all surrounding numbered cells.

Our argument by cases produces the following truth table:

X	Y	$Z = X \vee Y$
T	T	T
T	F	F
F	T	F
F	F	F

□

Our AND gate may be combined with a NOT gate to produce a NAND gate. Kaye offers an alternate construction of a NAND gate that can be found in the appendix. To ensure that all components line up, Kaye also shows that a phase shifter may be easily constructed by composing NOT gates.

We can convert any given Boolean circuit $\mathcal{C}$ into a corresponding Minesweeper grid in polynomial time, since, like Nurikabe tiles,

- each circuit component can be placed in a bounding box of size at most k^2 for some precomputed k, and

- the number of circuit components in our Minesweeper board is polynomial in the number of edges and vertices in $\mathcal{C}$.

It follows that the circuit is polynomial-time constructible. □

Theorem 5.4. *Minesweeper is NP-complete.*

Proof. The assertion is an immediate consequence of Theorem 2.5 applied to the results of Lemmas 5.1 and 5.2. □

5.3 Finding another solution

Perhaps, if we cannot create an efficient solver for Minesweeper, we can at least modify the Minesweeper program to produce Minesweeper grids that can always be solved[2]. This does not guarantee that every game of Minesweeper is easy for the player (Minesweeper is, after all, NP-complete), but it does ensure that the solution can always be found[3].

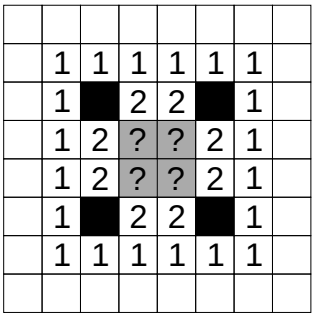


Figure 5.6: A sweeper's dilemma

Our motivation begins with a dilemma. Every Minesweeper enthusiast is likely familiar with the following predicament: After a tremendous display of finger dexterity and mental acuity, the locations of all but a small number of mines have been deduced. Intuition might suggest that with most of the grid solved, locating the last few mines should be a cinch. In fact, this is often not the case. Instead, an instance of Minesweeper may emerge from multiple ways to place the mines. In this case, it is up to the puzzler to enumerate the solutions or, in the case of the computer game, to simply abandon certainty and resort to guessing. Through experience, it seems that the denser the mines, the more frequently guessing becomes an unavoidable nuisance.

We would prefer perhaps that every instance of Minesweeper emerge from only a single configuration of mines. Such a restriction on allowable Minesweeper grids

[2]The reader may object that since all cells in the Minesweeper grid are initially covered, it is impossible to find a "safe" first click. The computer game actually ensures that the first click of every game does not land on a mine. If a mine was initially placed in the first cell to be clicked, it is moved. This is one approach that we could adopt for this problem of the first click.

[3]The card game of Solitaire, for example, does not always guarantee a solution; however, the similar card game of Free Cell does and likely owes much of its popularity to this fact.

would ensure a unique solution for every "offline" game of Minesweeper and, resultingly, never force us to guess or enumerate possible locations of mines. How then, as puzzle designers, can we ensure that each of our Minesweeper grids has only a single solution? The question becomes particularly pressing if we intend to write a program to generate instances of Minesweeper which never require the player to guess. A general algorithm for constructing puzzles will likely take the form of the following:

1. Begin with an $n \times m$ grid of ? cells.
2. Replace some of the ? cells with numbered cells, perhaps randomly picking which cells to replace.
3. Check to see if the puzzle has a solution. If it doesn't, undo the modifications and return to step 2.
4. Check to see if the puzzle has another solution. If it does, undo the modifications and return to step 2. Otherwise, we're done.

Our task as computer programmers is to design an efficient method for each of the steps that allows the automatic generation of any one of all uniquely satisfiable Minesweeper instances. Since Kaye[11] has shown us that determining the existence of a solution to an instance of Minesweeper is NP-complete, we should make this our first avenue of attack. By refining step 2 to eliminate step 3, we can avoid resolving the satisfiability problem for a given instance. If we first choose a solution and only to modify the puzzle in a manner that is consistent with that solution, we are guaranteed a puzzle with a solution. In addition, any satisfiable Minesweeper instance may be generated with this method. Our modified algorithm might look like this:

1. Begin with an $n \times m$ grid G of cells over Σ and an $n \times m$ grid S of cells over Γ. Initially,
$$\forall x \in G\colon x = \boxed{0} \text{ and } \forall y \in S\colon y = \square.$$
2. Pick some cell $y_{ij} \in S$ and $x_{ij} \in G$ and set $y_{ij} \leftarrow$ ■ and $x_{ij} \leftarrow$?. Then
$$\text{for all } z \in \aleph_8(x_{ij}),\ \text{if } z \neq \boxed{?}, \text{ let } z \leftarrow z + 1.$$
3. Repeat step 2 some number of times, then pick a subset $A \in G$ and for all $x \in A$, set $x \leftarrow$?.
4. We know that the puzzle has a solution, since it was generated from one. Check to see if the puzzle has another solution. If it does, undo our changes and return to step 2. Otherwise, we're done.

We still have to deal with step 4. We are presented with an instance of Minesweeper and one solution to that instance. Can we devise an algorithm to determine the existence of a second solution to this instance? The question is complexity theoretic, so an exact phrasing of the problem should help us determine how "hard" step 4 really is.

5.3.1 Another Solution Problem (ASP)

First we should make precise our notion of finding a *different* solution given an instance and one solution to it.

Definition 5.6. *The **solution set** for an instance I of Minesweeper is the set of all solutions to I. If I has n solutions, then the solution set is denoted by*

$$\sigma(I) = \{\alpha_1, \alpha_2, \ldots, \alpha_n\}.$$

Previously, we were only interested in an answer to the question, "Given an instance I of a problem, does a solution exists?" As it turns out, asking whether a second solution exists is somewhat equivalent to asking, "How many solutions are there to I?" Problems of this sort that require an algorithm to count the number of solutions are called **enumeration problems**. For example, consider the following enumeration problem.

Definition 5.7. (#SAT)
Given a Boolean expression ϕ, what is the number of truth assignments $\tau_1, \tau_2, \ldots$ that satisfy ϕ?

The enumeration problem for a Minesweeper board is then, "Given an instance I of Minesweeper, what is the cardinality of $\sigma(I)$?" Important to this discussion is the concept of **parsimony**.

Definition 5.8. *A polynomial-time reduction R where $\Pi_1 \leq_P \Pi_2$ is **parsimonious** if for all I in Π_1, the number of solutions to I is the same as the number of solutions to an instance $R(x) = I' \in \Pi_2$, where R is our computable function.*

We are greatly assisted by the results of Ueda and Nagao[18]. In addition to publishing some of the first NP-completeness results for paper-and-pencil puzzles, the authors introduce a new class of problems called Another Solution Problems. The Another Solution Problem for a given problem can be briefly stated as follows:

Definition 5.9. (ANOTHER SOLUTION PROBLEM)
*Given any instance of a problem in NP and a solution to that instance, does **another solution** exist for this instance?*

Clearly, our initial attempts to generate instances of Minesweeper with unique solutions have brought us to consider an instance of Another Solution Problem (or ASP) for Minesweeper. We are now in the position to accurately describe the problem presented in step 4 of our algorithm.

Definition 5.10. (ASP-MINESWEEPER)
Given an instance I of Minesweeper and a solution $\alpha_1 \in \sigma(I)$, does there exist a second solution $\alpha_2 \in \sigma(I)$ such that $\alpha_1 \neq \alpha_2$?

In their paper, Ueda and Nagao outline the following method for proving NP-completeness results for ASP of any given problem in NP.

Theorem 5.5. *Given a problem Π_1 in NP, ASP for Π_1 is NP-complete if and only if the following conditions apply*

1. *For some problem Π_2, ASP of Π_2 is NP-complete.*
2. *There exists a parsimonious polynomial-time reduction of Π_2 to Π_1.*

A proof of Theorem 5.5 can be found in [18].

5.3.2 ASP-Minesweeper is NP-complete

Lemma 5.6. *ASP-Circuit-SAT is NP-complete.*

Proof. Cook's reduction to SAT is parsimonious[22] as is the reduction of SAT to Circuit-SAT[14]. □

We need to show that every satisfying assignment of an instance of Circuit-SAT corresponds to a unique satisfying assignment of an instance of Minesweeper that results from our reduction.

Lemma 5.7. *There exists a parsimonious polynomial-time reduction of Circuit-SAT to Minesweeper.*

We have already shown the existence of a reduction, so a proof of Lemma 5.7 reduces to a proof of the following assertion:

Claim 5.8. *Every component in Lemma 5.2 yields a parsimonious reduction.*

Proof. The only variable cells in our wires and gates were those labeled with a x, x', y, y', z, z', and ?. If more than one truth assignment exists for each of these cells that give rise to the same Boolean circuit, then our reduction is not parsimonious. By definition, there is a one-to-one correspondence between the cells x, y, z and the Boolean variables X, Y, Z, respectively. In addition, the truth value for each x', y', z' is determined uniquely by each x, y, z, respectively. Only the truth values of cells in the AND gate labeled ? were not bound directly to those of Boolean variables. The proof of Claim 5.3, however, shows that every case for Boolean values of the inputs results in a unique combination of values for the row of cells ? ? ?. The reduction preserves the number of solutions and is, therefore, parsimonious. □

The reader may find it an interesting exercise to check whether our reduction remains parsimonious if we use the construction of a NAND gate presented in Fig. A.10 instead of our AND gate[4].

Theorem 5.9. *ASP-Minesweeper is NP-complete.*

[4]The reduction is not parsimonious, since, while one set of truth assignments for inputs X and Y always determines a truth value for the output Z, the states of cells labeled with ?s is not completely determined.

Proof. This is an immediate consequence of Theorems 5.4 and 5.5 and Lemmas 5.6 and 5.7 above. □

We should conclude, as programmers, that the problem of generating uniquely satisfiable puzzles is NP-complete, unless we are willing to accept some compromises. We could, for example, limit the set of puzzles that can be generated by our program and hope to find a set of heuristics that lets us quickly construct puzzles with exactly one solution. The set, however, would be only a proper subset of all uniquely satisfiable instances of Minesweeper. Of course, if it isn't, then we've solved one of the greatest open questions in the field of mathematics and proved $P = NP$.

5.4 Restricting Minesweeper

A common practice among complexity theorists after proving complexity results for a certain problem Π is to restrict the problem slightly to see if the property still holds. We introduce a restricted version of Minesweeper called k-Minesweeper.

Definition 5.11. (k-MINESWEEPER)
An instance I of k-Minesweeper is an $n \times m$ grid of cells over the alphabet

$$\Sigma = \left\{ \boxed{0}, \boxed{1}, \ldots, \boxed{k}, \boxed{?} \right\}$$

where $k \leq 8$. Given an instance I of k-Minesweeper, is I satisfiable?

5.4.1 3-Minesweeper is NP-complete

For some small enough k, Minesweeper, we conjecture, is no longer NP-complete, and we may be able to show that the problem lies in P. Theorem 5.4 provides us with a proof that 6-Minesweeper is NP-complete, but what about values of k smaller than 6?

We have a proof of NP-completeness for the case $k = 3$. We should note that we do not allow the puzzler to gain any information here. That is, the following style of reasoning cannot be used:

"If x hides a mine, then y does not and must have a label of 4 to describe the four mines surrounding it. However, no cell may receive a labeling greater than 3, so x cannot hide a mine."

Theorem 5.10. *3-Minesweeper is NP-complete.*

Proof. The following gadgets of our reduction in the proof of Lemma 5.2 require numbered cells $\boxed{k}$ where $k > 3$:

1. The variable terminal (Fig. 5.3) contains a $\boxed{4}$ if $\boxed{x'}$ = ■. We provide a 3-Minesweeper version of a variable terminal in Fig. 5.7.

2. When assigning truth values to the bends in our wires (Fig. 5.4), one of $\boxed{x'}$ or $\boxed{x}$ always contains a $\boxed{4}$ when not covering a mine. We provide a construction in Fig. 5.7 that is valid in restricted 3-Minesweeper.

3. The AND gate (Fig. 5.5) most conspicuously contains a 6 in the "kernel" of its workings in addition to many other cells with labels greater than 3. Fig. 5.8 presents a construction of an OR gate that, for each truth value of X and Y has no labelings greater than 3.

Claim 5.11. *Fig. 5.8 depicts an OR gate and an XOR gate.*

We leave the proof of Claim 5.11 as an entertaining proof to the reader. Here is the truth table for Fig. 5.8:

X	Y	$Z = X \vee Y$	$A = X \oplus Y$
T	T	T	F
T	F	T	T
F	T	T	T
F	F	F	F

As before, we can use this gate to construct NAND and AND gates and wire crossings[5]. □

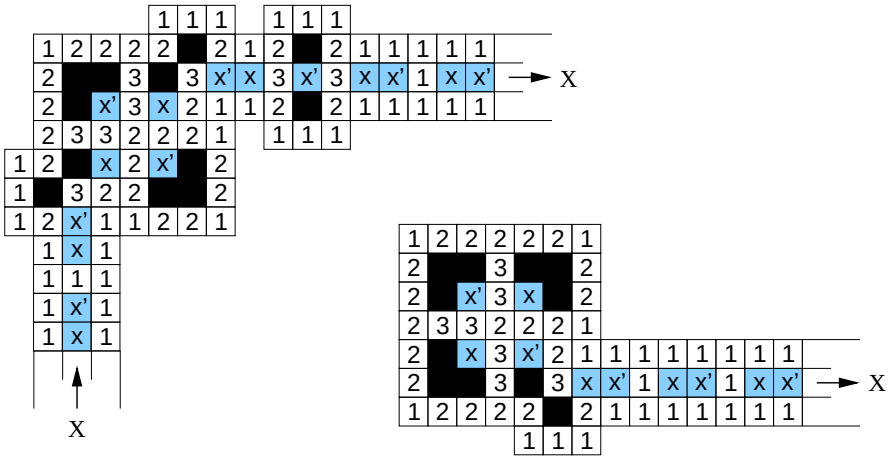


Figure 5.7: A bend in our wire (*upper left*) and a variable terminal (*lower right*)

Note that all of our 3-Minesweeper gates also yield a parsimonious reduction, so ASP-3-Minesweeper is also NP-complete. Complexity seems to be gained by including $\boxed{3}$ in our alphabet Σ. We have no proof that k-Minesweeper is NP-complete for $k < 3$; however, we do make the following conjecture.

Conjecture 5.12. *2-Minesweeper* $\in P$.

Perhaps the reader will find a polynomial-time algorithm for 2-Minesweeper. When examining restricted puzzles such as k-Minesweeper, one feels slightly closer to the hypothesized boundary between P and NP.

[5]The constructions for the NAND and AND gates can be found in the appendix.

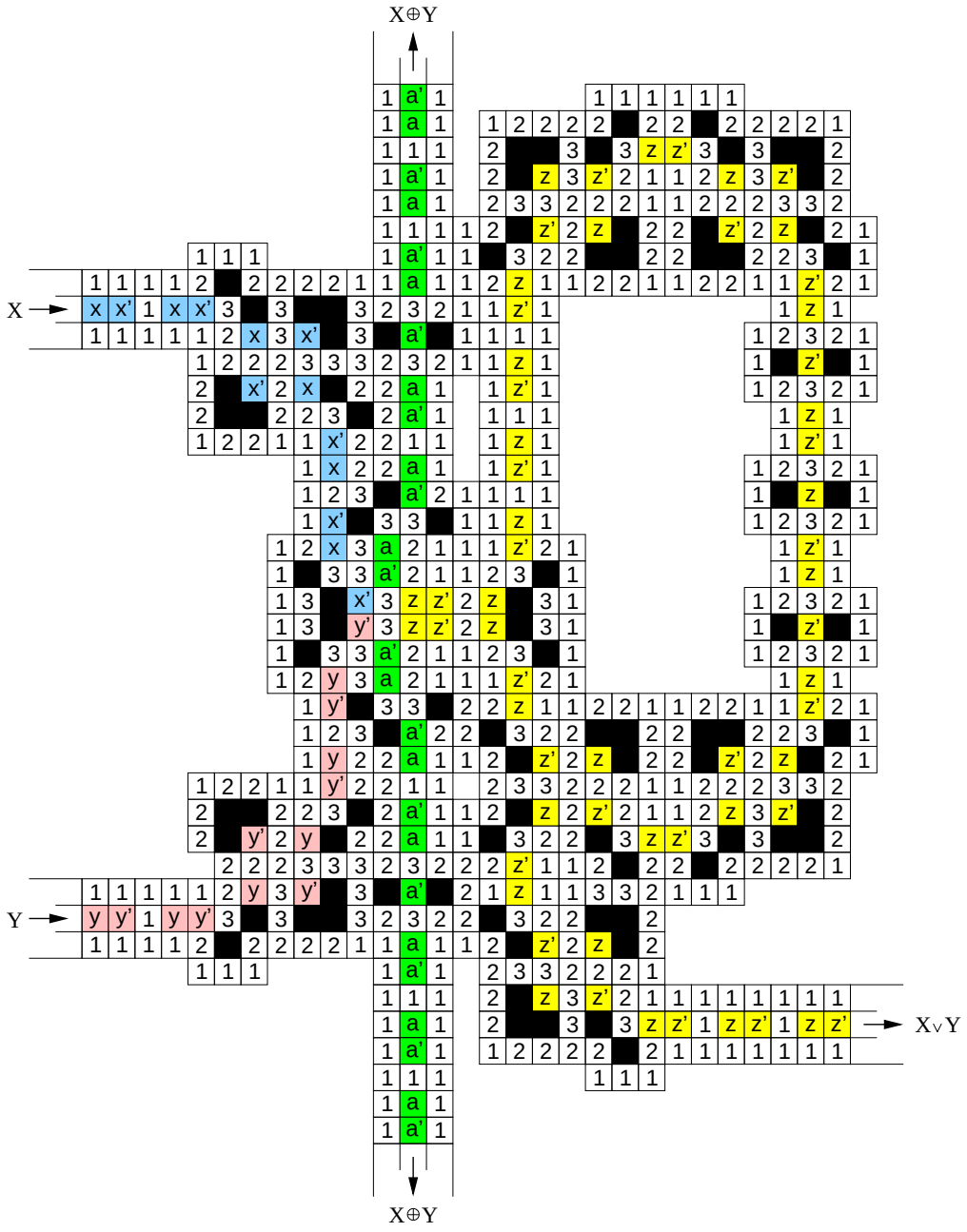


Figure 5.8: An OR/XOR gate

Appendix A

Puzzles and Circuit Components

A.1 Some Nurikabe puzzles

Here are some additional Nurikabe puzzles for the aspiring Will Shortz in all of us. Each has a unique solution. Photocopy at will.

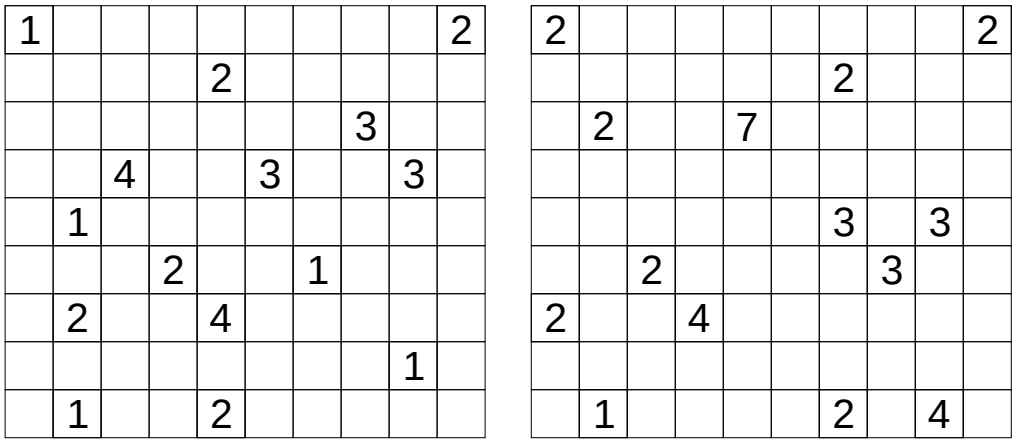


Figure A.1: An easy Nurikabe puzzle (*left*) and a more difficult puzzle (*right*)

A.2 Additional circuit components

The following constructions add to the flexibility of constructing circuits in Nurikabe and Minesweeper but were not essential to the proof of NP-completeness. I include them here for the interested reader.

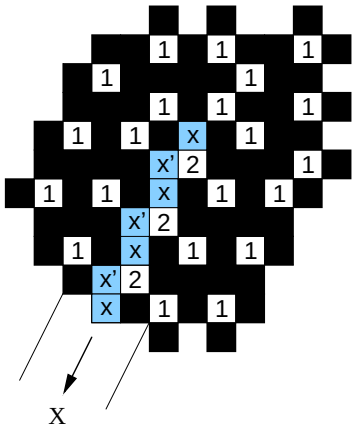


Figure A.2: An alternate variable terminal

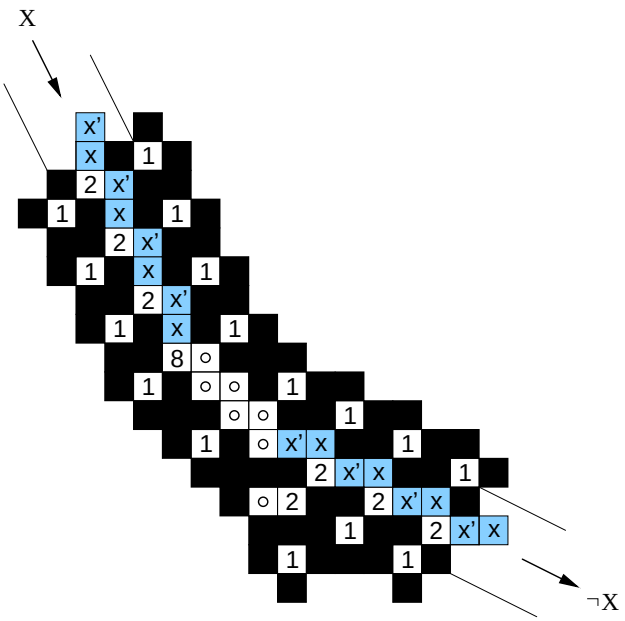


Figure A.3: A sharper corner that allows wires to travel horizontally as well as vertically

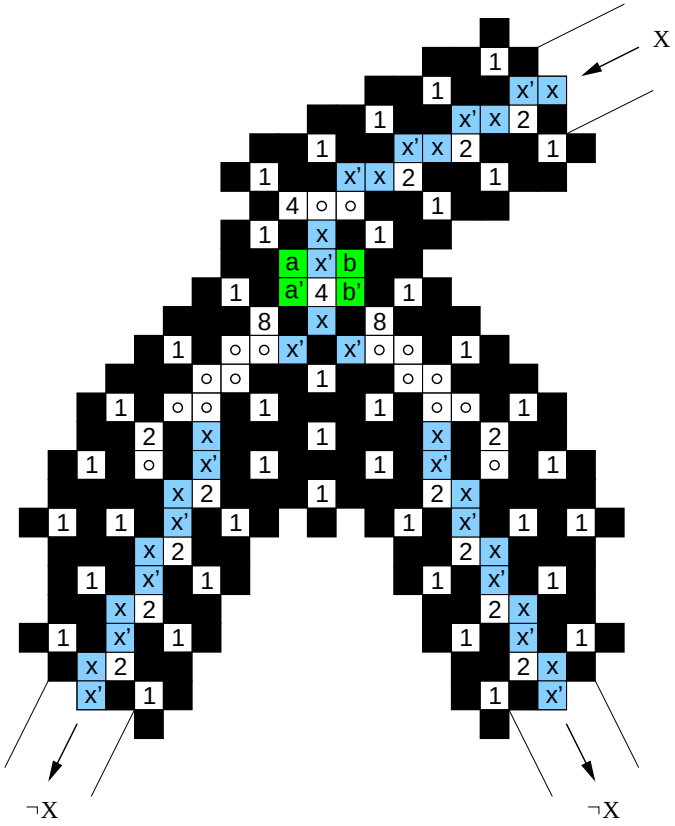


Figure A.4: An alternate splitter

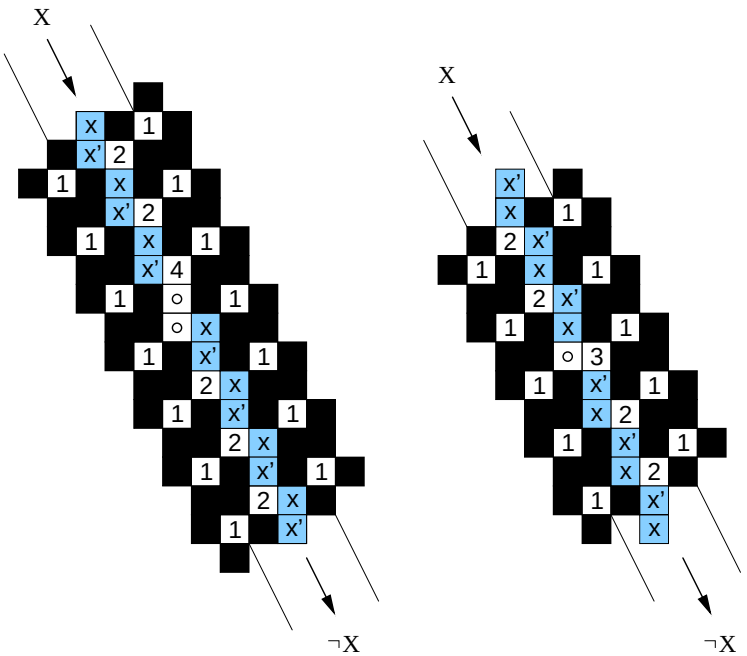


Figure A.5: Two "flipper" gates that allow a different placement of 2s within the wire

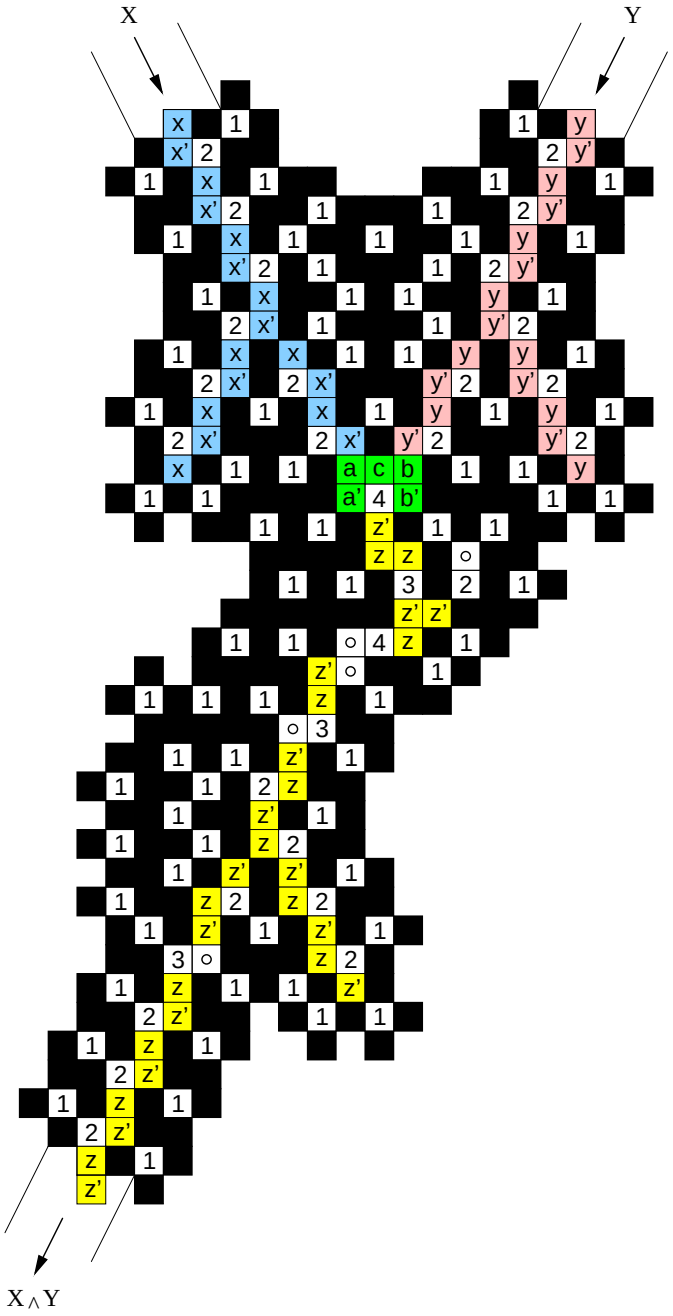


Figure A.6: An AND gate

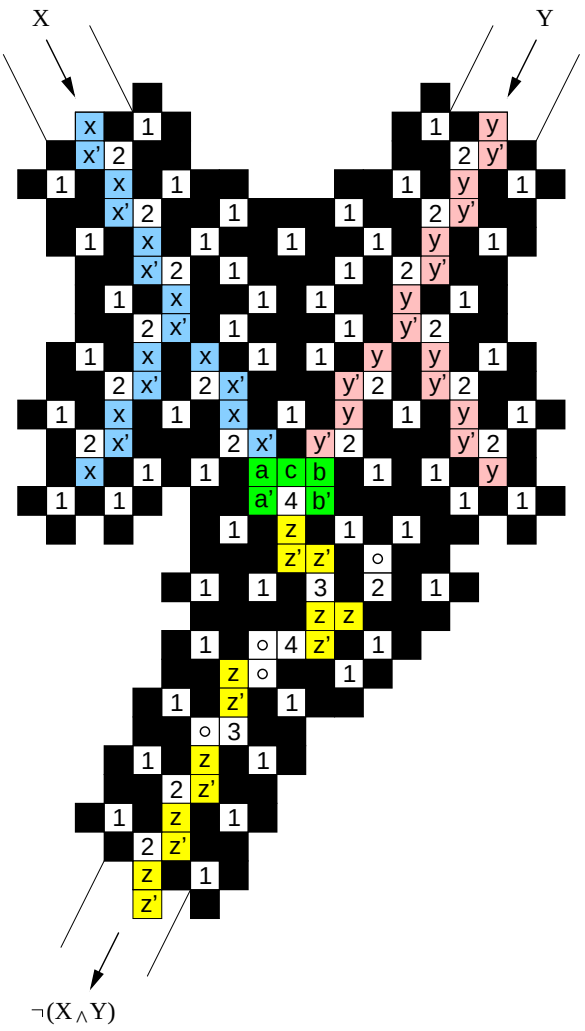


Figure A.7: A NAND gate

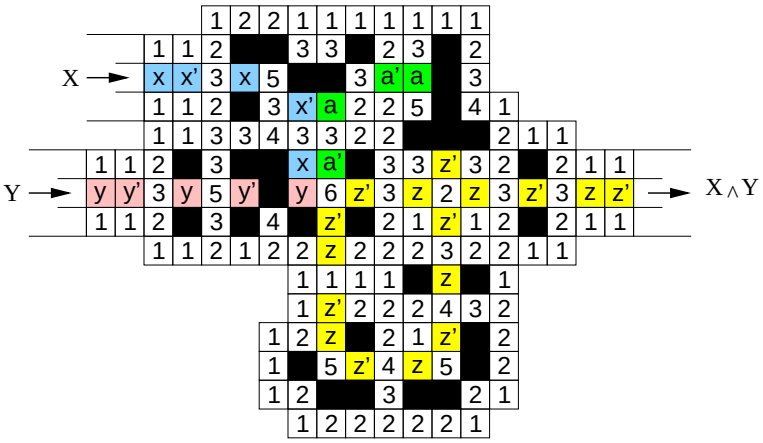


Figure A.8: An AND gate

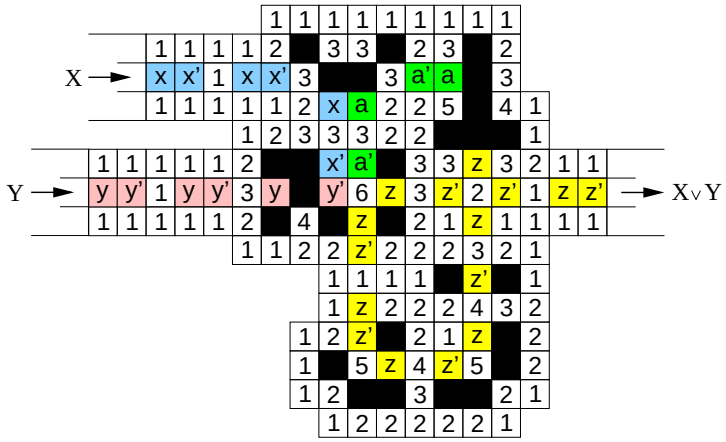


Figure A.9: An OR gate

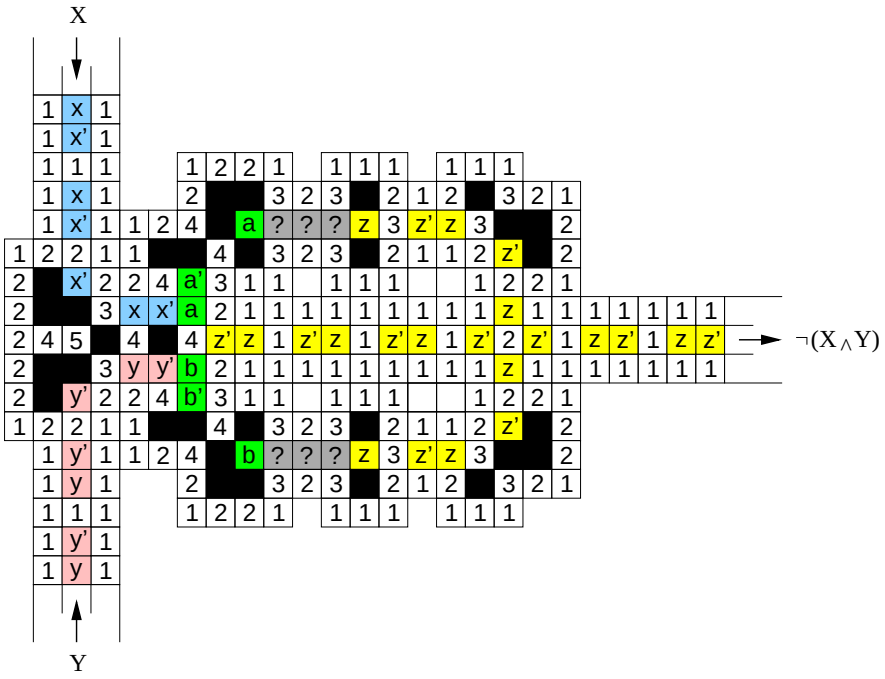


Figure A.10: R. Kaye's construction of a NAND gate

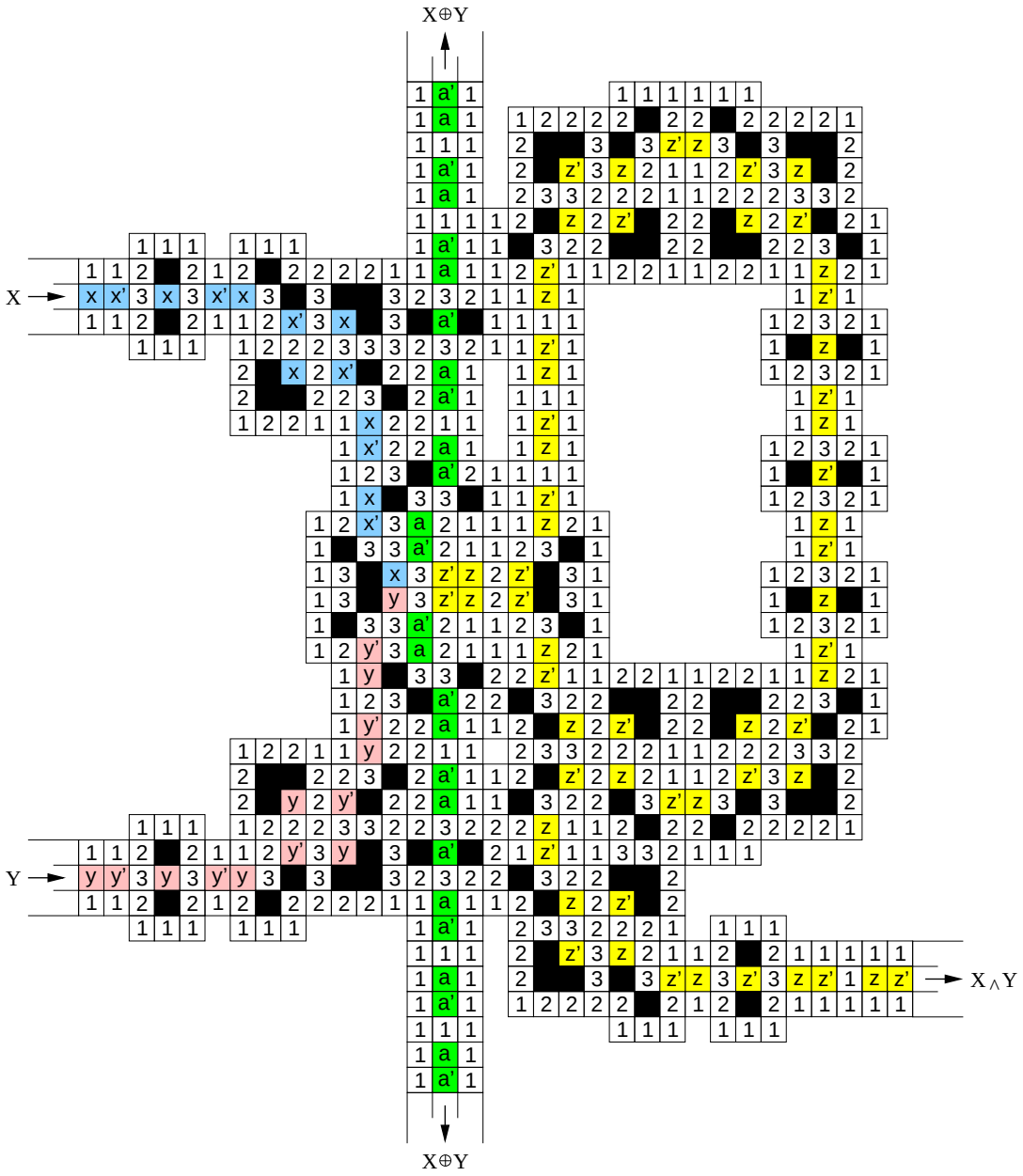


Figure A.11: An AND/XOR gate

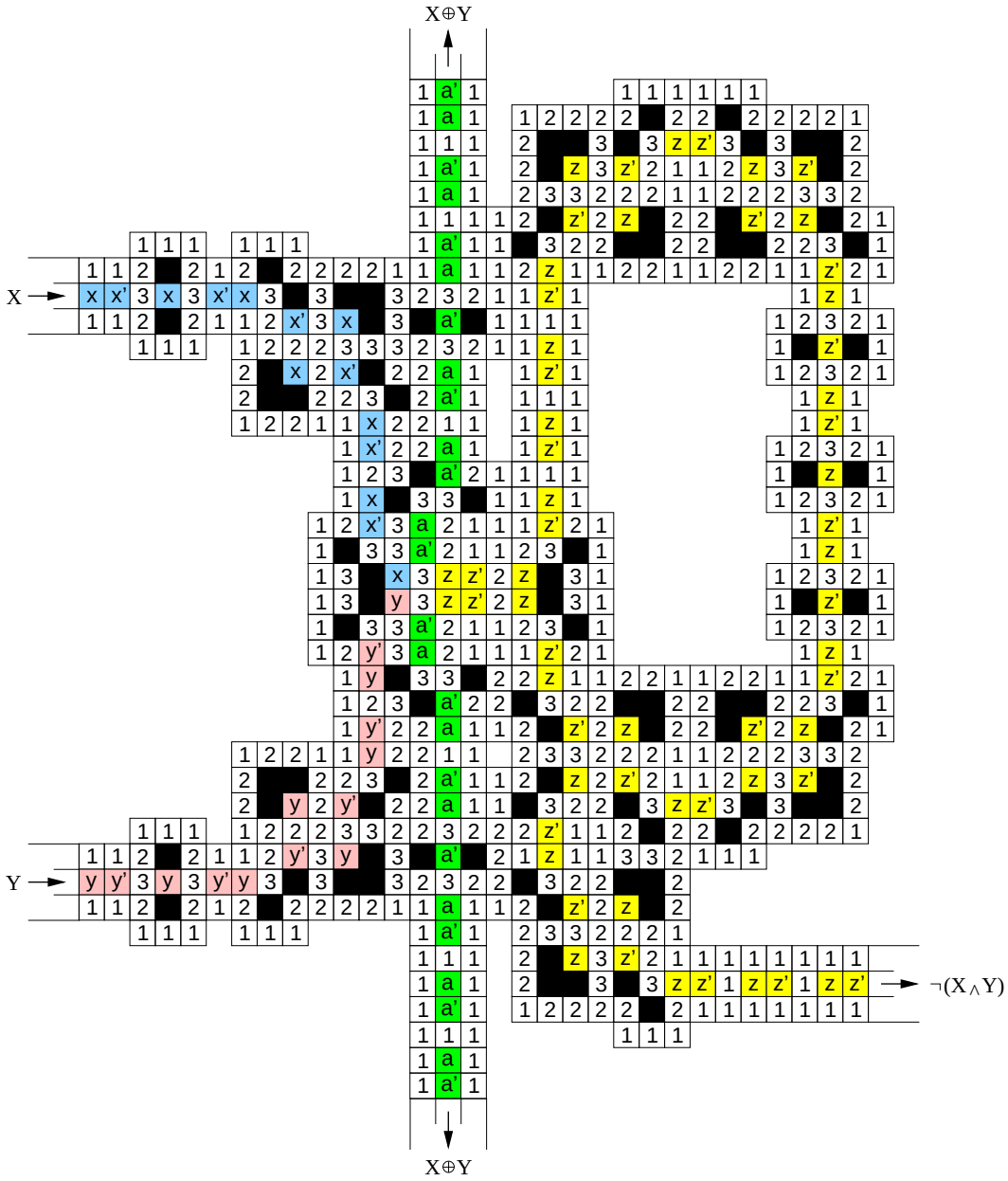


Figure A.12: A NAND/XOR gate

Bibliography

[1] L. Auslander and S. Parter. On Imbedding Graphs in the Sphere. *J. Math. Mechanics*, (10):517–523, 1961.

[2] Therese C. Biedl, Erik D. Demaine, Martin L. Demaine, Rudolf Fleischer, Lars Jacobsen, and J.Ĩan Munro. The Complexity of Clickomania, July 2000. preprint.

[3] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing*, pages 151–158, New York, New York, 1971. Association for Computing Machinery, ACM Press.

[4] H. de Fraysseix, J. Pach, and R. Pollack. How to draw a planar graph on a grid. *Combinatorica*, 1(10):41–51, 1990.

[5] Erik D. Demaine, Robert A. Hearn, and Michael Hoffman. Push-2-F is PSPACE-Complete, August 2002.

[6] Erich Friedman. Spiral Galaxies Puzzles are NP-complete. Technical report, Stetson University, 2000.

[7] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[8] John P. Hayes. *Digital System Design and Microprocessors*. McGraw-Hill, 1984.

[9] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[10] Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103, New York, New York, 1972. Plenum Press.

[11] Richard Kaye. Minesweeper is NP-complete. *The Mathematical Intelligencer*, 22(2):9–15, 2000.

[12] Richard Kaye. Some Minesweeper Configurations. Technical report, The University of Birmingham, August 2000. http://for.mat.bham.ac.uk/R.W.Kaye.

[13] C. Moore and J.M. Robson. Hard Tiling Problems with Simple Tiles. *Discrete & Computational Geometry*, 26(4):573–590, 2000.

[14] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[15] Michael Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997.

[16] Seta Takahiro. The Complexities of Puzzles, Cross Sum, and their Another Solution Problems (ASP). The University of Tokyo, 2001. Undergraduate Thesis.

[17] Alan Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society, Series 2*, 42(230):230–265, 1936.

[18] Nobuhisa Ueda and Tadaaki Nagao. NP-completeness Results for NONOGRAM via Parsimonious Reductions. Technical report, Tokyo Institute of Technology, 1996.

[19] L.G. Valiant and V.V. Vazirani. NP Is As Easy As Detecting Unique Solutions, 1985.

[20] Thomas Ryan Wilson. NP Completeness: Why Some Problems Are Hard. Reed College, 1995. Undergraduate Thesis.

[21] Takayuki Yato. On the NP-completeness of the Slither Link Puzzle. In *IPSJ SIGNotes ALgorithms*, pages 25–32, 2000.

[22] Takayuki Yato. Complexity and Completeness of Finding Another Solution and its Application to Puzzles. Master's thesis, The University of Tokyo, 2003.