2022

Games for One, Games for Two: Computationally Complex Fun for Polynomial-Hierarchical Families

Kye Shi

Games for one, games for two!

COMPUTATIONALLY COMPLEX FUN FOR POLYNOMIAL-HIERARCHICAL FAMILIES

Kye Shi

Nicholas Pippenger, Advisor

Arthur T. Benjamin, Reader

Department of Mathematics

May 2022

Abstract

In the first half of this thesis, we explore the *polynomial-time hierarchy*, emphasizing an intuitive perspective that associates decision problems in the polynomial hierarchy to combinatorial games with fixed numbers of turns. Specifically, problems in $\mathbf{P}$ are thought of as 0-turn games, $\mathbf{NP}$ as 1-turn "puzzle" games, and in general $\mathbf{\Sigma}_k\mathbf{P}$ as k-turn games, in which decision problems answer the binary question, "can the starting player guarantee a win?" We introduce the formalisms of the polynomial hierarchy through this perspective, alongside definitions of k-turn Circuit Satisfiability games, whose $\mathbf{\Sigma}_k\mathbf{P}$-completeness is assumed from prior work (we briefly justify this assumption on intuitive grounds, but no proof is given).

In the second half, we introduce and explore the properties of a novel family of games called the k-turn Graph 3-Colorability games. By embedding boolean circuits in proper graph 3-colorings, we construct reductions from k-turn Circuit Satisfiability games to k-turn 3-Colorability games, thereby showing that k-turn 3-Colorability is $\mathbf{\Sigma}_k\mathbf{P}$-complete.

Finally, we conclude by discussing possible future generalizations of this work, vis-à-vis extending arbitrary $\mathbf{NP}$-complete puzzles to interesting $\mathbf{\Sigma}_k\mathbf{P}$-complete games.

Contents

Acknowledgments

First and foremost, I wish to thank my thesis advisor, Professor Nicholas Pippenger, for generously signing on to advise me despite being retired, for being tremendously patient with me despite how frequently I showed up to our weekly meetings empty-handed, and of course for his helpful guidance throughout this project. While we're at it, let me also thank all the other wonderful professors who have mentored me through the years—Professor Stephan Garcia, Professor Dagan Karp, Professor Lucas Bang, Professor Weiqing Gu—you helped make me the mathematician and the person I am now.

I am also grateful to my friends at MathILy—sarah-marie, Tom, Hannah, Brian, Corrine, Max, Josh—you're the reason I fell in love with math in the first place, and working with you continually inspires me to approach math with unrelenting levity.

And finally, dearest thanks to my close friends outside of math: Sophia Cheng, for supporting me in every way and for being a fabulous dance partner; Forest Kobayashi, for befriending me ; Cole Kurashige, for his sagely life advice; Kaveh Pezeshki, my OneWheel buddy for life; and last but not least, my mom, for funding my exorbitant college education (and also, of course, for taking care of me my entire life). Cheers!

Chapter 1

Introduction

A quick note before we start: an up-to-date version of this document, along with its full LaTeX source code, is published on the GitHub repository https://github.com/kwshi/hmc-ph-thesis. I encourage the interested reader to engage with this project (suggesting revisions, requesting clarifications, pointing out errors or typos, etc.) by submitting issue reports on the GitHub repository or directly contacting me at kwshi@hmc.edu.

The basic question of computational complexity—"how hard is this problem for a computer to solve?"—is central to nearly every topic in computer science. And yet the formalisms of complexity theory often seem, in my own experience, intimidatingly abstract, phrased in terms of intangible models of computation such as non-deterministic Turing machines and oracles.

The remedy, I believe, lies in studying complexity theory through the lens of *puzzles* and *games*. Not only do they provide a concrete grounding for the abstractions, they also offer a particularly insightful, accessible, and most importantly fun approach to understanding complexity theory. In fact, many of the most popularly known and appreciated results in complexity theory are those about so-called "**NP**-complete puzzles", such as Sudoku, and "**PSPACE**-complete games", such as Checkers and Go.

This thesis emphasizes that approach in its exploration of a particularly foundational, yet often overlooked, ladder of complexity classes known as the *polynomial hierarchy*. **NP** is the class of (one-player) "puzzles", and **PSPACE** is the class of (two-player) "games" of polynomial length; the polynomial hierarchy, then, lies in the middle, encompassing games of *fixed* length. Through this lens, the (in)famous **P**-vs-**NP** question is but the first in a ladder of questions that are, arguably, just as crucial and impactful.

1.1 Overview

This document is structured as follows. First, chapters 2 and 3 establish preliminary background concepts and conventions adopted throughout this thesis. Next, chapter 4 lays the central theoretical groundwork, defining the *polynomial hierarchy* through a fundamental family of problems known as the Circuit Satisfiability games. Next, chapter 5 explores a novel family of games

generalized from the Graph 3-Colorability puzzle and establishes *hardness* bounds on each of those games. Finally, chapter 6 concludes by discussing the future directions of this work and its broader implications.

1.2 Prior work and inspirations

Much of the background exposition on complexity theory referenced in this thesis is reproduced from Christos Papadimitriou's textbook, Papadimitriou (1993) (though many of the foundational ideas were originally introduced/proven elsewhere, e.g. Cook (1971), Levin (1973), and Stockmeyer (1976)), reframed through the puzzles-and-games perspective and supplemented with a few comments on intuition.

The main family of games explored in this thesis, fixed-turn 3-Colorability games (chapter 5), is a generalization of (one-turn) 3-Colorability, a well-known **NP**-complete puzzle originally proven **NP**-complete by Karp (1972). Others have studied (multi-turn) game generalizations of 3-Colorability, but all versions that I've encountered are **PSPACE**-complete, in which the number of turns played during the game scales proportionally with the size of the graph (Bodlaender 1991; Kyle Burke and Hearn 2019; Beaulieu, K. Burke, and Duchêne 2013; Costa et al. 2019; Schaefer 1978). As far as I'm aware, the variations I explore here—with fixed numbers of turns regardless of the size of the graph—is unexplored, and the main theorem about its $\mathbf{\Sigma}_k\mathbf{P}$-completeness (theorem 5.10) is novel. The basic idea underlying my proof is the composition of two well-known results:

- Karp (1972)'s classic proof of the **NP**-hardness of the 3-Colorability puzzle, via a reduction from 3CNF-Satisfiability;
- Tseitin (1970)'s transformation from boolean circuits to equivalent 3CNF-clauses.

Without further ado, let's begin.

Chapter 2

Basic concepts in complexity theory

The fundamental question driving the study of computational complexity theory is, "how difficult are certain problems for computers to solve?" In order to answer this question precisely, we must start by figuring out what exactly it asks. That is, formally, what do we mean by *difficulty*? For that matter, what constitutes a *problem*? What counts as a *computer*?

Conventionally, *computers* are formalized as Turing machines, with *difficulty* being measured by the number of Turing machine execution steps. For the purposes of this thesis, we avoid delving into the formalism of Turing machines. Instead, we assume an informal notion of computers given by any algorithm or procedure straightforwardly implementable in modern, high-level programming languages such as C/C++, Python, Java, etc. Detailed treatment of the relevant formalisms may be found in Papadimitriou (1993, Chapter 2). In particular, there are theorems (Papadimitriou 1993, Theorem 2.5) showing that modern CPU/RAM-based computer architectures are, for our purposes, equivalent to Turing machines, thereby justifying the informal approach we take here.

In the following sections, we discuss what exactly constitutes a *problem*, how we describe the complexity (i.e., difficulty) of problems, and how we categorize them into *complexity classes*.

2.1 Decision problems

The simplest flavor of computational problem is a *decision problem*, or a yes/no question: given an input X, does X satisfy certain conditions? Here are some examples of decision problems:

- Given an integer K, is K even?
- Given a string of letters S, is S a palindrome?
- A silly decision problem, but nevertheless a valid one: given any input X, always return "yes".

In order for a yes/no question to qualify as a decision problem, it must be stated in terms of an arbitrary input. For instance, consider the following question:

- Is 314159 a prime number?

This is a yes/no question, but it takes no inputs (the value 314159 is not an input; it is merely part

of the question statement). In this sense, it is computationally uninteresting: in order to solve this question, an algorithm only needs to return the fixed answer "yes". In contrast, what we're really interested in is the general problem of primality testing:

- Given an arbitrary positive integer K, is K prime?

We formalize the definition of decision problems below.

Definition 2.1 ▸ (decision) problem

A **decision problem** is a function $\Pi : \{0,1\}^* \to \{\text{yes}, \text{no}\}$. Equivalently, a **decision problem** is the set $\Pi \subseteq \{0,1\}^*$ comprising exactly the inputs, a.k.a. **instances**, that result in "yes" answers.

That is, for any input $X \in \{0,1\}^*$, we say $X \in \Pi$ (in the *set* sense) if $\Pi(X) = \text{yes}$ (in the *function* sense), and $X \notin \Pi$ to mean $\Pi(X) = \text{no}$. The two formalisms are equivalent.

> ASIDE Formally, inputs to decision problems are always encoded as binary strings. Essentially, this requirement follows from the fact that all modern computers encode data in binary anyway. Furthermore, it allows us to rigorously discuss notions such as *input size*. This is an important formal detail, but for the most part, we avoid dealing with any binary encoding/decoding technicalities. We mention this detail here only to clarify the role of $\{0,1\}^*$ in the definition above.

A more general notion of *problems* considers arbitrary (binary-encoded) functions $\{0,1\}^* \to \{0,1\}^*$; such a problem is termed a **function problem**. However, we will focus mainly on decision problems for two reasons. First, decision problems are easier to work with than function problems. Second, function problems can be encoded in terms of decision problems (e.g., mapping each output bit to its own decision problem). Essentially, the *decision problem* formalism is conceptually simpler but remains versatile enough to capture the important core ideas of complexity theory. As such, the vast majority of the problems examined in this thesis are decision problems. For convenience we will simply say "problems" to mean decision problems, unless otherwise specified.

2.2 Complexities and classes

When we ask how difficult a problem is, we are essentially asking, how much time (or other resources, such as memory) does a computer need to solve that problem? Of course, the answer depends on the input: some inputs are easy to solve, and others are harder. Certainly, we expect the difficulty to scale with input size: the larger the input, the more work it generally takes an algorithm to process it. Thus, the complexity of a problem is given as a function of the input size. Specifically, we ask, if an algorithm is given an input string (recall, encoded in binary) of length n, how much time in the worst case is required, as a function of n?

However, exact function bounds are unnecessarily sensitive to pedantic technicalities, e.g., slight variations in implementations of the same algorithm, or specific details in the formal models of "computer". Instead, loosely speaking, we are mostly interested in how these costs asymptotically *scale* as the input size gets large. Thus, we categorize problems with "similar" complexities into

complexity classes.

So then, what counts as *similar*? As a starting approximation, we assert that *polynomials are small*: any algorithm whose running time is bounded by some polynomial function is considered relatively "fast"; problems with polynomial-time solutions are considered relatively "easy". We formalize this idea in the definition of the complexity class **P** below.

Definition 2.2 ▸ Polynomial-time problems, P

Let A be an algorithm computing some (decision) problem (i.e., it takes a binary string as input and returns "yes" or "no"). We say A runs in **polynomial time** if there exists some polynomial p such that, on any input $X \in \{0,1\}^*$, the algorithm A *always* terminates in $\leq p(|X|)$ steps.

The complexity class **P** is the set of (decision) problems correctly solvable in polynomial time.

> ASIDE For contrast, an algorithm is **super-polynomial** if its running time isn't bounded by any polynomial. Examples of super-polynomial functions include $n^{\log n}$, 2^n, etc.

To be clear, taking polynomials to mean "easy" is a very crude rule-of-thumb: there are important practical subdivisions *within* **P** that this categorization plainly ignores (e.g., linear-time vs. quadratic-time); there are also a few notable examples of super-polynomial-time algorithms that are, by this rule, slow, but quite efficient *in practice* (e.g., the simplex algorithm for linear programming). Nevertheless, this delineation remains an extremely useful (and arguably elegant) starting point for the classification of problems.

2.3 Hard problems and reductions

Above, we establish that a problem is considered *easy* if it has a polynomial-time solution. Hard problems, then, are those without polynomial-time solutions... right? Sure. But how do we go about showing that a problem is actually hard? And how hard, exactly?

For an easy problem, proving *existence* of a polynomial-time algorithm is straightforward—simply construct one. On the other hand, for a problem that appears to be hard, we would have to prove *non-existence* of a polynomial-time algorithm—that it is *impossible* to find a polynomial-time algorithm. In general, this is incredibly difficult to show; this difficulty is largely why the infamous **P**-vs-**NP** question remains unsolved.

Instead, we take a different approach to understanding hard problems: comparing them to each other. To illustrate, consider the following two problems:

LATIN SQUARE Given a square grid of dimensions $n \times n$ (for some n), with some of its cells filled in with a number in $\{1, \ldots, n\}$, is it possible to complete the remaining cells so that each row and each column contains each number exactly once?

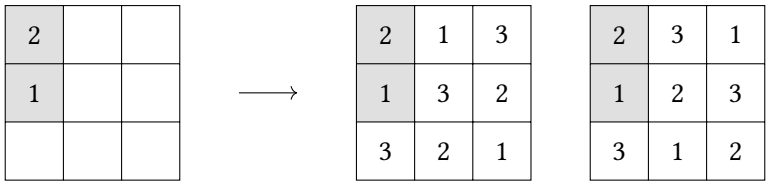


Figure 2.1. A 3×3 Latin Square instance and its two possible completions.

Graph Colorability Given a positive integer k, and a graph with $n > k$ vertices, some of which are assigned a number (a.k.a. "color") in $\{1, \ldots, k\}$, is it possible to assign numbers to the remaining cells so that no neighboring vertices receive the same assignment?

Is one of these problems "easier" than the other? In a sense, yes: the Latin Square problem is just a special case of the Graph Coloring problem, where the vertices are arranged into a square grid, and all vertices in the same row or column neighbor each other.

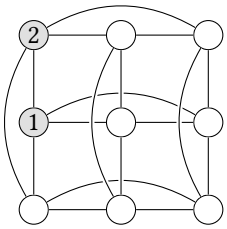


Figure 2.2. The Latin Square from figure 2.1, represented as a graph.

More precisely, this argument describes a way to convert any Latin Square instance (a partially-filled $n \times n$ grid) into a Graph Coloring instance (a partially-colored graph with n^2 vertices) with the same yes/no answer. Formally, we call this conversion a **reduction**:

Definition 2.3 ▸ reductions

Let Π_1 and Π_2 be decision problems. A **reduction** from Π_1 to Π_2 is an algorithm $R : \{0,1\}^* \to \{0,1\}^*$ such that, for each $X \in \{0,1\}^*$, $X \in \Pi_1$ if and only if $R(X) \in \Pi_2$.

In other words, R converts problem-inputs (a.k.a. *instances*) of Π_1 to problems-inputs of Π_2 such that the yes/no answers on the original and converted inputs exactly match.

Revisiting the above example, the existence of a reduction from Latin Square to Graph Colorability captures the idea that Latin Square is easier than Graph Colorability in the following sense. Suppose that we already know how to solve Graph Colorability. Then, we automatically also know how to solve Latin Square: given an arbitrary Latin Square input, apply the reduction to convert it into a Graph Colorability input, feed that input into the known Graph Coloring solver, then directly return its answer.

However, in order to authentically capture the idea of *easiness*, we must also account for computation time. Namely, we stipulate that the reduction itself must be efficient—formally, that it runs in

polynomial time.

Definition 2.4 ▸ polynomial-time reducibility

Let Π_1 and Π_2 be decision problems. We say Π_1 is **polynomial-time-reducible** to Π_2, which we denote as

$$\Pi_1 \leq \Pi_2,$$

if there exists a reduction from Π_1 to Π_2 that runs in polynomial time. (The $\leq$ notation evokes the intuition that Π_1 is easier than, or *at most as hard as*, Π_2.)

We say Π_1 and Π_2 are **equivalent**, denoted

$$\Pi_1 \equiv \Pi_2,$$

if $\Pi_1 \leq \Pi_2$ and $\Pi_2 \leq \Pi_1$.

Finally, we introduce some terminology to describe comparisons against entire *classes* of problems:

Definition 2.5 ▸ hardness and completeness

Let Π be a decision problem, and let $\boldsymbol{C}$ be a class of decision problems.

We say Π is **hard** for $\boldsymbol{C}$, or $\boldsymbol{C}$**-hard**, if every problem in $\boldsymbol{C}$ is polynomial-time-reducible to Π.

We say Π is **complete** for $\boldsymbol{C}$, or $\boldsymbol{C}$**-complete**, if Π is $\boldsymbol{C}$-hard and $\Pi \in \boldsymbol{C}$.

> **Aside** Following the intuition that reducibility ($\leq$) orders/compares problems by difficulty, $\boldsymbol{C}$-hard problems are just those that are (non-strictly) harder than all problems in $\boldsymbol{C}$, and $\boldsymbol{C}$-complete problems are just the *maximal/hardest* problems within $\boldsymbol{C}$.

Complete problems are especially useful, first and foremost, because they are tangible. They have accessible, interesting, and often real-world-applicable examples that help us understand complexity classes in concrete, intuitive terms, rather than pure abstractions. At the same time, complete problems are also very general. As *tight* difficulty upper bounds of a complexity class, they are perfect characterizations of these classes; determining the exact difficulty of a complete problem automatically essentially determines the difficulty of the entire complexity class.

2.3.1 Complements of decision problems

Every decision problem Π has a complement Π^{c}, whose yes/no question is opposite that of Π. For instance, if Π returns "yes" if a given integer n (encoded in binary) is prime, then Π^{c} returns "yes" if n is composite.

Definition 2.6 ▸ complement problems

Let $\Pi \subseteq \{0,1\}^*$ be a decision problem. The **complement of** Π, denoted Π^{c}, is same as the *set* complement of Π:

$$\Pi^{\mathrm{c}} = \{X \in \{0,1\}^* \mid X \notin \Pi\}.$$

The yes/no answer to Π^{c} is always opposite that of Π. That is, for any $X \in \{0,1\}^*$,

$$\Pi^{\mathrm{c}}(x) = \begin{cases} \text{yes} & \Pi(x) = \text{no} \\ \text{no} & \Pi(x) = \text{yes}. \end{cases}$$

> **Aside** In practice, what we call "complement" problems aren't *exactly* set complements; rather, they are complements *within* a space of strings following a particular format.
>
> For instance, in the two examples given earlier—Latin Square and Graph Colorability—the input strings encode more complicated data structures, such as tuples, arrays, and graphs. In these cases, the literal set complement would therefore also include all invalidly-formatted strings in addition to validly-formatted-but-condition-failing strings, even though in practice we use *complement* to refer only to the validly-formatted instances.
>
> Thankfully, this difference is inconsequential: assuming that string formattings/encodings can be checked and parsed in polynomial time, as is almost always the case, then it is easy to distinguish, in the complement problem, between invalid and valid-but-failing strings; thus the *true* set complement (which includes invalid strings) is equivalent to the *practical* set complement (which excludes invalid strings). Therefore, we can conflate the two modes of complement without repercussion.

It is worth mentioning that, under *our* definition of reduction, in general Π and Π^{c} aren't necessarily reducible to each other. This is perhaps counter to what one might expect, because aren't Π and Π^{c} just two faces of the same problem? Not entirely.

To understand why not, consider again the example from earlier, $\Pi =$ Primes and $\Pi^{\mathrm{c}} =$ Composites. Suppose some integer n is given, and you are asked to *prove* that either $n \in \Pi$ (n is prime) or $n \in \Pi^{\mathrm{c}}$ (n is composite). Which proof would be more straightforward?

- If n is composite, the proof is straightforward: simply provide an example of a divisor $1 < m < n$, and demonstrate that indeed m divides n.
- If n is prime, on the other hand, the proof appears less straightforward: there is way to provide no *one* example to demonstrate the proof; *every possible* divisor up to n must be checked to ensure that none of them divide n.

Basically, while Π and Π^{c} do represent two sides of the same problem, the difficulty of proving/demonstrating membership in the two sets may be different. Our definition of reduction, therefore, distinguishes between the difficulty of proving "yes"-ness and the difficulty of proving "no"-ness. (There are other more generous definitions of reducibility out there, under which Π and Π^{c} do reduce to each other, but that's besides the point: our definition makes a finer, more careful distinction and is also simpler to work with.)

For a complexity class of problems $\boldsymbol{C}$, the "dual" class formed by taking the complement of each problem in $\boldsymbol{C}$ is denoted co-$\boldsymbol{C}$.

Definition 2.7 ▸ co- **of a complexity class**

Let C be a complexity class. The complexity class co-C comprises the complement of each problem in C:

$$\text{co-}C = \{\Pi^c \mid \Pi \in C\}.$$

Note that co-C is *not* the same thing as the set complement of C (the complements are of *elements in* C, not C itself).

In general, complementation *preserves* reducibility, in the following sense:

Theorem 2.1

Let Π_1 and Π_2 be decision problems. $\Pi_1 \leq \Pi_2$ under some (polynomial-time) reduction $R : \{0,1\}^* \to \{0,1\}^*$ if and only if $\Pi_1^c \leq \Pi_2^c$ under the same reduction R.

Proof. Suppose $\Pi_1 \leq \Pi_2$ under the (polynomial-time) reduction R. Then for all $X \in \{0,1\}^*$,

$$X \in \Pi_1 \iff R(X) \in \Pi_2.$$

Equivalently, $X \notin \Pi_1 \iff R(X) \notin \Pi_2$, which is to say,

$$X \in \Pi_1^c \iff R(X) \in \Pi_2^c.$$

Thus R is also a reduction from Π_1^c to Π_2^c. Thus $\Pi_1 \leq \Pi_2$ under R implies $\Pi_1^c \leq \Pi_2^c$ under R.

The converse is true by the same proof, since $\Pi = (\Pi^c)^c$. □

A corollary of this result is that hardness and completeness (definition 2.5) are also preserved under complementation.

Corollary 2.2

Let Π be a decision problem, and C a complexity class. Π is C-hard if and only if Π^c is co-C-hard. Also, Π is C-complete if and only if Π^c is co-C-complete.

Proof. Suppose Π is C-hard; we wish to show that Π^c is co-C-hard. Let Ψ be an arbitrary problem in co-C. Then $\Psi^c \in C$, by definition of co-, and therefore $\Psi^c \leq \Pi$ because Π is C-hard. Then, by theorem 2.1, $\Psi \leq \Pi^c$. Thus every problem in Ψ reduces to Π^c, so Π^c is co-C-hard.

Next, if Π is C-complete, then Π is C-hard *and* $\Pi \in C$. By the above argument, Π^c is co-C-hard, and by definition of co-, $\Pi^c \in$ co-C. Thus Π^c is co-C-complete.

The converses of both statements hold by the same proof, since $\Pi = (\Pi^c)^c$. □

We noted earlier that decision problems in general aren't guaranteed to have same complexity as their complements, so C and co-C aren't necessarily the same complexity class.

However, it *is* true that $\mathbf{P} = \text{co-}\mathbf{P}$. In fact, this property holds for any complexity class defined by *deterministic* algorithm running times, essentially because inverting a (deterministic) algorithm's responses solves the complement problem without changing the algorithm's running time. We state and prove this result more precisely below. (For contrast, complexity classes *not* defined by deterministic running times, and therefore not subject to this result, are introduced later in chapter 4.)

Theorem 2.3

Let Π be a decision problem solvable by a deterministic algorithm in time $T(n)$, where T is a function of the input size n. Then Π^{c} is also solvable in time $T(n)$.

Proof. Let A be an algorithm that solves Π within $T(n)$ steps. Obtain a new algorithm A' by inverting all responses in A. That is, run exactly as A does, but return "yes" everywhere A returns "no", and vice versa.

By construction, A' solves the complement problem Π^{c}. Also, because A' changes nothing about the execution of A except the final return values, it runs in the exact same amount of time as A. □

Just to be explicit, we prove below the aforementioned result that $\mathbf{P} = \text{co-}\mathbf{P}$.

Corollary 2.4

$\mathbf{P} = \text{co-}\mathbf{P}$.

Proof. If $\Pi \in \mathbf{P}$, then by theorem 2.3, $\Pi^{\mathsf{c}} \in \mathbf{P}$ as well, which by definition of co- means $\Pi \in \text{co-}\mathbf{P}$. Conversely, if $\Pi \in \text{co-}\mathbf{P}$, then $\Pi^{\mathsf{c}} \in \mathbf{P}$, and again by theorem 2.3, $(\Pi^{\mathsf{c}})^{\mathsf{c}} = \Pi \in \mathbf{P}$. □

Chapter 3

A primer on boolean logic

Mathematical logic is founded on true-or-false statements—statements such as:

- property A is *true* when condition B is *false*,
- property X is *true* when both condition Y and condition Z are *true*,

and so on. Boolean logic refers to the algebra of how *truthiness* and *falsiness* combine and transform under various logical operations.

It is no surprise, given the foundational role of booleans in mathematical logic, that they also underpin all computational logic. For instance, all modern computer architectures deal with data encoded in binary 0s (*false*) and 1s (*true*). Furthermore, it follows that everything we conceive of as "computer" can be represented as boolean circuits—because, essentially, they literally are boolean circuits.

In this short chapter, we outline some basic definitions and facts about boolean-logical operations and circuits, along with some notational conventions used throughout the rest of this thesis.

Definition 3.1 ▸ basic boolean operations: NOT**,** AND**,** OR

NOT takes one input and outputs its opposite value. In boolean-algebraic expressions, we denote NOT with the symbol $\neg$.

$$\neg : \{\text{True}, \text{False}\} \to \{\text{True}, \text{False}\} \qquad \neg x = \begin{cases} \text{True} & x = \text{False} \\ \text{False} & x = \text{True} \end{cases}.$$

The NOT operation is also commonly known as **negation**.

AND takes two inputs and outputs True if and only if *both* of its inputs are True. We denote AND with the symbol $\wedge$.

$$\wedge : \{\text{True}, \text{False}\}^2 \to \{\text{True}, \text{False}\} \qquad x \wedge y = \begin{cases} \text{True} & x = y = \text{True} \\ \text{False} & \text{otherwise} \end{cases}.$$

For convenience, we sometimes omit the ∧ and simply denote AND by concatenating the operands, as in xy instead of $x \wedge y$. (This notation looks like multiplication because it is: if we represent boolean values with $\{1, 0\}$ instead of $\{\text{True}, \text{False}\}$, then $x \wedge y = x \cdot y$.)

AND is also known as the **conjunction** operation.

OR takes two inputs and outputs True if *at least one* of its inputs are True. We denote OR with the symbol ∨.

$$\vee : \{\text{True}, \text{False}\}^2 \to \{\text{True}, \text{False}\} \qquad x \vee y = \begin{cases} \text{False} & x = y = \text{False} \\ \text{True} & \text{otherwise} \end{cases}.$$

OR is also known as the **disjunction** operation.

Notationally, ∧ takes higher precedence than ∨. For instance, we interpret $x \vee y \wedge z = x \vee yz = x \vee (y \wedge z)$, and so on.

> **Aside** I personally find the ∧ and ∨ symbols for AND and OR quite easy to mix up with each other. Here's a mnemonic that helps me remember which is which:
> - ∧ looks like the A in AND, so ∧ means AND...
> - ∨ is the other one.

3.1 Algebraic properties of ¬, ∧, ∨

What algebraic behaviors do ¬, ∧, and ∨ exhibit?

Commutativity & associativity It follows straightforwardly from their definitions that they are both commutative and associative. In general, for any $x_1, x_2, \ldots, x_n \in \{\text{True}, \text{False}\}$,

$$\bigwedge_{i=1}^{n} x_i = x_1 \wedge \cdots \wedge x_n = \text{True} \text{ if and only if } \textit{every one} \text{ of } x_1, \ldots, x_n \text{ is True},$$
$$\bigvee_{i=1}^{n} x_i = x_1 \vee \cdots \vee x_n = \text{True} \text{ if and only if } \textit{at least one} \text{ of } x_1, \ldots, x_n \text{ is True}.$$

Distributivity Another interesting, sometimes useful, property of ∧ and ∨ is that they distribute over each other. For all $x, y, z \in \{\text{True}, \text{False}\}$,

$$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z), \qquad x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z).$$

3.1.1 DeMorgan's identities

Consider the statement, "x, y are both False". There are two equivalent ways to express this statement algebraically:

- x is False, and y is False: $\neg x \wedge \neg y$.
- Neither of x, y is True: $\neg(x \vee y)$.

The equivalence of these two expressions gives rise to an identity: for all $x, y \in \{\text{True}, \text{False}\}$,

$$\neg x \wedge \neg y = \neg(x \vee y).$$

Similarly, the statement "at least one of x, y is False" can be expressed in two ways,

- x is False, or y is False: $\neg x \vee \neg y$.
- x, y are not both simultaneously True: $\neg(x \wedge y)$.

This equivalence gives rise to a dual identity,

$$\neg x \vee \neg y = \neg(x \wedge y).$$

A particularly useful consequence of these DeMorgan identities is that having all three logical operations is *redundant*. We didn't need to define all three as the basic building-block operations; having only NOT/OR or NOT/AND suffices, since the third operation can simply be constructed in terms of the other two:

$$x \wedge y = \neg(\neg x \vee \neg y), \qquad x \vee y = \neg(\neg x \wedge \neg y).$$

We make use of this convenience later in chapter chapter 5, when we try to embed boolean logic within other "boolean-like" systems such as graph 3-colorings.

3.2 Boolean circuits

Boolean *expressions* such as $\neg x \wedge y$ are one way to specify computations on boolean variables. *Circuits* generalize expressions by essentially chaining together a pipeline of expressions, allowing intermediate results at each stage to be saved and reused. To illustrate, consider the following example expression:

$$\phi(x_1, x_2, y_1, y_2, z_1, z_2) = (x_1 \vee x_2)(y_1 \vee y_2) \vee (y_1 \vee y_2)(z_1 \vee z_2) \vee (z_1 \vee z_2)(x_1 \vee x_2).$$

Notice that each $(\square_1 \vee \square_2)$ term appears twice in the expression, making the expression inefficient to evaluate (each repeated term would be unnecessarily recomputed), not to mention cumbersome to specify. A more elegant way to specify this computation is to store and reuse intermediate terms in the expression:

$$\begin{aligned} X &= x_1 \vee x_2, \\ Y &= y_1 \vee y_2, \\ Z &= z_1 \vee z_2, \\ \phi &= XY \vee YZ \vee ZX. \end{aligned}$$

This chain of assignments may be visualized as a sort of data-processing "pipeline", with intermediate inputs and outputs at each stage:

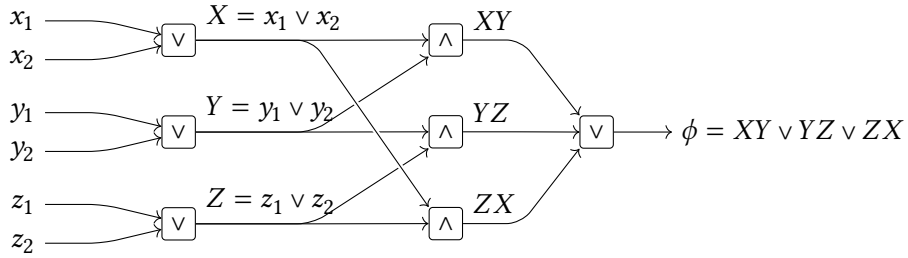


Figure 3.1. A "pipeline" of boolean operations; *almost* a boolean circuit, excepting that the last OR-gate takes three inputs.

This is *essentially* a boolean circuit. More precisely, in a boolean circuit, each variable (e.g., x_2 or Y) is represented as a *wire* carrying a boolean value, and each "stage" of computation, called a *logic gate*, computes an individual boolean operation.

For simplicity's sake, we also require that each AND/OR gate operates on exactly two inputs. Thus the last OR operation $XY \vee YZ \vee ZX$ should actually be associatively grouped as $(XY \vee YZ) \vee ZX$. The corrected circuit is shown below:

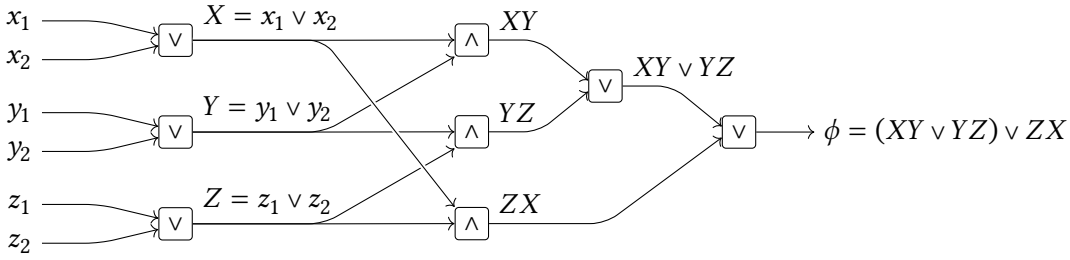


Figure 3.2. A boolean circuit.

Below, we state a precise definition of boolean circuits and introduce some relevant terminology.

Definition 3.2 ▸ boolean circuits

A **boolean circuit** C consists of:

- A set of **(circuit-level) input wires**.
- A sequence of **logic gates**. Each logic gate computes a logical operation on one or two previously-computed wires and produces its output on a new wire. More precisely, each logic gate defines a new (intermediate) **output wire** w, related to previously-defined wires by one of the following:
  - NOT gate: $w = \neg w_1$, where w_1 is an input wire or an the output wire of another logic gate specified earlier in the sequence.
  - AND gate: $w = w_1 \wedge w_2$, where w_1, w_2 are earlier-defined wires.
  - OR gate: $w = w_1 \vee w_2$, where w_1, w_2 are earlier-defined wires.

Finally, exactly one of the output wires is labeled the **circuit-level output wire** of C, representing the overall/final output of the circuit.

Assume the circuit-level input wires are ordered as $w_1, w_2, \dots, w_n$. Then the circuit C defines a boolean function $\phi_C : \{\text{True}, \text{False}\}^n \to \{\text{True}, \text{False}\}$ as follows. Given $(x_1, x_2, \dots, x_n) \in \{\text{True}, \text{False}\}^n$, assign x_j to w_j for each $j = 1, \dots, n$. Then, for each logic gate, in order of specification, evaluate the gate's boolean operation on its inputs to compute its output value, assigning that value to its output wire. Finally, after all gates have been evaluated, the boolean value of the circuit-level output wire is the final output, $\phi(x_1, \dots, x_n)$.

To assist discourse, we say a wire w *(directly) depends on* another wire w', or that w' is a **(direct) dependency of** w, if there exists some logic gate with w' as one of its inputs and w as its output.

> **Aside** We require that logic gates be specified sequentially, operating only on previous gate-outputs, to ensure that there are no cyclic dependencies, which give rise to ill-defined computations such as $x = \neg x$ (contradictory) or $x = \neg y, y = \neg x$ (not contradictory, but ill-defined because the output of a computation must be unique/deterministic), etc.

Chapter 4

Boolean circuit puzzles and games

In this chapter, we begin to explore landscape of puzzle-and-game complexity classes—specifically, the *polynomial hierarchy*—through a series of games played on boolean circuits.

4.1 The Circuit Value problem, and P

To set the stage, we start with a relatively "easy" problem, known as the Circuit Value problem, or CircVal for short:

Decision problem 4.1 ▸ Circuit Value / CircVal

Given: a boolean circuit with all inputs specified
Return whether: the circuit outputs True

It is well-known that CircVal $\in \mathbf{P}$ (i.e., it is actually "easy"). We give one version of a proof below.

Theorem 4.1 ▸ CircVal $\in \mathbf{P}$

CircVal is solvable in polynomial time.

Proof. We give a straightforward polynomial-time algorithm for Circuit Value. By our definition (definition 3.2), a boolean circuit is specified by listing out its logic gates in an order sorted by (acyclic) dependencies, so that each logic gate operates only on outputs of previous gates, or the circuit-level input wires. Thus, to evaluate any circuit, we simply iterate through and evaluate each gate, in the order that they are specified.

Algorithm 4.1 ▸ a polynomial-time CircVal solver

given: C, a boolean circuit with all inputs fully specified
▷ *Call a wire* finished *if it has been assigned a boolean value. Initially, all the input wires are finished, since their values were given, and all output wires are unfinished.* ◁
for each logic gate g in C **do**
if g is a NOT gate with input w and output w' **then**
assign the value $\neg w$ to w', thereby marking w' as *finished*
else if g is an AND gate with inputs w_1, w_2 and output w **then**
assign $w_1 \wedge w_2$ to w, marking w as *finished*
else if g is an OR gate with inputs w_1, w_2 and output w **then**
assign $w_1 \vee w_2$ to w, marking w as *finished*
end if
end for
return the boolean value assigned to the circuit-level output wire of C

The number of iterations done by the algorithm is the number of logic gates in C, which by definition is bounded by the size of C. Each iteration performs only a constant-time amount of work, plus some (insubstantial) polynomial-time bookkeeping factor—e.g., keeping track of how many gates have been visited, marking wires with their values, etc. Thus the overall running time of the algorithm is polynomial. □

To kickstart the puzzles-and-games perspective, we think of Circuit Value—and actually, every problem in **P**—as a game with 0 turns: the player does nothing, and an (efficient) algorithm automatically decides whether the player wins or loses.

This seems like a silly (arguably boring) idea. But, as we see in the next few sections, this approach allows us to generalize Circuit Value into very powerful puzzles and games.

4.2 The Circuit Satisfiability puzzle, and NP

By *puzzle*, we really mean 1-turn games: games in which a player makes a sequence of "moves" on a given "game board", and an (efficient) algorithm then determines whether the player's moves constitute a win. Formulated as decision problems, the computational puzzle is the yes/no question:

Does the player have a winning strategy?

For example, consider a puzzle-ification of CircVal, where the circuit's inputs are no longer specified but rather chosen by the player (this is the "move" made by the player). Recall that the player wins if the circuit's output is True. Thus, when we allow the player to choose inputs, a winning strategy means a combination of inputs causing the circuit to output True. The decision problem asking whether such a winning move exists is called Circuit Satisfiability, or CircSat for short:

Decision problem 4.2 ▸ Circuit Satisfiability / CircSat

Given a circuit C, determine whether there exists some assignment to its inputs causing its output to be set to True. Such an assignment is called a **satisfying assignment of** C.

Briefly: how (computationally) difficult is CircSat? As it turns out, nobody knows for sure, but it seems *quite* difficult. Loosely speaking, all known algorithms for solving CircSat amount to brute force with optimizations that enhance performance on "practical", real-world inputs but do not save them from performing poorly in the worst case. Tentatively, then, most computer scientists suspect that CircSat $\notin$ **P**—i.e., there is no polynomial-time solution for CircSat (Gasarch 2002).

Anyway, back to puzzles. CircSat is one example of how a 0-turn game such as CircVal may be generalized into a 1-turn game—a puzzle. How can we do this in general, for arbitrary games?

In the example of CircSat, we do this by making the player supplement the input to the the 0-turn analog, CircVal. This approach is readily generalized. Given some input X (the "game board"), construct a 1-turn game in which the player specifies a supplementary input Y; victory is decided by whether the pair of inputs (X, Y) meets the 0-turn winning condition, which should be an efficiently-computable condition—a problem in **P**. As before, the decision problem asks whether the player can win: does there exist Y such that (X, Y) meets the winning condition?

The complexity class of all 1-turn game problems constructed in this manner is called **NP**. Before we give the formal definition of **NP**, we need to introduce one more technical detail. In the discussion above, call Π the 0-turn winning condition problem. We said above that Π should be computable in polynomial-time. More precisely, we want it to be computable in polynomial-time with respect to the size of the *game board* X. However, the input string to the Π isn't just X but the pair (X, Y), so simply requiring $\Pi \in$ **P** is insufficient: polynomial with respect to the size of (X, Y) does not imply polynomial with respect to the size of X (Y could be arbitrarily long). To fix this disparity, we additionally require that the player's input scales controlledly with the game board: the size of Y must be polynomially-bounded by the size of X.

We are now ready to give the full definition of **NP** (the class of all 1-turn games).

Definition 4.1 ▸ **NP**

NP is the class of decision problems Π such that
there exists a $\Pi' \in$ **P** (the 0-turn winning condition) and a polynomial p such that
for each input $X \in \{0,1\}^*$ (the game board)
$X \in \Pi$ (the player can guarantee a win) if and only if
there exists $Y \in \{0,1\}^*$ (the player's move) such that $|Y| \leq p(|X|)$, and
$(X, Y) \in \Pi'$ (the move meets the winning condition).

Notice the inductive relationship between **P** and **NP**. Each problem $\Pi \in$ **NP** is constructed by *adding one turn* to some other problem $\Pi' \in$ **P**.

Unsurprisingly, CircSat is in **NP** (after all, we used it as the example problem to motivate the general definition of **NP**). To demonstrate this inclusion formally, we show how the definition of CircSat fits the definition of **NP**.

Theorem 4.2

CircSat $\in$ **NP**.

Proof. Let $\Pi' =$ CircVal. Specifically, think of Π' as a set of *pairs* (C, X) where C specifies the boolean circuit, and X specifies a input assignment to C so that $C(X) =$ True. The length of X always matches the number of input variables in C, which by definition is polynomially bounded by the size of C.

CircSat comprises exactly the set of circuits C (the game board) for which there exists an X (the player's move) such that $(C, X) \in \Pi' =$ CircVal. Thus CircSat fits the definition of an **NP** problem. □

4.2.1 Circuit Satisfiability is NP-complete

What makes CircSat especially interesting, compared to all the other puzzles in **NP**, is that CircSat is **NP**-*complete*. In other words, CircSat is the hardest of the **NP** puzzles: any other **NP** problem reduces to CircSat. This result is known as the Cook–Levin theorem:

Theorem 4.3 ▸ Cook–Levin

CircSat is **NP**-complete (Cook 1971; Levin 1973).

A full proof of the Cook–Levin theorem would require delving into formal technicalities about Turing Machines, which is beyond the scope of this thesis. Instead, we give here some informal intuition about the basic idea underlying the proof and why the Cook–Levin result makes sense.

As mentioned in chapter 3, any computer can be expressed in terms of boolean circuits; in fact, modern computers literally are implemented using boolean circuits. Therefore, the execution of any algorithm A is just a sequence of circuit computations, one for each time-step of the algorithm. Thus every 1-turn game is really just a *special case* of the Circuit Satisfiability problem.

4.3 Two-player circuit games, and the polynomial hierarchy

Recall, in the 1-turn game CircSat, a single player assigns inputs to a given circuit, with the goal of getting the circuit to output True. Now, we introduce a second player, an *antagonist*, working towards the opposite goal. The two players take turns assigning inputs in the circuit; when all inputs have been assigned, the circuit's final output dictates the winner (True $\implies$ first player wins, False $\implies$ second player wins). Now, framing this as a decision problem, we ask the yes/no question,

Does the *first* player have a winning strategy?

To start with a concrete example, consider the version of this game with 2 turns. A circuit C is given; its (unassigned) inputs are partitioned into two groups, I_1 and I_2. Two turns proceed: the first player assigns values to all inputs in I_1, then the second player assigns values to all inputs

in I_2. Finally, if the circuit outputs True, the first player wins; otherwise, the second player wins. Now, we ask, does the first player have a winning strategy?

To be more precise, by *winning strategy*, we mean a move the first player can make in order to guarantee a win, no matter what the second player plays in response. In other words, if the first player plays a winning move, then there *does not exist* a counter-winning move by the second player. Thus, in this example, what we are actually asking is, does there exist X_1 such that there does not exist X_2 setting $C(X_1, X_2) = \text{False}$? We call this decision problem CircSat$_2$.

Decision problem 4.3 ▸ Circuit Satisfiability **with** 2 **turns** / CircSat$_2$

Given: a boolean circuit C, with inputs partitioned into two groups I_1, I_2

Determine whether: there exists some $X_1 \in \{\text{True}, \text{False}\}^{|I_1|}$ such that
there does *not* exist any $X_2 \in \{\text{True}, \text{False}\}^{|I_2|}$ such that
$C(I_1 := X_1, I_2 := X_2) = \text{False}$

Earlier, we observed that CircSat could be thought of as an add-one-turn extension of CircVal. Similarly, we can formulate CircSat$_2$ as such an extension of CircSat.

To help do so, let's first formalize what it means for a player to take a single turn. When the player makes an assignment to some inputs, we say the player *augments* the circuit, creating a new circuit in which those inputs are fixed to the (constant) assigned values.

Definition 4.2 ▸ **augmented circuit**

Let C be a circuit, and let I refer to a subset of the inputs of C.

Let $X \in \{\text{True}, \text{False}\}^{|I|}$ be a boolean assignment to the inputs in I. We call the new circuit C' produced by fixing inputs I to values X an **augmented circuit** $C' = C[I := X]$.

To simplify notation, if I comprises *all* inputs of C, then we simply denote $C[I := X] = C[X]$.

Now, in the two-turn game CircSat$_2$, we start with a circuit C, whose inputs are partitioned into $I_1 \sqcup I_2$. On the first turn, the first player makes an assignment $X_1 \in \{\text{True}, \text{False}\}^{|I_1|}$ to the inputs I_1. After that assignment, the remaining circuit is the augmented circuit $C' = C[I_1 := X_1]$, whose inputs are just I_2. The first player's initial move is a winning move if and only if C' is now *unfalsifiable*—or, in other words, its negation $\neg C'$ is unsatisfiable.

$$\text{CircSat}_2 = \left\{ \begin{array}{l} \text{circuit } C \text{ with inputs } I_1 \sqcup I_2 \mid \\ \quad \exists X_1 \in \{\text{True}, \text{False}\}^{|I_1|}. \neg C[I_1 := X_1] \notin \text{CircSat} \end{array} \right\}.$$

Continuing this process gives a general construction for k-turn circuit games, in which the two players take turns assigning values to groups of inputs. Start with a boolean circuit C, with inputs partitioned into k groups, $I_1, I_2, \ldots, I_k$. On the i-th turn, the $(i \bmod 2)$-th player assigns values to the inputs in I_i; the initial player wins if the final circuit outputs True.

We formulate this game inductively as follows.

- In the base-case game with $k = 0$ turns, all inputs have been assigned values. The winning condition is determined by whether the circuit's output is True. This is the CircVal problem.
- For $k \geq 1$, the game starts with a circuit C with inputs partitioned as $I_1 \sqcup I_2 \sqcup \cdots \sqcup I_k$.

  On the first turn, the first player assigns values $X \in \{\text{True}, \text{False}\}^{|I_1|}$ to the inputs I_1, resulting in the augmented circuit $C[I_1 := X]$ with (unassigned) inputs now partitioned into $k - 1$ remaining groups, $I_2 \sqcup \cdots \sqcup I_k$.

  Now, a $(k-1)$-turn game is played, starting with the opposite player, on the *negated circuit* $C' = \neg C[I_1 := X_1]$. The negation ensures that the opposite player wins (satisfying $C' \equiv \neg C$) exactly by falsifying C. Thus the original first player wins if and only if C' is un-winnable for the second player.

Decision problem 4.4 ▸ Circuit Satisfiability **with k turns** / $\textsc{CircSat}_k$

For $k = 0$, define $\textsc{CircSat}_0 = \textsc{CircVal}$. For $k \geq 1$, define $\textsc{CircSat}_k$ as follows:

Given: a circuit with inputs partitioned into k groups, $(C, (I_1, \ldots, I_k))$
Determine whether: there exists an $X \in \{\text{True}, \text{False}\}^{|I_1|}$ such that
$(\neg C[I_1 := X], (I_2, \ldots, I_k)) \notin \textsc{CircSat}_{k-1}$

Aside. Also, observe that $\textsc{CircSat}_1 = \textsc{CircSat}$.

Finally, we can generalize this construction to arbitrary games beyond those played on circuits. Consider an arbitrary game of k turns, played on some game board $X \in \{0,1\}^*$. Two players take turns making moves $Y_1, Y_2, \ldots, Y_k \in \{0,1\}^*$. At the end, an efficient algorithm determines who wins. Stating this inductively:

- 0-turn games (winning conditions) are problems in $\mathbf{P}$.
- k-turn games start with a game board $X \in \{0,1\}^*$. The first player makes a move $Y \in \{0,1\}^*$, and then wins if and only if the "augmented" $(k-1)$-turn game, (X, Y), is a losing game for the opposite player.

For each k, the complexity class of all such decision problems is called $\boldsymbol{\Sigma}_k\mathbf{P}$. There are also the complements of problems in $\boldsymbol{\Sigma}_k\mathbf{P}$, which, instead of asking whether the first player has a winning strategy, asks whether the first player is *doomed* to lose; the class of these decision problems is called $\boldsymbol{\Pi}_k\mathbf{P} = \text{co-}\boldsymbol{\Sigma}_k\mathbf{P}$.

Together, $\boldsymbol{\Sigma}_k\mathbf{P}$s and $\boldsymbol{\Pi}_k\mathbf{P}$s constitute the *polynomial hierarchy*.

Definition 4.3 ▸ **polynomial hierarchy**

$\boldsymbol{\Sigma}_0\mathbf{P} = \boldsymbol{\Pi}_0\mathbf{P} = \mathbf{P} = \text{co-}\mathbf{P}$ (corollary 2.4) is the class of (efficient) 0-turn game deciders.

$\boldsymbol{\Sigma}_1\mathbf{P} = \mathbf{NP}$ is the class of 1-turn game "possible to win" problems (given a 1-turn board, return "yes" if the player has a winning move). $\boldsymbol{\Pi}_1\mathbf{P} = \text{co-}\mathbf{NP}$ is the class of 1-turn game, "impossible to win" problems (given a 1-turn board, return "yes" if the player has *no* winning move).

In general, for any k, $\boldsymbol{\Sigma}_k\mathbf{P}$ is the class of k-turn "possible to win" problems, and $\boldsymbol{\Pi}_k\mathbf{P} = \text{co-}\boldsymbol{\Sigma}_k\mathbf{P}$

the class of k-turn "impossible to win" problems.

Formally: let Π be any decision problem; we say Π is in $\mathbf{\Sigma}_k\mathbf{P}$ if
there exists a $\Pi' \in \mathbf{\Pi}_{k-1}\mathbf{P}$ and a polynomial p such that
for each (game board) $X \in \{0,1\}^*$
$X \in \Pi$ (is a winning game for the first player) if and only if
there exists an (initial move) $Y \in \{0,1\}^*$ such that $|Y| \leq p(|X|)$, and
$(X, Y) \in \Pi'$ (the remaining game guarantees loss for the responding player).

Notably, the circuit games CircSat$_k$ are $\mathbf{\Sigma}_k\mathbf{P}$-complete for each k.

Theorem 4.4

For each $k = 1, 2, ...,$ CircSat$_k$ is $\mathbf{\Sigma}_k\mathbf{P}$-complete (Wrathall 1976).

Again, a full proof of this result is beyond the scope of this thesis. Essentially, this theorem holds for the same reason as the Cook–Levin theorem (theorem 4.3): all algorithms can be encoded as circuits, so all problems are just special-cases of circuit problems. For our purposes, we take this theorem to be given.

In the next chapter, we will use this theorem as the central starting point for exploring and "benchmarking" the complexities of other puzzles and games.

Chapter 5

Graph 3-coloring games

In the last chapter, we set the polynomial-hierarchy stage, focusing on circuit games CircSat$_k$ as canonical examples of $\mathbf{\Sigma}_k\mathbf{P}$-complete problems. In this chapter, we expand that landscape by exploring another collection $\mathbf{\Sigma}_k\mathbf{P}$-complete games, played via colorings on graph vertices.

It is only due to time constraints on this thesis that we stop at one game: I hope to convey, through the examples presented in this chapter, the sense that there are many, many $\mathbf{\Sigma}_k\mathbf{P}$-complete games out there, all of which intuitively stem from classic, well-known **NP**-complete puzzles.

5.1 Preliminaries: graphs and proper colorings

First, we introduce some preliminary definitions about graphs and colorings. A graph is a network of *vertices* connected by *edges*. Formally:

Definition 5.1 ▸ (undirected) graphs

A **graph** Γ is a pair $(\mathrm{V}(\Gamma), \mathrm{E}(\Gamma))$ consisting of:

- A finite set of **vertices** $\mathrm{V}(\Gamma)$.
- A finite set of **edges** $\mathrm{E}(\Gamma) \subseteq \{u \leftrightarrow v \mid u, v \in \mathrm{V}(\Gamma)\}$. Visually, an edge is illustrated as a *connection* between a pair of vertices.

For our purposes, edges have no directionality. That is, when specifying an edge, the ordering of vertices doesn't matter: $u \leftrightarrow v$ is the same edge as $v \leftrightarrow u$.

We say that two vertices $u, v \in \mathrm{V}(\Gamma)$ are **adjacent**, or that they **neighbor** each other, if $(u, v) \in \mathrm{E}(\Gamma)$.

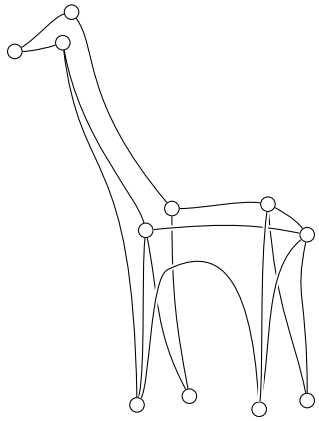


Figure 5.1. A giraph.

The graph coloring games we explore in this thesis are about assigning colors to vertices on a graph. We call such an assignment a *vertex coloring*. Specifically, for sake of simplicity, we restrict our attention to colorings that involve only three colors. The main rule constraining these color assignments is that neighboring vertices must always be colored distinctly—we call this the *properness* condition. These terms are defined precisely below.

Definition 5.2 ▸ vertex 3-colorings, properness

Let Γ be a graph. A **vertex 3-coloring** of Γ is a map $\kappa : V(\Gamma) \to \{\mathbf{0}, \mathbf{1}, \mathbf{2}\}$, which assigns to each vertex one of three colors. In this thesis, we generally just say "coloring" to refer to "vertex 3-colorings", except when specified otherwise.

A vertex coloring κ is a **proper** coloring if, for every edge $(u, v) \in E(\Gamma)$, $\kappa(u) \neq \kappa(v)$—i.e., no neighboring vertices share the same color. To simplify discourse, we also call a particular edge/neighboring-pair $(u, v) \in E(\Gamma)$ is **proper** if $\kappa(u) \neq \kappa(v)$. Thus a proper coloring is one where all edges are proper; an improper coloring contains at least one improper edge.

Having established the basic terminology, we now introduce the graph (3-)coloring games.

5.2 The 0-turn game

The goal of graph coloring games is to assign colors to all vertices so that the resulting coloring is proper. To this end, the 0-turn winning-condition problem is that of checking properness of colorings, called the 3-Coloring Properness problem, or 3ColProp for short:

Decision problem 5.1 ▸ 3-Coloring Properness / 3ColProp

Given: a 3-colored graph (Γ, κ) (specified by listing out each vertex with its color)
Determine whether: κ is a proper coloring

In order for 3ColProp to be usable as a basis for polynomial-hierarchy games, we must first ensure

that it itself is in **P**. Indeed, it is:

Theorem 5.1 ▸ 3ColProp ∈ **P**

3ColProp is solvable in polynomial time.

Proof. We describe below a straightforward polynomial-time algorithm computing 3ColProp. Simply iterate through and verify properness on each edge:

Algorithm 5.1 ▸ **a polynomial-time** 3ColProp **solver**

given: a graph Γ and a coloring $\kappa : V(\Gamma) \to \{0, 1, 2\}$
  for each $(u, v) \in E(\Gamma)$ **do**
    if $\kappa(u) = \kappa(v)$ **then**
      ▷ (u, v) *is improper!* ◁
      return no
    end if
  end for
  ▷ *all edges have been checked, no improper ones were found; the coloring is proper* ◁
  return yes

The number of edges is, by definition, bounded by the size of the graph, so the number of "for each" iterations is polynomial. Within each iteration, the $\kappa(u) = \kappa(v)$ check runs within polynomial time, so the overall algorithm runs in polynomial time as well. □

5.3 The k-turn games

A graph coloring game is played on an initially uncolored graph Γ. In a k-turn game, the graph's vertices are partitioned into k groups, $V_1, V_2, \ldots, V_k$, and players alternate turns assigning colors to the vertices in each group. If, on any turn, a player introduces an improper edge in the (partial) coloring, the other player wins. If, after all turns, no improper edges have been introduced—that is, the resulting coloring is proper—then the *last* player wins.

To help formalize this game, we define exactly what we mean by *partial coloring*.

Definition 5.3 ▸ **partial (vertex 3-)colorings**

Let Γ be a graph. A **partial (vertex 3-)coloring** is a map $\kappa : V(\Gamma) \to \{0, 1, 2, \text{None}\}$, which *optionally* assigns a color to each vertex in Γ (None means no color is assigned). Where necessary, we refer to fully-completed colorings as **total colorings** to differentiate them from partial colorings.

A partial coloring κ is **proper** if, among the vertices it *does* assign a color, there are no improper edges. That is, for all $(u, v) \in E(\Gamma)$, if both $\kappa(u)$ and $\kappa(v)$ are not None, then $\kappa(u) \neq \kappa(v)$.

At the start of the game, no vertices are colored yet—the partial coloring assigns None to every

vertex. When a player makes a move, they *augment* the partial coloring with new assignments:

Definition 5.4 ▸ augmented coloring

Let Γ be a graph, and κ be a partial coloring on Γ.

Let $U \subseteq V(\Gamma)$ be a subset of the vertices such that, for each $u \in U$, $\kappa(u) = \text{None}$ (all vertices in U are uncolored), and let $\delta : U \to \{0, 1, 2\}$ be an assignment of colors to every vertex in U. Then the **augmented coloring** $\kappa[\delta] : V(\Gamma) \to \{0, 1, 2, \text{None}\}$ is another partial coloring formed by the combining the two color assignments:

$$\kappa[\delta](v) = \begin{cases} \delta(v) & v \in U \\ \kappa(v) & v \notin U \end{cases}.$$

For any partial coloring κ', we say κ' is an **extension** of κ if there exists some $\delta : U \to \{0, 1, 2\}$ such that $\kappa' = \kappa[\delta]$.

Now, we are ready to give the full inductive formulation of k-turn graph coloring games.

- The 0-turn winning condition is 3ColProp: given a totally-colored graph, decide whether the coloring is proper.

- k-turn games begin on a partially-colored graph (Γ, κ), where the partial coloring κ comprises color assignments made in previous turns. (We discuss in section 5.6 restrictions of these games to entirely uncolored graphs.)

  If κ is improper to begin with, then we posit that the first player automatically wins, since that means that the opposite player must have made an improper move on their previous turn. Otherwise, the first player colors U_1 with a coloring δ, and wins if and only if the remaining $(k-1)$-turn game $(\Gamma, \kappa[\delta], (U_2, \ldots, U_k))$ is now un-winnable by the opposite player.

Decision problem 5.2 ▸ 3-Colorability with k turns / 3Col_k

For $k = 0$, define $3\text{Col}_0 = 3\text{ColProp}^c$. For $k \geq 1$, define 3Col_k as follows:

Given: a partially-colored graph and a partitioning of its uncolored vertices, $(\Gamma, \kappa, (U_1, U_2, \ldots, U_k))$
Determine whether: κ is improper, or there exists some $\delta : U_1 \to \{0, 1, 2\}$ such that $(\Gamma, \kappa[\delta], (U_2, \ldots, U_k)) \in (3\text{Col}_{k-1})^c$

Pay particular attention to the fact that 3Col_0 is defined as the *complement* of 3ColProp—that is, return "yes" if the graph coloring is *improper*. This might seem a little weird, but using 3ColProp^c rather than 3ColProp as the "base case" game turns out to be the more natural definition: it matches the $k \geq 1$ definition better (specifically, the first half of the winning condition, "κ is improper") overall makes generalizations on 3Col_k cleaner to state and prove.

5.4 *k*-turn 3-Colorability is in $\Sigma_k\mathbf{P}$, right?

Having defined 3Col_k as a k-turn game problem, we naturally expect that $3\text{Col}_k \in \mathbf{\Sigma}_k\mathbf{P}$ (the class of all ("reasonable") k-turn game problems). Indeed, we claim it is, but it isn't immediately obvious *how*. Specifically, membership in 3Col_k is conditioned on an extra "κ is improper or" clause that isn't present in the definition of $\mathbf{\Sigma}_k\mathbf{P}$ problems (definition 4.3):

$$3\text{Col}_k = \left\{ (\Gamma, \kappa, \dots) \;\middle|\; \underline{\kappa \text{ is improper, or}} \quad \exists\ \delta.\ (\Gamma, \kappa[\delta], \dots) \in (3\text{Col}_{k-1})^{\text{c}} \right\},$$
$$\underbrace{\Pi}_{\in \mathbf{\Sigma}_k\mathbf{P}} = \left\{ \quad B \quad \middle| \quad \exists M. \quad (B, M) \quad \in \quad \underbrace{\Pi'}_{\in \mathbf{\Pi}_{k-1}\mathbf{P}} \quad \right\},$$

By splitting up the two conditions, we can think of 3Col_k *union* of two problems:

$$3\text{Col}_k = \{(\Gamma, \kappa, \dots) \mid \kappa \text{ is improper}\} \cup \{(\Gamma, \kappa, \dots) \mid \exists \delta.\ (\Gamma, \kappa[\delta], \dots) \in 3\text{Col}_{k-1}\}$$

The first term in the union, determining improperness of κ, is basically equivalent to $3\text{ColProp}^{\text{c}}$, which is in $\mathbf{P}$ (theorem 5.1 and corollary 2.4). Meanwhile, the second term appears to comply with the $\mathbf{\Sigma}_k\mathbf{P}$ definition—if we assume (yet unproven, but sort of as an inductive hypothesis) that $3\text{Col}_{k-1} \in \mathbf{\Sigma}_{k-1}\mathbf{P}$, then the second term is indeed in $\mathbf{\Sigma}_k\mathbf{P}$.

So 3Col_k is the union of a problem in $\mathbf{P}$ with a problem *allegedly* in $\mathbf{\Sigma}_k\mathbf{P}$.

Then, it makes sense to expect $3\text{Col}_k \in \mathbf{\Sigma}_k\mathbf{P}$ for the following (conjectured) reasons:

- We expect $\mathbf{P} \subseteq \mathbf{\Sigma}_k\mathbf{P}$: any game with 0 turns can be thought of as game with k no-op turns. More generally, any k-turn game is also a $(k+1)$-turn game, with an extra no-op move by the first (or last) player; games with fewer turns are no harder than games with more turns.
- The union of two problems in $\mathbf{\Sigma}_k\mathbf{P}$ should also be in $\mathbf{\Sigma}_k\mathbf{P}$ (i.e., $\mathbf{\Sigma}_k\mathbf{P}$ is *closed* under union): intuitively, directly combining two problems doesn't make them harder.

Below, we state these conjectures in general terms and prove them.

Theorem 5.2 ▸ polynomial hierarchy inclusions

For every $k = 0, 1, 2, \dots$,

$$\mathbf{\Sigma}_k\mathbf{P} \subseteq \mathbf{\Sigma}_{k+1}\mathbf{P}, \qquad \mathbf{\Sigma}_k\mathbf{P} \subseteq \mathbf{\Pi}_{k+1}\mathbf{P}, \qquad \mathbf{\Pi}_k\mathbf{P} \subseteq \mathbf{\Sigma}_{k+1}\mathbf{P}, \qquad \mathbf{\Pi}_k\mathbf{P} \subseteq \mathbf{\Pi}_{k+1}\mathbf{P}.$$

Proof. We prove each of the four inclusions separately.

[1] Claim: $\mathbf{\Pi}_k\mathbf{P} \subseteq \mathbf{\Sigma}_{k+1}\mathbf{P}$.

This follows directly from the definition of $\mathbf{\Sigma}_{k+1}\mathbf{P}$. Let $\Pi \in \mathbf{\Pi}_k\mathbf{P}$, and define

$$\Pi' = \{(X, (\text{empty})) \mid X \in \Pi\}, \qquad p(n) = 0.$$

Note that Π' is the same problem as Π, differing only in "formatting" of inputs, so $\Pi' \in \mathbf{\Pi}_k\mathbf{P}$ as well. Thus Π fits the definition of a $\mathbf{\Sigma}_{k+1}\mathbf{P}$ game:

For all inputs $X \in \{0,1\}^*$, $X \in \Pi$ if and only if
letting Y be the empty string, we have $|Y| = 0 \leq p(|X|)$, and
$(X, Y) = (X, \text{(empty)}) \in \Pi'$.

Thus $\Pi \in \mathbf{\Sigma}_{k+1}\mathbf{P}$.

> **ASIDE** The intuition here: $\Pi \in \mathbf{\Pi}_k\mathbf{P}$ is an impossible-to-win k-turn game. Then, Π' is a $(k+1)$-turn game in which, on the first turn, the other player does *nothing*. Still, they guarantee a win, because the remaining game already dooms the second player to a loss.

[2] Claim: $\mathbf{\Sigma}_k\mathbf{P} \subseteq \mathbf{\Pi}_{k+1}\mathbf{P}$.

This follows directly from the previous result [1], since $\mathbf{\Sigma}_k\mathbf{P} = \text{co-}\mathbf{\Pi}_k\mathbf{P}$ and $\mathbf{\Pi}_{k+1}\mathbf{P} = \text{co-}\mathbf{\Sigma}_{k+1}\mathbf{P}$.

[3] Claim: $\mathbf{\Sigma}_k\mathbf{P} \subseteq \mathbf{\Sigma}_{k+1}\mathbf{P}$.

We prove this by induction on k.

- For $k = 0$, $\mathbf{\Sigma}_0\mathbf{P} = \mathbf{P} = \mathbf{\Pi}_0\mathbf{P}$ by definition. Thus the argument from part [1] applies in this case: $\mathbf{\Sigma}_0\mathbf{P} = \mathbf{\Pi}_0\mathbf{P} \subseteq \mathbf{\Sigma}_1\mathbf{P}$.
- For $k \geq 1$, assume $\mathbf{\Sigma}_{k-1}\mathbf{P} \subseteq \mathbf{\Sigma}_k\mathbf{P}$. Suppose $\Pi \in \mathbf{\Sigma}_k\mathbf{P}$. Then there exists a $\Pi' \in \mathbf{\Pi}_{k-1}\mathbf{P}$ and a polynomial p such that

$$\Pi = \{X \mid \exists Y.\ |Y| \leq p(|X|), \quad (X, Y) \in \Pi'\}.$$

Recalling that $\mathbf{\Pi}$ is just co-$\mathbf{\Sigma}$, the induction hypothesis implies $\mathbf{\Pi}_{k-1}\mathbf{P} \subseteq \mathbf{\Pi}_k\mathbf{P}$. Thus $\Pi' \in \mathbf{\Pi}_{k-1}\mathbf{P}$ is also in $\mathbf{\Pi}_k\mathbf{P}$. Consequently, Π is also in $\mathbf{\Sigma}_{k+1}\mathbf{P}$. Since Π was arbitrary, we conclude $\mathbf{\Sigma}_k\mathbf{P} \subseteq \mathbf{\Sigma}_{k+1}\mathbf{P}$.

[4] Claim: $\mathbf{\Pi}_k\mathbf{P} \subseteq \mathbf{\Pi}_{k+1}\mathbf{P}$.

This follows directly from the previous result [3], since $\mathbf{\Pi} = \text{co-}\mathbf{\Sigma}$. □

As a side note, 5.2 justifies calling the collection of complexity classes $\mathbf{\Sigma}_k\mathbf{P}/\mathbf{\Pi}_k\mathbf{P}$ a *hierarchy*—each level of the hierarchy is contained within the next, etc. The following diagram illustrates this hierarchy of containments:

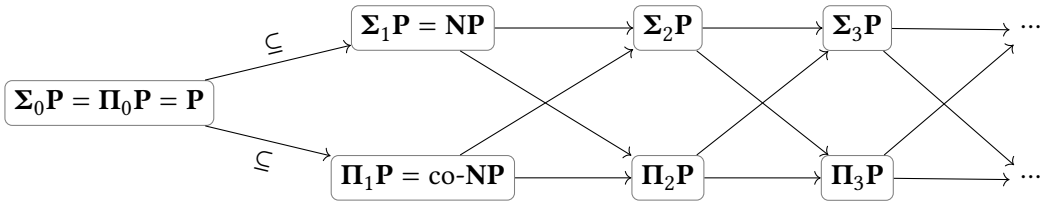


Figure 5.2. Hierarchy of inclusions in the polynomial hierarchy.

Theorem 5.3 ▸ $\Sigma_k\mathbf{P}$ and $\Pi_k\mathbf{P}$ are closed under union, intersection

If $\Pi_1, \Pi_2 \in \mathbf{\Sigma}_k\mathbf{P}$, then $\Pi_1 \cup \Pi_2$ and $\Pi_1 \cap \Pi_2$ are both in $\mathbf{\Sigma}_k\mathbf{P}$.

Likewise, if $\Pi_1, \Pi_2 \in \mathbf{\Pi}_k\mathbf{P}$, then $\Pi_1 \cup \Pi_2$ and $\Pi_1 \cap \Pi_2$ are both in $\mathbf{\Pi}_k\mathbf{P}$.

Proof. By induction on k.

- For $k = 0$, let $\Pi_1, \Pi_2 \in \mathbf{\Sigma}_0\mathbf{P} = \mathbf{\Pi}_0\mathbf{P} = \mathbf{P}$; we wish to show that $\Pi_1 \cup \Pi_2, \Pi_1 \cap \Pi_2 \in \mathbf{P}$.

  Let A_1, A_2 be polynomial-time algorithms deciding Π_1, Π_2 respectively. To decide $\Pi_1 \cup \Pi_2$, run the two algorithms in sequence, returning "yes" if at least one of the two algorithms returns "yes"; to decide $\Pi_1 \cap \Pi_2$, return "yes" if both return "yes".

  Algorithm 5.2 ▸ decider for union or intersection of two P problems

  given: an arbitrary input string $X \in \{0,1\}^*$
  $y_1 \leftarrow A_1(X)$
  $y_2 \leftarrow A_2(X)$
  if deciding $\Pi_1 \cup \Pi_2$ **then**
  return "yes" if at least one of y_1, y_2 is "yes" (i.e., $y_1 \vee y_2$)
  else if deciding $\Pi_1 \cap \Pi_2$ **then**
  return "yes" if both y_1, y_2 are "yes" (i.e., $y_1 \wedge y_2$)
  end if

  This algorithm runs in polynomial time because both A_1 and A_2 run in polynomial time; the overall running time is a sum of two polynomials (plus some constants for the last comparison), which is still a polynomial. Thus $\Pi_1 \cup \Pi_2, \Pi_1 \cap \Pi_2 \in \mathbf{P}$.

- Suppose that the claim holds for all levels below some $k \geq 1$. First, we show that $\mathbf{\Sigma}_k\mathbf{P}$ is closed under union and intersection.

  Let $\Pi_1, \Pi_2 \in \mathbf{\Sigma}_k\mathbf{P}$. Then there exist $\Pi_1', \Pi_2' \in \mathbf{\Pi}_{k-1}\mathbf{P}$ and polynomials p_1, p_2 such that

$$\Pi_1 = \{X \mid \exists Y.\ |Y| \leq p_1(|X|), \quad (X,Y) \in \Pi_1'\},$$
$$\Pi_2 = \{X \mid \exists Y.\ |Y| \leq p_2(|X|), \quad (X,Y) \in \Pi_2'\}.$$

  Define two new problems

$$\Pi_1'' = \{(X,(Y_1,Y_2)) \mid (X,Y_1) \in \Pi_1'\},$$
$$\Pi_2'' = \{(X,(Y_1,Y_2)) \mid (X,Y_2) \in \Pi_2'\}.$$

  Notice that Π_i'' is the same problem as Π_i', differing only in that it takes in and ignores an additional component in the input. Therefore, they are equivalent; $\Pi_i' \in \mathbf{\Pi}_{k-1}\mathbf{P}$ implies $\Pi_i'' \in \mathbf{\Pi}_{k-1}\mathbf{P}$ as well.

Now, construct the problems

$$\Pi_\cup = \{X \mid \exists (Y_1, Y_2).\ (X, (Y_1, Y_2)) \in \Pi_1'' \cup \Pi_2''\},$$
$$\Pi_\cap = \{X \mid \exists (Y_1, Y_2).\ (X, (Y_1, Y_2)) \in \Pi_1'' \cap \Pi_2''\}.$$

By the induction hypothesis, $\mathbf{\Pi}_{k-1}\mathbf{P}$ is closed under union and intersection, so $\Pi_1'' \cup \Pi_2'', \Pi_1'' \cap \Pi_2'' \in \mathbf{\Pi}_{k-1}\mathbf{P}$. Additionally, the length of the second component (Y_1, Y_2) is bounded by $p_1 + p_2$ (plus some constants to account for delimiters), polynomial in the size of the "board" X. Thus $\Pi_\cup, \Pi_\cap \in \mathbf{\Sigma}_k\mathbf{P}$.

Finally, we claim that $\Pi_\cup = \Pi_1 \cup \Pi_2$, and $\Pi_\cap = \Pi_1 \cap \Pi_2$. We show both equalities below.

- Claim: $\Pi_\cup = \Pi_1 \cap \Pi_2$. The following statements are equivalent:
  - $X \in \Pi_\cup$.
  - There exists (Y_1, Y_2) so that $(X, (Y_1, Y_2))$ is in either (or both) of Π_1'', Π_2''.
  - There exists Y_1 so that $(X, Y_1) \in \Pi_1'$, or there exists Y_2 so that $(X, Y_2) \in \Pi_2'$.
  - $X \in \Pi_1$ or $X \in \Pi_2$.
  - $X \in \Pi_1 \cup \Pi_2$.
- Claim: $\Pi_\cap = \Pi_1 \cap \Pi_2$. The following are equivalent:
  - $X \in \Pi_\cap$.
  - There exists (Y_1, Y_2) so that $(X, (Y_1, Y_2))$ is in both of Π_1'', Π_2''.
  - There exists Y_1 so that $(X, Y_1) \in \Pi_1'$, and there exists Y_2 so that $(X, Y_2) \in \Pi_2'$.
  - $X \in \Pi_1$ and $X \in \Pi_2$.
  - $X \in \Pi_1 \cap \Pi_2$.

This concludes the main proof: for any $\Pi_1, \Pi_2 \in \mathbf{\Sigma}_k\mathbf{P}$, both $\Pi_1 \cup \Pi_2 = \Pi_\cup$ and $\Pi_1 \cap \Pi_2 = \Pi_\cap$ are in $\mathbf{\Sigma}_k\mathbf{P}$, as desired. Thus $\mathbf{\Sigma}_k\mathbf{P}$ is closed under union and intersection.

Closure of $\mathbf{\Pi}_k\mathbf{P}$ under union and intersection follows from $\mathbf{\Pi}_k\mathbf{P} = \text{co-}\mathbf{\Sigma}_k\mathbf{P}$, and from DeMorgan's set identities:

$$(\Pi_1 \cup \Pi_2)^c = \Pi_1^c \cap \Pi_2^c, \qquad (\Pi_1 \cap \Pi_2)^c = \Pi_1^c \cup \Pi_2^c. \qquad \square$$

We may now confidently conclude, having proven these two theorems, that $3\text{COL}_k \in \mathbf{\Sigma}_k\mathbf{P}$. We discussed why earlier, but just to be thorough, we restate the full proof below.

Corollary 5.4

$3\text{COL}_k \in \mathbf{\Sigma}_k\mathbf{P}$.

Proof. By induction on k.

- For $k = 0$, we have $3\text{COL}_0 = 3\text{COLPROP}^c \in \mathbf{P} = \mathbf{\Sigma}_0\mathbf{P}$.

- For some $k \geq 1$, assume $3\text{Col}_{k-1} \in \mathbf{\Sigma}_{k-1}\mathbf{P}$. We have

$$3\text{Col}_k = \{(\Gamma, \kappa, \dots) \mid \kappa \text{ is improper}\} \cup \left\{(\Gamma, \kappa, \dots) \;\middle|\; \exists \delta.\ (\Gamma, \kappa[\delta], \dots) \in (3\text{Col}_{k-1})^{\mathrm{c}}\right\}.$$

  The first set in the union is equivalent to $3\text{ColProp}^{\mathrm{c}}$, which is in $\mathbf{P}$ and therefore also in $\mathbf{\Sigma}_k\mathbf{P}$ (theorem 5.2). The second set in the union is by construction a $\mathbf{\Sigma}_k\mathbf{P}$ problem, since $(3\text{Col}_{k-1})^{\mathrm{c}} \in \mathbf{\Pi}_{k-1}\mathbf{P}$. Thus the union of the two is also in $\mathbf{\Sigma}_k\mathbf{P}$ (theorem 5.3). □

Of course, theorems 5.2 and 5.3 are useful beyond 3Col_k; they make it much more convenient for us to construct and describe $\mathbf{\Sigma}_k\mathbf{P}/\mathbf{\Pi}_k\mathbf{P}$ problems in general. One important use-case, as exemplified by 3Col_k, is for incorporating game rules checked at *each turn* of gameplay, rather than only at the end after all turns have been played. These rules, for example, can stipulate conditions on what types of moves are valid, shortcuts to winning/losing, etc.

5.5 *k*-turn 3-Colorability is Σ_kP-complete

So, we just showed that $3\text{Col}_k \in \mathbf{\Sigma}_k\mathbf{P}$. But that's hardly surprising: given what we understand now about $\mathbf{\Sigma}_k\mathbf{P}/\mathbf{\Pi}_k\mathbf{P}$, almost anything we can conceive of as a k-turn game—that is, with polynomial-time-checkable rules, and k fixed turns of reasonable size—most likely falls within $\mathbf{\Sigma}_k\mathbf{P}/\mathbf{\Pi}_k\mathbf{P}$.

Really, the more interesting, more profound result is that 3Col_k is among the *hardest* k-turn games: it is $\mathbf{\Sigma}_k\mathbf{P}$*-complete*. Before jumping into the proof of this claim, we first discuss the key idea behind it: graph 3-colorings are powerful enough to "encode" boolean circuits.

5.5.1 Using 3-colorings to emulate circuits

Graph 3-colorings can *emulate* boolean circuits. To illustrate what this means, associate each boolean value with a color: we take the (convenient) convention that **0** means False and **1** means True (**2** is an "auxiliary" color used to enforce intermediate constraints but never to represent a boolean value). Then, it is possible to convert any boolean circuit into a graph so that properness on the graph's colorings causes them to exactly compute the circuit.

We define this idea more precisely below.

Definition 5.5 ▸ boolean 3-coloring graphs

Let Γ be a graph, and let κ be a proper *partial* coloring on Γ.

A vertex v is called a **boolean vertex** if v neighbors some vertex v' such that $\kappa(v') = 2$.

Assume Γ contains the following "special" boolean vertices:

- n distinct **input vertices** $\boldsymbol{i}_1, \dots, \boldsymbol{i}_n$;
- an **output vertex** $\boldsymbol{o}$.

We call any non-input/output vertex as a **internal vertex** of Γ.

Let κ' be an arbitrary proper total coloring that *extends* κ. For each input or output vertex v, since v neighbors a (pre-colored) **2** by construction and κ' is proper, we know $\kappa'(v) \in \{\text{True}, \text{False}\}$. We say the **boolean value** assigned by κ' to v is True if $\kappa'(v) = \mathbf{1}$, and False if $\kappa'(v) = \mathbf{0}$.

To simplify notation, we will conflate the colors **1**/**0** with their corresponding boolean values True/False(respectively), except where the distinction is needed for clarity.

Next, let $\phi : \{\text{True}, \text{False}\}^n \to \{\text{True}, \text{False}\}$ be a boolean function. We say that Γ **computes** ϕ if the following properties hold:

Attainability For every combination of boolean values $(x_1, \dots, x_n) \in \{\text{True}, \text{False}\}^n$, there exists at least one proper 3-coloring κ such that $\kappa(\boldsymbol{i}_j) = x_j$ for each $j = 1, \dots, n$.

Consistency For every proper coloring κ,

$$\kappa(\boldsymbol{o}) = \phi(\kappa(\boldsymbol{i}_1), \kappa(\boldsymbol{i}_2), \dots, \kappa(\boldsymbol{i}_n)).$$

In order to convert circuits in general to boolean graphs, we start by converting the basic building blocks of circuits. As mentioned in the definition, we represent each *wire* to a boolean vertex, i.e. a vertex joined to a pre-colored **2**, so that it can only be colored **0** or **1** (namely, a boolean value).

Figure 5.3. Representation of a wire as a boolean vertex.

To negate the boolean value of a vertex, create an edge joining the input and output vertices, which forces them to have opposite colors:

Definition 5.6 ▸ NOT **graph**

Let $\boldsymbol{i}$ be a boolean vertex. Construct a NOT **graph** on input $\boldsymbol{i}$ by introducing a new boolean vertex $\boldsymbol{o}$, i.e., the *output* vertex, and an edge $\boldsymbol{i} \leftrightarrow \boldsymbol{o}$.

Figure 5.4. Conversion of NOT gates to NOT graphs.

If, for whatever reason, we wish to replicate/propagate a boolean value across multiple vertices (called a "buffer" gate), we can use two NOT gates in a row ($\neg\neg x = x$):

Definition 5.7 ▸ **buffer graph**

Let $\boldsymbol{i}$ be a boolean vertex. Construct a **buffer graph** on input $\boldsymbol{i}$ by creating two new vertices, $\bar{i}$ and $\boldsymbol{o}$ (the *output*), and edges $\boldsymbol{i} \leftrightarrow \bar{\boldsymbol{i}} \leftrightarrow \boldsymbol{o}$.

Figure 5.5. Conversion of buffer gates to buffer graphs.

AND and OR gates are somewhat trickier to implement as graphs. To help in their construction, we first introduce an auxiliary graph that "approximates" AND/OR gates, which we call the **semi-OR graph**:

Definition 5.8 ▸ **semi-OR graph on two (boolean) vertices**

Let x, y be boolean vertices. The **semi-OR graph** on x, y is constructed as follows:

- Create three vertices x', y', t and edges joining them in a triangle.
- Create edges $x \leftrightarrow x'$ and $y \leftrightarrow y'$.
- Pre-assign t the color **1**.

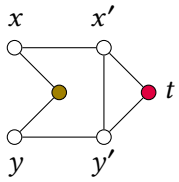


Figure 5.6. A semi-OR graph on two vertices.

The semi-OR graph is named as such because it *constrains* at least one of x, y to have the **1**; no proper coloring exists if both x, y are colored **0**. We state and prove this property below.

Lemma 5.5

Let x, y be boolean vertices, and let κ be a proper coloring of the semi-OR graph on x, y. Then it must *not* be the case that $\kappa(x) = \kappa(y) = \mathbf{0}$.

In other words, the following implications hold:

- If $\kappa(x) = \mathbf{0}$, then $\kappa(y) = \mathbf{1}$.
- If $\kappa(y) = \mathbf{0}$, then $\kappa(x) = \mathbf{1}$.

Proof. Suppose towards a contradiction that $\kappa(x) = \kappa(y) = \mathbf{0}$. Then, x' and y', which neighbor x and y respectively, cannot be colored **0**. Furthermore, x' and y' neighbor t, which

has color $\kappa(t) = \mathbf{1}$, so they also cannot be colored **1**. Therefore, both x' and y' can only be colored **2**. However, they also neighbor each other, so they must receive different colors, a contradiction. □

The semi-OR graph doesn't yet behave like a *real* OR graph because, in a real OR graph, coloring x, y both **0** should merely cause the output vertex to be colored **0**, rather than rule out proper-colorability altogether.

Multiple semi-OR graphs can be "stacked" together to impose the the same OR-like colorability constraint on three or more boolean vertices. For example, we define below the semi-OR graph on three vertices.

Definition 5.9 ▸ semi-OR graph on three vertices

Let x, y, z be boolean vertices. Construct the **semi-OR graph** on x, y, z as follows:

- Create vertices labeled x', y', t_1 and edges joining them in a triangle.
- Create edges $x \leftrightarrow x'$ and $y \leftrightarrow y'$.
- Join t_1 to another vertex pre-colored **2**, thereby making t_1 a boolean vertex. This step differs from the two-vertex semi-OR construction, wherein we would just pre-color t_1 **1**.
- Create a two-vertex semi-OR graph on t_1 and z.

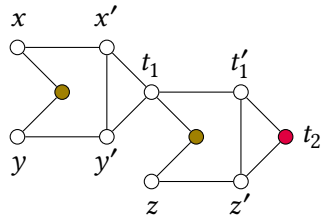


Figure 5.7. A three-vertex semi-OR graph.

Again, the three-vertex semi-OR graph constrains at least one of its inputs to be **1**.

Lemma 5.6

Let x, y, z be boolean vertices, and let κ be a proper coloring of the semi-OR graph on x, y, z. Then it is *impossible* that $\kappa(x) = \kappa(y) = \kappa(z) = \mathbf{0}$.

Proof. Suppose towards a contradiction that $\kappa(x) = \kappa(y) = \kappa(z) = \mathbf{0}$. Then x' and y', which neighbor x and y respectively, must be colored **1** or **2**. Since they also neighbor each other, they must receive different colors:

- either $\kappa(x') = \mathbf{1}$ and $\kappa(y') = \mathbf{2}$,
- or $\kappa(x') = \mathbf{2}$ and $\kappa(y') = \mathbf{1}$.

In either case, t_1, which neighbors both x' and y', can only be colored **0**. Therefore, observe that t_1 and z both have color **0**.

By construction, t_1 and z are constrained by a two-vertex semi-OR graph. Thus $\kappa(t_1) = \kappa(z) = \mathbf{0}$ violates lemma 5.5, a contradiction. □

We are now prepared to construct full-blown OR gates. To do so, we create three boolean vertices x, y, z; the plan is to use combinations of NOT graphs and semi-OR graphs to *constrain* x, y, z under the relation

$$z = x \vee y,$$

so that z corresponds to the output wire of the OR gate with inputs x and y. To do so, first observe that the condition $z = x \vee y$ is equivalent to

$$z \wedge (x \vee y) \quad \vee \quad \neg z \wedge \neg(x \vee y).$$

In other words, $z = x \vee y$ holds if and only if the values of z and of $x \vee y$ are both True or both False. Using DeMorgan's identities and distributivity properties, we derive that this condition is equivalent to

$$\begin{aligned}(z \wedge (x \vee y)) \vee (\neg z \wedge \neg(x \vee y)) &= \cancel{(z \vee \neg z)}\big((z \vee \neg(x \vee y)\big)\big((x \vee y) \vee \neg z\big)\cancel{\big((x \vee y) \vee \neg(x \vee y)\big)} \\ &= \big((z \vee (\neg x \wedge \neg y)\big)(x \vee y \vee \neg z) \\ &= (z \vee \neg x)(z \vee \neg y)(x \vee y \vee \neg z).\end{aligned}$$

Thus $z = x \vee y$ holds if and only if all three of the following conditions simultaneously hold:

- At least one of z and $\neg x$ is **1**.
- At least one of z and $\neg y$ is **1**.
- At least one of x, y, and $\neg z$ is **1**.

> ASIDE I recently learned that this conversion—from "structured" relations like $z = x \vee y$ to conjunctions of constraints like $(z \vee \neg x)(z \vee \neg y)(x \vee y \vee \neg z)$—is a well-known result called the Tseitin transformation (Tseitin 1970).
>
> I happened to derive this on my own before coming across it on Wikipedia, which I found rewarding for two reasons: first, it tells me that I, too, am capable of coming up with fame-worthy math, like the "pro" mathematicians of old; second, it reveals that these famous results are often backed by exceedingly simple ideas—in this case, basic algebra.

We construct an OR graph by using NOT graphs to compute $\neg x, \neg y, \neg z$, and then constraining them using semi-OR graphs on $z, \neg x$; $z, \neg y$; and $x, y, \neg z$.

Definition 5.10 ▸ OR **graph**

Let $\boldsymbol{i}_1, \boldsymbol{i}_2$ be boolean vertices. Construct an OR **graph** on inputs $\boldsymbol{i}_1, \boldsymbol{i}_2$ as follows:

- Introduce a new vertex $\boldsymbol{o}$, the output vertex.
- Construct NOT graphs on each of $\boldsymbol{i}_1, \boldsymbol{i}_2, \boldsymbol{o}$; name their output vertices $\bar{i}_1, \bar{i}_2, \bar{o}$, respectively.
- Construct two-vertex semi-OR graphs on $\boldsymbol{o}, \bar{i}_1$ and on $\boldsymbol{o}, \bar{i}_2$.

- Construct a three-vertex semi-OR graph on $\boldsymbol{i}_1, \boldsymbol{i}_2, \bar{o}$.

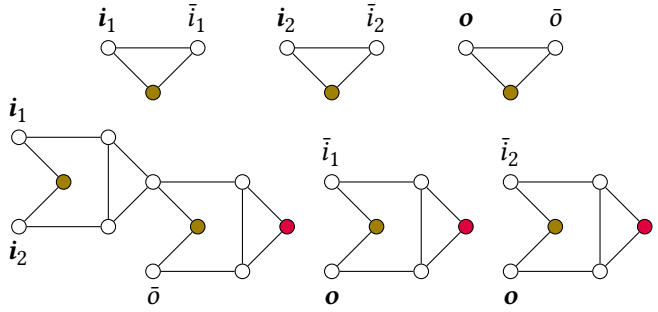


Figure 5.8. An OR graph. For readability, some vertices are drawn as duplicates, with copies sharing the same label to indicate that they refer to the same vertex.

Aside Just for fun, here's what the fully-assembled OR graph looks like. This isn't *exactly* the same graph as the one above: some 2 vertices have been added/elided/rearranged, but the functional structure is otherwise the same.

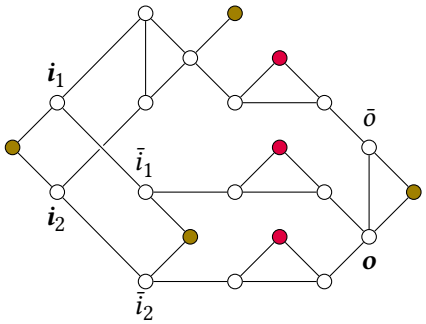


Figure 5.9. A fully-assembled OR graph (drawn without duplicated boolean vertices).

Finally, to construct an AND graph, we leverage DeMorgan's identity to represent AND operations in terms of NOT and OR operations, which we already know how to implement:

$$z = x \wedge y \qquad \iff \qquad \neg z = \neg x \vee \neg y.$$

Therefore we construct an AND graph simply by switching the labels x, y, z with $\neg x, \neg y, \neg z$ in the construction of the OR graph.

Definition 5.11 ▸ AND **graph**

Let $\boldsymbol{i}_1, \boldsymbol{i}_2$ be boolean vertices. Construct an AND **graph** on inputs $\boldsymbol{i}_1, \boldsymbol{i}_2$ as follows:

- Introduce a new vertex $\boldsymbol{o}$, the output vertex.
- Construct NOT graphs on each of $\boldsymbol{i}_1, \boldsymbol{i}_2, \boldsymbol{o}$; name their output vertices $\bar{i}_1, \bar{i}_2, \bar{o}$, respectively.

- Construct two-vertex semi-OR graphs on $\bar{o}, \boldsymbol{i}_1$ and $\bar{o}, \boldsymbol{i}_2$.
- Construct a three-vertex semi-OR graph on $\bar{i}_1, \bar{i}_2, \boldsymbol{o}$.

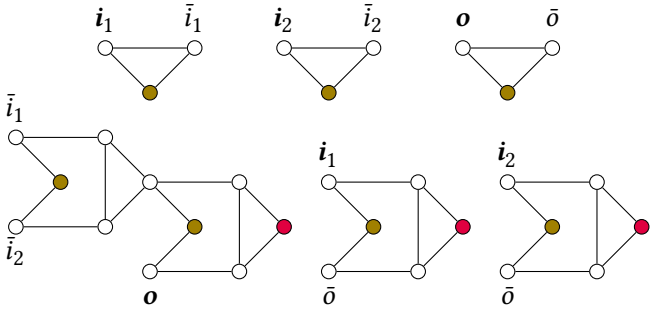


Figure 5.10. An AND graph.

ASIDE I did it for the OR graph, so I might as well also do it for the AND graph:

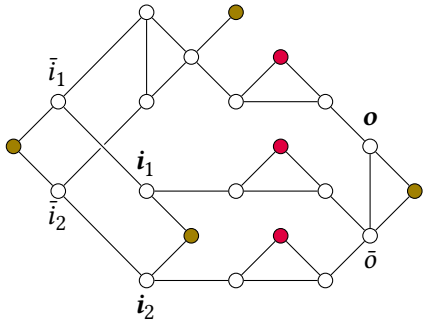


Figure 5.11. A fully-assembled AND graph.

Thus we have defined graph encodings for all three logic gates. We summarize this discussion by proving the correctness of these constructions, according to definition 5.5.

Theorem 5.7

[1] The NOT graph (definition 5.6) computes the function $\phi(x) = \neg x$.
[2] The OR graph (definition 5.10) computes the function $\phi(x, y) = x \vee y$.
[3] The AND graph (definition 5.11) computes the function $\phi(x, y) = x \wedge y$.

Proof. In each case, we wish to show that for each combination of input boolean values, the following properties hold:

Attainability There exists a proper total coloring κ of the graph such that κ assigns those values to the input vertices.

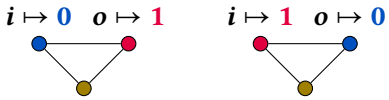
Consistency Every coloring under this input combination assigns the correct boolean value

to the output vertex.

We prove each part separately.

[1] Let $\boldsymbol{i}$ be the input vertex and $\boldsymbol{o}$ be the output vertex of a NOT graph.

Attainability Both input settings are attained by some proper coloring:

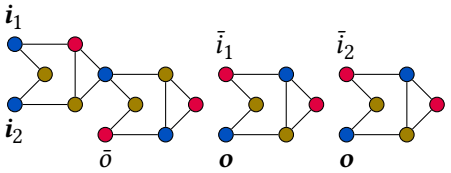
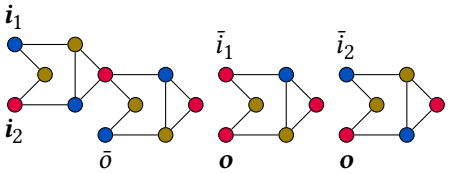
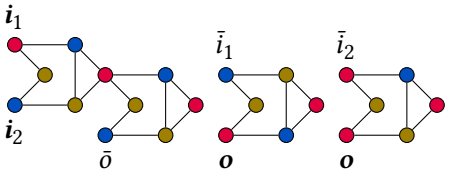
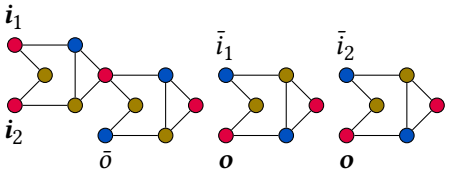


Consistency We claim that the colorings shown above are the only possible proper colorings (and therefore all proper colorings assign the same output values). This is straightforward to see. Since $\boldsymbol{i}$ and $\boldsymbol{o}$ are both boolean vertices (they are connected to a 2), they can only take colors in $\{0, 1\}$. Then, since $\boldsymbol{i}$ and $\boldsymbol{o}$ are joined to each other by an edge, their colors must be opposite, so for every proper coloring κ, we have $\kappa(\boldsymbol{o}) = \neg\kappa(\boldsymbol{i})$, as desired.

Thus the NOT graph indeed computes $\phi(x) = \neg x$.

[2] Let $\boldsymbol{i}_1, \boldsymbol{i}_2, \boldsymbol{o}$ be boolean vertices such that $\boldsymbol{o}$ is the output vertex of an OR graph on input vertices $\boldsymbol{i}_1$ and $\boldsymbol{i}_2$.

Attainability As shown in the diagrams below, each input combination is properly attainable (this time, the colorings for each combination are not necessarily unique). In each diagram, the NOT-graphs joining $x/\neg x$, $y/\neg y$, and $z/\neg z$ have been omitted to avoid visual cluttering; assume that they are implicitly present.

$i_1 \mapsto x_1$	$i_2 \mapsto x_2$	$o \mapsto x_1 \vee x_2$	coloring
False	False	False	
False	True	True	
True	False	True	
True	True	True	

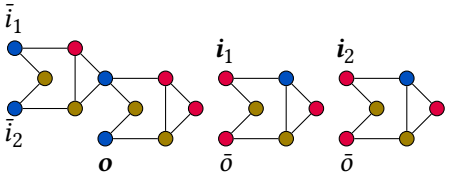
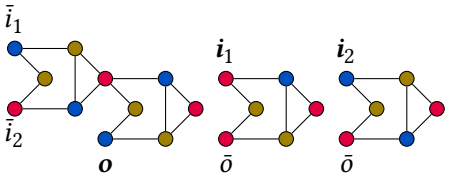
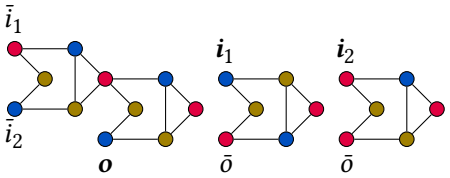
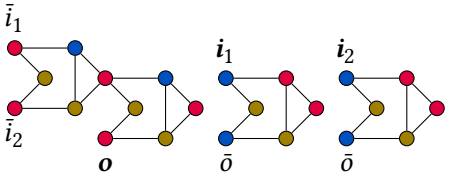
Consistency Let κ be an arbitrary proper coloring of the OR graph. We wish to show that $\kappa(o) = \kappa(i_1) \vee \kappa(i_2)$ always holds. In other words, we will argue that $\kappa(o) =$ True if and only if at least one of $\kappa(i_1), \kappa(i_2)$ is True.

[⇒] Suppose that $\kappa(o) =$ True. Then consistency of the NOT graph implies that $\kappa(\bar{o}) =$ False. By construction of the OR graph, the vertices $i_1, i_2, \bar{o}$ are constrained by a three-vertex semi-OR, which necessitates that at least one of $i_1, i_2, \bar{o}$ is colored True. Since $\bar{o}$ isn't, that means at least one of i_1, i_2 must be colored True, as desired.

[⇐] Suppose that at least one of i_1, i_2 is colored True; assume without loss of generality that $\kappa(i_1) =$ True. Consistency of the NOT graph implies $\kappa(\bar{i}_1) =$ False. There is a two-vertex semi-OR constraining $\bar{i}_1$ and o, so at least one of them must be colored True; $\bar{i}_1$ isn't, leaving $\kappa(o) =$ True, as desired.

[3] Let i_1, i_2, o be boolean vertices such that o is the output vertex of an AND graph on input vertices i_1 and i_2.

Attainability Each input combination is attainable, per the diagrams below (again, not necessarily unique):

$i_1 \mapsto x_1$	$i_2 \mapsto x_2$	$o \mapsto x_1 \wedge x_2$	coloring
False	False	False	
False	True	False	
True	False	False	
True	True	True	

Consistency Note that the AND graph is constructed by swapping each boolean vertex i_1, i_2, o with its negation $\neg i_1, \neg i_2, \neg o$, respectively. Thus consistency of the AND graph follows from consistency of the OR graph (proven above) and DeMorgan's identities: for any proper coloring κ,

$$\kappa(\bar{o}) = \kappa(\bar{i}_1) \vee \kappa(\bar{i}_2) \iff \neg\kappa(o) = \neg\kappa(i_1) \vee \neg\kappa(i_2) \iff \kappa(o) = \kappa(i_1) \wedge \kappa(i_2). \qquad \square$$

Finally, any circuit can now be converted to a boolean 3-coloring graph computing it, by converting each gate in the circuit to its corresponding graph. We define this construction exactly below.

Definition 5.12 ▸ boolean graph of a circuit

Let C be a boolean circuit. The **boolean graph of** C, denoted Γ_C, is constructed as follows:

- For each wire w in C, construct a corresponding boolean vertex v_w.

  If w is an input wire of C, then label v_w an input vertex of Γ_C. Similarly, if w is the output wire, then label v_w the output vertex of Γ_C.

- For each NOT gate with input wire w_1 and output wire w_2, construct a NOT graph (definition 5.6) on input vertex v_{w_1} and output vertex v_{w_2}.

- For each AND/OR gate with input wires w_1, w_2 and output wire w_3, construct a corresponding AND/OR graph (definitions 5.10 and 5.11) on input vertices v_{w_1}, v_{w_2} and output vertex v_{w_3}.

For each logic gate g in C, let Γ_g denote its corresponding subgraph in Γ_C.

First, observe that this definition directly describes an algorithm that takes a circuit C and returns its boolean graph Γ_C. This algorithm runs in polynomial-time, because constructing each subgraph "component" (boolean vertices, logic-gate subgraphs) incurs a constant amount of work (plus some polynomial-time bookkeeping), and the total number of component-construction iterations is bounded by the size of C itself.

Next, we prove that Γ_C correctly computes C; this result follows straightforwardly from correctness of each logic-gate subgraph (theorem 5.7).

Theorem 5.8

Let C be a boolean circuit, and let Γ_C be the boolean graph of C (definition 5.12). Then Γ *computes* (definition 5.5) C.

Proof. Assume that the input wires of C are $w_1^*, \dots, w_n^*$, and that the corresponding boolean vertices in Γ_C are $\boldsymbol{i}_j = v_{w_j^*}$ for each $j = 1, \dots, n$.

We wish to show that any input combination $X \in \{\text{True}, \text{False}\}^n$ is attainable by some proper coloring on Γ_C, and that all proper colorings generate outputs consistent with C.

Attainability Let $X \in \{\text{True}, \text{False}\}^n$ be arbitrary. We describe a procedure to construct a proper coloring on Γ_C attaining X on the input vertices.

Initially, for each wire w in C, its boolean vertex v_w is uncolored; correspondingly, mark w as "unvisited". Start by assigning X, as colors, to the input vertices $\boldsymbol{i}_1, \dots, \boldsymbol{i}_n$, and correspondingly mark each input wire $w_1^*, \dots, w_n^*$ as "visited".

Next, as long as there remain unvisited wires in C, repeatedly color logic-gate subgraphs in Γ as follows. C contains no cyclic dependencies, so there must exist at least one gate g with already-visited input wire(s) w_i and unvisited output wire w'. Then the corresponding vertex/vertices v_{w_i} is/are colored, while $v_{w'}$ remains uncolored. By consistency of each logic-gate subgraph (theorem 5.7), there exists a proper coloring of Γ_g extending the pre-existing coloring on v_{w_i}; assign those colors to Γ_g, and mark the wire w' as visited. This assignment preserves properness of the partial coloring because, by construction of Γ_C, there are never any edges joining the internal or output vertices across different logic-gate subgraphs, so we may freely color Γ_g without worrying about introducing improper edges with neighbors *outside* Γ_g. By induction on the number of visited wires, this procedure preserves properness at each iteration, thereby resulting in a proper total coloring of Γ_C.

Consistency Consistency of each logic-gate subgraph (theorem 5.7) implies that the boolean

relations at each logic gate in C are exactly matched by the boolean colorings at each logic-gate subgraph in Γ_C. Thus by induction on the structure of C, the output vertex in Γ_C must always have the same boolean value as the output wire in C. □

The takeaway here, intuitively, is that 3-coloring graphs are as computationally powerful as circuits are. If any algorithm can be encoded as a sequence of circuit computations, and circuits can be embedded in graph 3-colorings, then effectively, any algorithm is essentially a graph 3-coloring computation—with data stored as colors, and logic being carried out by properness constraints. Therefore, we would be unsurprised to find, as we discuss in the next subsection, that every circuit game is also just a graph 3-coloring game, under the appropriate translations of terminology/game-rules.

5.5.2 Translating Circuit Satisfiability games to 3-Colorability games

Recall that in a CircSat$_k$ game, two players alternate turns assigning inputs to a circuit, with victory decided by the final output of the circuit: True means the first player wins; False means the second player wins. We now wish to encode these objectives in the language of 3-colorings and properness.

To start with, we are given a circuit C, with inputs partitioned into k groups, $I_1, \dots, I_k$. Naturally, we start by converting C to a boolean graph Γ_C, mapping the groups of circuit inputs directly to groups of vertices,

$$U_i = \{v_w \mid w \in I_i\}$$

for $i \in \{1, 2, \dots, k\}$. So far, this doesn't define a valid 3Col$_k$ game yet, as $U_1, \dots, U_k$ don't partition all vertices in Γ_C—the uncolored internal vertices are unaccounted for.

In choosing who colors the remaining vertices, we wish to ensure that, whatever coloring gets assigned to the internals of Γ_C, it should be consistent with the wire values in C, so that theorem 5.8 applies. The easiest way to do so is to wait until all *input* vertices have been colored before coloring the internal vertices. To this end, we place all internal vertices in the last group, U_k. Of course, in the special case that $k = 0$, there are no turns to be played, so we include the full coloring of the graph (computable by simply evaluating the circuit and filling in the proper colorings at each gate, per theorem 5.7) in the pre-coloring.

Finally, we translate the winning condition as follows. In CircSat$_k$, the winning condition is that the *first* player wins if and only if the final output is True. Meanwhile, in 3Col$_k$, the winning condition is that the *last* player wins if and only if all turns finish, resulting in a proper total coloring. The asymmetry between CircSat$_k$ and 3Col$_k$ is the difference in who wins according to the parity of the number of turns:

- The first player plays on odd-numbered turns, so when k is odd, the first player is the last player. Thus for odd k the winning incentive of the first player in CircSat$_k$, should exactly match that of the last player in 3Col$_k$: a proper total coloring should be obtained if and only if the circuit finally outputs True.

- When k is even, the second player is the last player. Thus for even k the winning incentive of the first player should be opposite that of the last player: the final coloring is proper if and only if the circuit outputs False.

In all cases, we enforce the winning condition in all cases by appending k NOT-graphs to the output vertex of the graph and pre-coloring the *final* output vertex to False.

We summarize this *reduction* from CIRCSAT$_k$ to 3COL$_k$, including the steps of the circuit-to-graph conversion (definition 5.12), in the algorithm below.

Algorithm 5.3 ▸ **reduction from CIRCSAT$_k$ to 3COL$_k$**

given: a circuit C with inputs partitioned as $I_1 \sqcup \cdots \sqcup I_k$
▷ *construct the boolean graph of C* ◁
create a blank graph Γ and partial-coloring κ
introduce (in Γ) a special vertex v_2
pre-color $\kappa(v_2) = 2$
for each wire w in C **do**
  introduce a vertex v_w
  introduce an edge $v_w \leftrightarrow v_2$
end for
for each logic gate g in C **do**
  if g is a NOT gate with input wire w_1 and output wire w_2 **then**
    add to Γ a NOT graph on input v_{w_1} and output v_{w_2}, following definition 5.6
  else if g is an AND gate with inputs w_1, w_2 and output w_2 **then**
    add to Γ an AND graph on inputs v_{w_1}, v_{w_2} and output v_{w_3}, per definition 5.11
  else if g is an OR gate with inputs w_1, w_2 and output w_2 **then**
    add to Γ an OR graph on inputs v_{w_1}, v_{w_2} and output v_{w_3}, per definition 5.10
  end if
end for
if $k = 0$ **then**
  ▷ *no turns, pre-color all vertices* ◁
  for each logic gate g in C **do**
    evaluate g and color Γ_g accordingly, per theorem 5.7
  end for
else
  ▷ *partition uncolored vertices by turn* ◁
  $U_1 \leftarrow \{\}; U_2 \leftarrow \{\}; \ldots; U_k \leftarrow \{\}$
  for each $i \in \{1, 2, \ldots, k\}$ **do**
    for each wire $w \in I_i$ **do**
      add v_w to U_i
    end for
  end for
  for each remaining (unused) vertex v in Γ **do**
    if v is not pre-colored **then**

```
            add v to U_k
        end if
    end for
end if
▷ encode parity of winning condition via k NOT graphs ◁
w ← output wire of C
for each i = 1, 2, ..., k do
    introduce a new vertex w′
    introduce an edge w′ ↔ v_2
    introduce an edge w ↔ w′
    w ← w′
end for
return (Γ, κ, (U_1, ..., U_k))
```

We prove the correctness of this reduction below, thereby showing that $\textsc{CircSat}_k \leq 3\textsc{Col}_k$.

Theorem 5.9 ▸ $\textsc{CircSat}_k \leq 3\textsc{Col}_k$

For any $\textsc{CircSat}_k$ instance $(C, (I_1, \ldots, I_k))$, let $(\Gamma, \kappa, (U_1, \ldots, U_k))$ denote the $3\textsc{Col}_k$ instance obtained by running algorithm 5.3 on $(C, (I_1, \ldots, I_k))$; then

$$(C, (I_1, \ldots, I_k)) \in \textsc{CircSat}_k \iff (\Gamma, \kappa, (U_1, \ldots, U_k)) \in 3\textsc{Col}_k.$$

Proof. By induction on k.

- In the base case, $k = 0$, the $\textsc{CircSat}_0 = \textsc{CircVal}$ instance has no inputs (all inputs are already specified as constants). Thus the resulting $3\textsc{Col}_0 = 3\textsc{ColProp}^c$ instance has no uncolored vertices; (Γ, κ) is a totally-colored graph. Since $k = 0$ additional parity-adjusting NOT-graphs are appended at the output, Γ is exactly same as the boolean graph Γ_C of C.

  By construction of the pre-coloring, κ is internally consistent. Thus theorem 5.8 implies that the output color is forced, under properness, to be same as the output value of C. But by construction the output is pre-colored **0**, so the coloring κ is proper if and only if C outputs False to match that pre-coloring. Equivalently, κ is *improper* if and only if C outputs True, or

  $$(\Gamma, \kappa) \in 3\textsc{ColProp}^c \iff C \in \textsc{CircVal},$$

  as desired.

- Assume $k > 1$. We show both directions of the implication separately.

  [$\Rightarrow$] Suppose that $(C, (I_i)) \in \textsc{CircSat}_k$; we wish to show that $(\Gamma, \kappa, (U_i)) \in 3\textsc{Col}_k$.

  $(C, (I_1, \ldots, I_k)) \in \textsc{CircSat}_k$ means that there exists $X \in \{\text{True}, \text{False}\}^{|I_1|}$ such that $\neg C[I_1 := X]$, with remaining inputs partitioned as $I_2, \ldots, I_k$, is in $(\textsc{CircSat}_{k-1})^c$.

Construct $\delta : U_1 \to \{\mathbf{0}, \mathbf{1}, \mathbf{2}\}$ by interpreting the boolean-value assignment $I_1 := X$ as an assignment of colors to U_1:

$$\delta(v_{w_i}) = \begin{cases} \mathbf{1} & x_i = \text{True}, \\ \mathbf{0} & x_i = \text{False} \end{cases}.$$

Then, observe that the remaining $(k-1)$-turn game instance, comprising Γ, the augmented coloring $\kappa[\delta]$, and with uncolored vertices partitioned as $U_2, \dots, U_k$, is *exactly* same as that obtained by the reduction algorithm 5.3 on $(\neg C[I_1 := X], (I_2, \dots, I_k))$. To see this, recall that:

- In the k-turn game, Γ is formed by appending k NOT-graphs to Γ_C.
- In the $(k-1)$-turn game, Γ is formed by appending $k-1$ NOT-graphs to $\Gamma_{\neg C}$.

In both instances, the original circuit C gets negated a total of k additional times.

Therefore, by induction,

$$(\neg C[I_1 := X], (I_2, \dots, I_k)) \in (\textsc{CircSat}_{k-1})^{\mathrm{c}} \iff (\Gamma, \kappa[\delta], (U_2, \dots, U_k)) \in (3\textsc{Col}_{k-1})^{\mathrm{c}}.$$

Consequently, δ is indeed a winning move for the original k-turn game; it *certifies* that

$$(\Gamma, \kappa, (U_1, \dots, U_k)) \in 3\textsc{Col}_k,$$

as desired.

[$\Leftarrow$] Suppose that $(\Gamma, \kappa, (U_i)) \in 3\textsc{Col}_k$; we wish to show that $(C, (I_i)) \in \textsc{CircSat}_k$.

$(\Gamma, \kappa, (U_i)) \in 3\textsc{Col}_k$ means that either κ is improper, or there exists a "winning" coloring $\delta : U_1 \to \{\mathbf{0}, \mathbf{1}, \mathbf{2}\}$ such that

$$(\Gamma, \kappa[\delta], (U_2, \dots, U_k)) \in (3\textsc{Col}_{k-1})^{\mathrm{c}}.$$

By construction, κ is proper, because the only pre-colored vertices are the special vertex $v_{\mathbf{2}} \mapsto \mathbf{2}$ and the output vertex $\boldsymbol{o} \mapsto \mathbf{0}$. Thus it must be that a winning δ exists.

Note that if $\kappa[\delta]$ were improper, we would immediately have

$$(\Gamma, \kappa[\delta], (U_2, \dots, U_k)) \in 3\textsc{Col}_{k-1}$$

(the responding player automatically wins). But by assumption, that is not the case: we actually have

$$(\Gamma, \kappa[\delta], (U_2, \dots, U_k)) \in (3\textsc{Col}_{k-1})^{\mathrm{c}}$$

(δ leads to a guaranteed *loss* for the responding player). Thus $\kappa[\delta]$ must be proper.

Therefore, for each vertex $\boldsymbol{i}_j \in U_1$, we know that $\kappa[\delta]$ only assigns colors **0** or **1** to $\boldsymbol{i}_j$ since by construction $\boldsymbol{i}_j$ neighbors v_2, which is pre-colored **2**. Thus construct a boolean assignment $X = \left(x_1, \dots, x_{|I_1|}\right) \in \{\text{True}, \text{False}\}^{|I_1|}$ by

$$x_j = \begin{cases} \text{True} & \delta(\boldsymbol{i}_j) = \mathbf{1} \\ \text{False} & \delta(\boldsymbol{i}_j) = \mathbf{0} \end{cases}.$$

As before, note that $(\Gamma, \kappa[\delta], (U_2, \dots, U_k))$ is exactly the same $(k-1)$-turn instance as obtained by algorithm 5.3 on $(\neg C[I_1 := X], (I_2, \dots, I_k))$. Thus by induction

$$(\Gamma, \kappa[\delta], (U_2, \dots, U_k)) \in (3\text{Col}_{k-1})^{\text{c}}$$

implies that

$$(\neg C[I_1 := X], (I_2, \dots, I_k)) \in (\text{CircSat}_{k-1})^{\text{c}},$$

and therefore X certifies that

$$(C, (I_1, \dots, I_k)) \in \text{CircSat}_k,$$

as desired. □

Finally, the correctness of this reduction implies the big theorem of this chapter: $\boldsymbol{\Sigma}_k\mathbf{P}$-completeness of 3Col_k games.

Theorem 5.10 ▸ yay!

3Col_k is $\boldsymbol{\Sigma}_k\mathbf{P}$-hard. Together with corollary 5.4, this implies that 3Col_k is $\boldsymbol{\Sigma}_k\mathbf{P}$-complete.

5.6 Can we avoid pre-coloring vertices?

Yes! For sake of brevity, we give no formalisms/proofs in this chapter; instead, we focus only on the intuition of why un-colored graphs, despite seeming more limited at first glance, are basically equivalent to pre-colored graphs. That is, the choice to work with pre-colored graphs makes virtually no practical difference; it is only a convenient device to simplify discourse.

At the start of this chapter, we assumed a specific convention for representing boolean values as colors: True is **1** and False is **0**, leaving **2** unused. Aside from being intuitively convenient, this convention is entirely arbitrary. In the context of proper 3-colorings, none of the three colors are special in any way; properness/improperness is preserved under any permutation of colors.

Therefore, when we start with an uncolored graph, there is no association *a priori* between specific colors and boolean values. However, an association is *induced* as soon as some vertices are colored in. For example, whatever color is received by the special v_2 vertex represents the "not-a-boolean" color. It doesn't matter what exactly that color is—red, yellow, green—by permuting the names of the colors, we can always *call* that color **2**, without loss of generality. Similarly, the AND-graphs

and OR-graphs link to a pre-colored **1** vertex. Here, that vertex starts out uncolored, but as soon as it becomes colored, we call that color **1**. Finally, we call the last remaining color **0**.

To ensure that these induced pre-coloring associations are consistent—e.g., that **1** and **2** don't accidentally coincide—we start out by constructing a special triangle comprising vertices $v_{\mathbf{0}}, v_{\mathbf{1}}, v_{\mathbf{2}}$. Because these special vertices are joined by a triangle, they must receive all three distinct colors (under any proper coloring). Regardless of which specific hues the players choose to color this triangle, we simply call the colors of this triangle by the names **0**, **1**, **2**, respectively.

In the original construction, wherever we would have made connections to a pre-colored vertex, we instead make connections to one of the special vertices instead. Finally, in $3\textsc{Col}_k$ games, we ensure that the triangle actually functions as a *pre*-coloring, by including special vertices in the first group of vertices U_1, so that they are the first to be colored, and subsequent properness constraints imply that all other colorings must be consistent with the special pre-colorings.

Chapter 6

Conclusion

In this thesis, I explore the computational complexity of fixed-turn games under the *polynomial hierarchy*, with $\mathbf{\Sigma}_k\mathbf{P}$ representing the class of "can the first player guarantee a win?" decision problems for all games with k turns, and $\mathbf{\Pi}_k\mathbf{P}$ representing the class of complement decision problems, "is the first player doomed to lose?". Using the foundational Circuit Satisfiability games as a starting point, we search for other $\mathbf{\Sigma}_k\mathbf{P}/\mathbf{\Pi}_k\mathbf{P}$-*complete* games—games that are maximally hard for each class and are therefore ideal representatives characterizing the difficulty of each class. Finally, we introduce the k-turn Graph 3-Colorability games on graphs, and by embedding circuits in graph 3-colorings show that k-turn 3-Colorability is $\mathbf{\Sigma}_k\mathbf{P}$-complete.

Unfortunately, in a mere half year of thesis (having spent the first half figuring out only *what* question to explore), I had time to include in my thesis the full treatment of only one flavor of game—3-Colorability—even though I am absolutely confident there are tons more.

For instance, consider the Exact Set Covering problem, which I like to informally call the "sushi problem": you run an eccentric sushi restaurant serving n distinct flavors of sushi, numbered $1, \dots, n$; your restaurant's menu contains a list of *combos*, each a subset of $\{1, \dots, n\}$; is there a way to order certain combos such that each sushi flavor is included *exactly* once? This is a well-known **NP**-complete problem, and it can be straightforwardly extended into a k-turn game by partitioning the combos into k groups and having players take turns choosing whether or not to order each combo, subject at each turn to the "properness" constraint that ordering a previously-ordered sushi flavor causes immediate defeat. There is a straightforward way to map logic gates to sushi flavors and combos, and this game is therefore $\mathbf{\Sigma}_k\mathbf{P}$-complete—though there is no room for me to detail this treatment here.

Moreover, my success with these two problems leads me to suspect that there should be a straightforward way to extend almost any well-known **NP**-complete *puzzle* into interesting $\mathbf{\Sigma}_k\mathbf{P}$-complete games. To this end, I can think of two overarching future directions for this work:

- Explore a ton of **NP**-complete problems (perhaps starting with Karp's famous 21 (Karp 1972)), and try to see how naturally they extend to $\mathbf{\Sigma}_k\mathbf{P}$-complete multi-turn games.
- Drawing on commonalities between the 3-colorability games explored in chapter 5 and the

sushi games sketched above, come up with a general framework for lifting 1-turn puzzles into k-turn games.

For instance, both the 3-colorability games and the sushi games share a *properness* constraint: in 3-colorings, it is that no two neighboring vertices may share a color; in sushi orders, it is that no two selected combos may share a flavor. Likewise, both games have a condition indicating the total completion of the game: in 3-colorings, it is totality of the coloring after all vertices have been filled in; in sushi orders, it is that each flavor is ordered at least once. Together, totality *and* properness result in successful "emulation" of circuits.

Perhaps there is some way to define general notions of *turn*, *properness*, *totality*, and *emulation* in order to streamline the process of showing $\mathbf{\Sigma}_k\mathbf{P}$-completeness of games.

Finally, to recap it all, the most important question surrounding this thesis (and really any work) is, why is it interesting? To this, I say: everybody knows about **P**-vs-**NP**—the pervasion of **NP**-completeness among interesting real-world puzzles makes the **P**-vs-**NP** question extremely impactful, both theoretically and practically. Taking the perspective of this thesis, we view **P** as the class of 0-turn games and **NP** the class of 1-turn games; why stop at **P**-vs-**NP**? Why not ask about **NP**-vs-$\mathbf{\Sigma}_2\mathbf{P}$? $\mathbf{\Sigma}_2\mathbf{P}$-vs-$\mathbf{\Sigma}_3\mathbf{P}$? The central question really boils down (or boils *up*, perhaps?) to, *does the number of turns in a game make a difference*? The answers to this question, at *every* level, not just 0-vs-1, are worth exploring, and together, they give resounding insight on the general structure of puzzles and games.

Bibliography

Beaulieu, G., K. Burke, and E. Duchêne (May 13, 2013). "Impartial coloring games". In: *Theoretical Computer Science* 485, pp. 49–60. DOI: 10.1016/j.tcs.2013.02.032.

Bodlaender, Hans L. (1991). "On the complexity of some coloring games". In: *WG 1990: Graph-Theoretic Concepts in Computer Science*. Vol. 484, pp. 30–40. DOI: 10.1007/3-540-53832-1_29.

Burke, Kyle and Robert A. Hearn (2019). "PSPACE-complete two-color planar placement games". In: *International Journal of Game Theory* 48, pp. 393–410. DOI: 10.1007/s00182-018-0628-8.

Cook, Stephen A. (May 1971). "The complexity of theorem-proving procedures". In: *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*, pp. 151–158. DOI: 10.1145/800157.805047.

Costa, Eurinardo et al. (Aug. 30, 2019). "PSPACE-hardness of Two Graph Coloring Games". In: *Electronic Notes in Theoretical Computer Science* 346, pp. 333–344. DOI: 10.1016/j.entcs.2019.08.030.

Gasarch, William I. (June 2002). "The P=?NP poll". In: *ACM SIGACT News* 33 (2), pp. 34–47. DOI: 10.1145/564585.564599.

Karp, Richard M. (1972). "Reducibility among Combinatorial Problems". In: *Complexity of Computer Computations*, pp. 85–103. ISBN: 978-1-4684-2001-2. DOI: 10.1007/978-1-4684-2001-2_9.

Levin, Leonid A. (1973). "Universal search problems". Trans. Russian by Boris A. Trakhtenbrot. In: *Problems of Information Transmission* 9 (3), pp. 115–116. DOI: 10.1109/MAHC.1984.10036.

Papadimitriou, Christos H. (1993). *Computational Complexity*. University of California – San Diego: Addison-Wesley. ISBN: 0-201-53082-1.

Schaefer, Thomas J. (Apr. 1978). "On the complexity of some two-person perfect-information games". In: *Journal of Computer and System Sciences* 16 (2), pp. 185–225. DOI: 10.1016/0022-0000(78)90045-4.

Stockmeyer, Larry J. (Oct. 1976). "The polynomial-time hierarchy". In: *Theoretical Computer Science* 3 (1), pp. 1–22. DOI: 10.1016/0304-3975(76)90061-X.

Tseitin, Gregory S. (1970). "On the complexity of derivation in propositional calculus". In: *Studies in Constructive Mathematics and Mathematical Logic, Part II*. Steklov Mathematical Institute, pp. 115–125.

Wrathall, Celia (Oct. 1976). "Complete sets and the polynomial-time hierarchy". In: *Theoretical Computer Science* 3 (1), pp. 23–33. DOI: 10.1016/0304-3975(76)90062-1.