Gottfried Wilhelm
Leibniz Universität Hannover
Fakultät für Elektrotechnik und Informatik
Institut für Theoretische Informatik

An Overview of Independence Results of Set Theory and Open Mathematical Problems in Computational Complexity

Bachelorarbeit

im Studiengang Informatik

von

Ahmed Habib Mahjoub

Prüfer: Prof. Dr. Heribert Vollmer
Zweitprüfer: PD Dr. Arne Meier
Betreuer: Prof. Dr. Heribert Vollmer

Hannover, 11. November 2021

Erklärung der Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig und ohne fremde Hilfe verfasst und keine anderen als die in der Arbeit angegebenen Quellen und Hilfsmittel verwendet habe. Die Arbeit hat in gleicher oder ähnlicher Form noch keinem anderen Prüfungsamt vorgelegen.

Sousse, Tunesien, den 11. November 2021

_____

Ahmed Habib Mahjoub

Abstract

A quite alluring part of theoretical computer science is dealing with the existence of non-computable functions. In other words, problems for which there are no terminating algorithms. Thanks to Church's thesis, we can safely affirm that being computable is equivalent to being Turing-computable. Turing machines are the universal model of computation as they can translate any computing the human brain could perform. In this regard, a famous non-Turing-computable problem that has eluded many mathematicians is the function devised by Tibor Radó, known as the Busy Beaver function in the modern era. The latter is exceptionally rapidly growing so that it even exceeds the fast-growing Ackermann function. An interesting question regarding Busy Beaver is to determine the smallest value of n for which $\mathbf{BB(n)}$ is independent of the modern axioms of mathematics like $\mathcal{ZFC}$. As an opening, we present an introduction to Busy Beaver and set theory. In the second chapter, we describe the programming language named Laconic developed by Adam Yedidia, with which he determines an upper bound to the question raised above. We also cite an overview of another related language serving the same purpose and producing enhanced results. Furthermore, we use Laconic to implement a few open mathematical problems both in number theory and graph theory, thereby proving the undecidability of several Busy Beaver values. Lastly, we tackle the famous and strenuous **P vs. NP** problem by demonstrating the adversity of proving its independence from $\mathcal{ZF}$ set theory.

Acronyms

- **Formal systems**:
  - $\mathcal{ZFC}$ Zermelo-Fraenkel Set Theory with the Axiom Of Choice
  - $\mathcal{ZF}$ Zermelo-Fraenkel Set Theory without the Axiom of Choice
  - $\mathcal{PA}$ Peano Arithmetic
  - $\mathcal{SRP}$ Stationary Ramsey Property with the axioms of $\mathcal{ZFC}$
- **BB** Busy Beaver
- **TMD** Turing Machine Descriptor
- **DAG** Directed Acyclic Graph
- **NQL** Not Quite Laconic

Contents

1. Introduction **1**

1.1. Motivation and Set Theory 1

1.2. Busy Beaver 3

1.2.1. Turing Machine 3

1.2.2. Non-Computability of the Busy Beaver Function 5

2. The Laconic Programming Language **9**

2.1. Concept and Compilation 9

2.1.1. Motivation 9

2.1.2. From Laconic Code to Parsimonious Turing Machine 10

2.2. Application Examples 16

2.2.1. The Friedman's Conjecture in Set Theory 16

2.2.2. The Lychrel Numbers 20

2.2.3. The Scholz Conjecture 21

2.2.4. The Lehmer's Totient Problem 25

2.2.5. The Total Coloring Conjecture 25

2.2.6. The Erdős–Gyárfás Conjecture 29

2.3. Related Work: Not Quite Laconic 34

3. P vs. NP and Formal Independence **37**

3.1. The Relativization Barrier 39

3.2. The Natural Proofs Barrier 41

4. Summary and Outlook **45**

A. Appendices **47**

A.1. Laconic Program: The Scholz Conjecture 47

A.2. Laconic Program: The Lychrel Numbers 50

A.3. Laconic Program: The Lehmer's Totient Problem 50

A.4. Laconic Program: The Total Coloring Conjecture 50

A.5. Laconic Program: The Erdős–Gyárfás Conjecture 51

List of Figures **53**

Bibliography **55**

1. Introduction

In this chapter, we present the concepts of set theory and Busy Beaver, along with the motivation behind the connection between both notions.

1.1. Motivation and Set Theory

This part is based on explanations from [Lia11]. We present a brief description of the set theory concerning our problem. Set theory is a branch of mathematics that treats assemblages of objects. A set represents a collection of distinct objects sharing the same properties. These objects are called elements of the set. Historically, the initial works on set theory started in the late 19th century with the advancements of Georg Cantor and Richard Dedekind. Further works over the 20th century revealed several paradoxes in set theory, namely Russel's Paradox [Wikd] which results from an unrestricted comprehension principle that allows a set to comprise itself. At the beginning of the 20th century, Ernst Zermelo introduced the first axiomatic set theory. Later on, Abraham Fraenkel proposed some revisions on Zermelo's works by adding a stronger axiom, thereby establishing the axiomatic system that became known as the *Zermelo-Fraenkel* ($\mathcal{ZF}$) set theory. Most mathematicians also include the axiom of choice without reservation, since several generally accepted mathematical results rely on it in elaborating their proofs. Thus, $\mathcal{ZF}$ with the axiom of choice ($\mathcal{ZFC}$) became the standard axiomatic set theory and the foundation of most contemporary mathematics.

The axiomatic set theory system that defines $\mathcal{ZFC}$ handles sets, whose elements are themselves sets. Consequently, it is formally a one-sorted theory in first-order logic, as it allows working with only one type of object in the universe, that is sets, and permits quantifying only over elements of the domain.

Definition 1.1 *Let σ_{ZFC} be the signature (a signature contains the allowed relations and constants) of $\mathcal{ZFC}$ in first-order logic, then:*

$$\sigma_{ZFC} = (\in, =)$$

The set membership $\in$ represents a binary relation. For example, for two sets x and y, the well-formed formula $x \in y$ means that x is a member of y.

Actually, the non-logical equality symbol in the signature could be omitted, as $x = y$ is none other than an abbreviation of the following conjunction:

$$\forall z(z \in x \Leftrightarrow z \in y) \wedge \forall w(x \in w \Leftrightarrow y \in w)$$

One of the motivations for $\mathcal{ZFC}$ is the concept of cumulative hierarchy or the so called von Neuman universe named after John von Neuman. In this perspective, the domain of discourse is created step by step, where each step, called stage, is recursively created from the previous one. Particularly, at stage 0 there is no set. In the next stage, a set is appended to the domain, if the entirety of its elements were already added in the previous stage. Therefore, the empty set is added in stage 1, the set that contains the empty set is added in stage 2, and the set that contains both the empty set and the set that contains the empty set is added in stage 3, etc. For instance, the set of natural numbers in $\mathcal{ZFC}$ is defined recursively. The following example illustrates the procedure:

$0 = \varnothing$
$1 = 0 \cup \{0\} = \{\varnothing\}$
$2 = 1 \cup \{1\} = \{\varnothing, \{\varnothing\}\}$
$3 = 2 \cup \{2\} = \{\varnothing, \{\varnothing\}, \{\varnothing, \{\varnothing\}\}\}$
$4 = 3 \cup \{3\} = \{\varnothing, \{\varnothing\}, \{\varnothing, \{\varnothing\}\}, \{\varnothing, \{\varnothing\}, \{\varnothing, \{\varnothing\}\}\}\}$

The successor of each set n is recursively defined as:

$$S(n) = n \cup \{n\}$$

With definitions similar in concept, one could define relations, functions, and other infinite sets like the integers, rational and real numbers. Each theorem characterized along the process can be proven using the axioms of $\mathcal{ZFC}$ and the deductive systems of first-order logic. The resulting sequences or collections are all anchored by 0. If 0 exists, then so does 1 and 2 and all other ordinals, and each more complex set and rule built upon them. Hence, $\mathcal{ZFC}$ requires the existence of one set, namely the set that has no element, which we referred to as $\varnothing$. The latter is uniquely defined by the axiom of existence.

For a rigorous description of the 9 axioms and their corresponding proofs, visit [Met], or take a look at the presentation of the fundamentals of $\mathcal{ZFC}$. [Lia11]

However, Gödel's incompleteness theorems restrict $\mathcal{ZFC}$ just like any other axiomatic system capable of processing arithmetic and working with natural numbers. The first incompleteness theorem declares that if a theory is consistent, then there is at least one provable theorem about the natural numbers that is unprovable using the axioms

of that theory. The second Gödel's incompleteness theorem states that if a theory is consistent, it cannot prove its consistency. As mentioned above, this axiomatic set theory is the basis of modern built mathematics and other set theories. Hence, all mathematical theorems and results are built on the assumption that $\mathcal{ZFC}$ is indeed consistent. Adam Yedidia and Scott Aaronson take advantage of this fact to draw a conclusion on a specific Turing machine. [YA16]

Let M be the Turing machine that uses an effective procedure to list the axioms of $\mathcal{ZFC}$. Because $\mathcal{ZFC}$ is encoded in first-order logic, we can utilize its inference rules to derive every provable theorem from the axioms and all theorems that have already been concluded. Let $\langle S \rangle$ be the set of sentences that are provable from $\mathcal{ZFC}$ axioms using first-order logic deductive systems. If M, by any chance, finds during the verification process a formula ϕ so that $\phi, \neg\phi \in \langle S \rangle$, then the machine halts. As a result, M will only halt if and only if it finds an inconsistency in $\mathcal{ZFC}$. Thus M will run forever if and only if $\mathcal{ZFC}$ is consistent. Accordingly, the question M's behavior (whether it halts or loops endlessly) is equivalent to *Con($\mathcal{ZFC}$)* (consistency of $\mathcal{ZFC}$). Therefore, as $\mathcal{ZFC}$ is not able to prove its consistency, it also cannot determine the behavior M. We refer to this unprovability as **independence**. Hence, the statements "M will halt" and "M will run forever" are independent of $\mathcal{ZFC}$ set theory.

In this context, Adam Yedidia and Scott Aaronson aimed to build such a machine, not only because its behavior is independent of set theory, but also because they wanted to find an upper bound on the smallest value n for which $\mathbf{BB(n)}$ is independent of a dominant formal system. They do so by building the machine M and exhibiting its states. However, Adam and Scott follow another approach and opt instead for a statement equivalent to *Con($\mathcal{ZFC}$)*. Stefan O'rear, on the contrary, profits from the technical results by Adam and Scott to indeed build the machine M that looks for contradictions in $\mathcal{ZFC}$ and achieves a smaller upper bound. Both ideas are discussed in chapter 2.

1.2. Busy Beaver

1.2.1. Turing Machine

Turing machines are primitive and universal models of computation and have been the inspiration for the invention of modern computers. As specified by Alan Turing in his work "On computable numbers" [TUR36], he defines a Turing machine as a model that allows a tape to use unlimited memory. When the Turing machine starts, the input is written on tape. In addition to that, there is a read-write head that could

move left, right, or remain at its position (neutral) at each configuration transition. Formally, the conventional definition of a Turing machine is as follows. [Vol20]

Turing Machine

Definition 1.2 *A Turing machine M is a 7-tuple $M := (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$, whereby the following applies for the each component:*

- *Q is the set of states.*
- *Σ is the input alphabet.*
- *$\Gamma \supset \Sigma$ is the set of tape symbols.*
- *q_0 is the starting state.*
- *$\square$ is the blank symbol.*
- *$F \subseteq Q$ is the set of final states.*
- *δ is the transition function.*
  For deterministic Turing machines (DTM, TM) the following applies:
  $$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, N, R\}$$
  For non-deterministic Turing machines (NTM) the following applies:
  $$\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, N, R\})$$

Busy Beaver Turing machines are a slightly narrowed version of this usual formalization. Such machines are deterministic, possess a single tape, that is, at the same time, the input and working tape. The latter is initially filled with the blank symbol and is overwritten with a non-blank symbol from the working alphabet. Furthermore, Busy Beavers exclude the neutral movement of the pointer and allow only a transition to the left or right. Hence, throughout this paper, all the discussed Turing machines have the following formalization:

Busy Beaver

Definition 1.3 *A Busy Beaver Turing machine M is a 7-tuple $M := (Q, \Sigma, \Gamma, a, \delta, q_0, F)$, whereby the following applies for the each component:*

- *Q is the set of states, $|Q| = n + 1$, $n \in \mathbb{N}$ (in relation to $\mathcal{BB}(n)$).*
- *$\Sigma = \{a, b\}$ is the input alphabet.*
- *$\Gamma \supset \Sigma$, $\Gamma = \{a, b\}$ is the set of tape symbols.*
- *q_0 is the starting state.*
- *a is the blank symbol.*
- *$F \subseteq Q$, $F = \{HALT, ERROR\}$ is the set of final states (the ERROR state is only used for testing and the machine should not commit a transition to it on correct behavior).*
- *$\delta = Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function (deterministic).*

The delta transition function uniquely determines the behavior of the Turing machine. It takes two inputs: the current state and the symbol read from the tape. It outputs the next state, the symbol to be written on the tape, and the next direction of the head.

Semantically, a Busy Beaver Turing machine with n states is the machine that produces the maximum number of transitions among all other n-state Turing machines before halting, knowing that it will eventually halt. The Busy Beaver function, therefore, $\mathbf{BB(n)}$ or the shift function $\mathbf{S(n)}$, as defined by Radó [RAD61], computes the number of transitions it takes a n-state Busy Beaver to halt. Radó also described the function $\Sigma(n)$, which outputs the number of 1s (here bs) written on the tape before the machine halts. However, in this paper we stick with $\mathbf{BB(n)}$ as it is more intuitive.

1.2.2. Non-Computability of the Busy Beaver Function

Most of the statements in this chapter were pointedly elaborated by Scott Aaronson [Aar20]. Let $T(n)$ be the set of all n-state Turing machines. One could simply determine the cardinality of $T(n)$ to be $|T(n)| = (4 \cdot n + 4)^{2 \cdot n}$ as an application of the generalized formula $(symbols \times directions \times (states + 1))^{states \times symbols}$ to our specific Turing machines structure. The formula emerges from the possible combinations the

δ transition function could have.

Let M be a Turing machine that terminates. Let $s(M)$ be the number of steps or δ transitions that M takes before halting. If M stops at the first step, then $s(M) = 1$, and if M loops endlessly, we set $s(M) = \infty$.

At this stage, we can formally define our Busy Beaver function $\mathbf{BB(n)}$ (or $\mathbf{S(n)}$) as the following:

$$BB(n) = S(n) \coloneqq \max_{M \in T(n) \mid s(M) < \infty} s(M)$$

Informally, we range over all the finitely many n-state Turing machines. If we can safely affirm that a Turing machine runs forever, then we disregard it as this contradicts the definition of a Busy Beaver. Afterward, we maximize over the number of steps the halting Turing machines take before eventually transitioning into the **HALT** state. In his publication [RAD61], Radó describes this iterating over n-state halting Turing machines as the Busy Beaver game. The latter represents a competition, in which the winners are machines that achieve the maximum number of steps before halting, thereby determining the corresponding value of $\mathbf{BB(n)}$.

Prior to presenting a straightforward proof of the non-computability of **BB** in conjunction with the famous halting problem, let's start by defining the terms *computability* and *decidability*. The following definitions are present in [Vol20].

Definition 1.4 *A function is computabe iff it is Turing-computable.*
A function $f : \Sigma^* \to \Delta^*$ is Turing-computable, if there is a deterministic Turing machine M, so that for all $x \in \Sigma^*$ and $y \in \Delta^*$: $f(x) = y \Longrightarrow M$ on input x halts with $\square \ldots \square y \square \ldots \square$ on the output tape (in case of multi-tape machines).
$f(x)$ is undefined $\Longrightarrow M$ on input x runs forever.

Definition 1.5 *A language $A \subseteq \Sigma^*$ is decidable, if the function $c_A : \Sigma^* \longrightarrow \{0, 1\}$ with*

$$c_A(w) = \begin{cases} 1 & , w \in A \\ 0 & , otherwise \end{cases} .$$

is (Turing)-computable. c_A is called the characteristic function of A.

Definition 1.6 *The halting problem is the language:*

$$H = \{w\#x \mid M_w \text{ halts on input } x \}$$

It is widely known that the halting problem is undecidable. To prove this, one could directly perform a reduction over the special halting problem [Vol20]. In other words,

there is no Turing machine that on input $w\#x$ can decide whether M_w halts on input x or runs forever.

Based on these statements we can devise a simple proof by contradiction that $\mathbf{BB(n)}$ is not computable. Following that, we suppose that $\mathbf{BB(n)}$ is indeed computable. Hence, there is a Turing machine that, given a natural k as input, outputs $\mathbf{BB(k)}$ on its output tape. We prove that the characteristic function of the halting problem is computable:

Algorithm 1 Proof of decidability of the halting problem

Input: $w\#x$

1: create M_w with w being a proper godelization of a Turing machine. Let n be the number of its states
2: compute $\mathbf{BB(n)}$
3: run M_w on input x for $\mathbf{BB(n)}$ steps
4: If M_w has not halted yet, then we can evidently conclude that M_w will never halt, because $\mathbf{BB(n)}$ is the maximum number of steps a n-state **halting** machine can perform before halting. **Output 0**
5: if M_w halted before or at $\mathbf{BB(n)}$ steps then **output 1**

Thus, we have just proved that the halting problem is decidable, which we know is impossible. As a result, the Busy Beaver function is not computable.

Historically, the Busy Beaver game has attracted considerable interest in the quest of determining as many $\mathbf{BB(n)}$ values as possible. The highest lower bound on the provable Busy Beaver value in modern mathematics systems is 4. Nonetheless, lower bounds have been proposed for 5, 6 and 7-state Busy Beavers:

$$\mathbf{BB}(1) = 1$$
$$\mathbf{BB}(2) = 6$$
$$\mathbf{BB}(3) = 21$$
$$\mathbf{BB}(4) = 107$$
$$\mathbf{BB}(5) \geq 47,176,870$$
$$\mathbf{BB}(6) > 7.4 \times 10^{36,534}1$$
$$\mathbf{BB}(7) > 10^{10^{10^{10^{18,705,353}}}}$$

$$\vdots$$

$\mathbf{BB}(22) > g_{64}$ which is an enormously large number called the Graham's number [mat10] that results from 64 iterations of the rapidly growing Ackermann function [mat10].

Naturally, one could raise an interesting question on the computability of Busy Beaver values, namely, if we are able to prove $\mathbf{BB(n)}$ for $n \in \{1, \ldots, 4\}$, then why don't we just utilize the same computation process of ranging overall potential Busy Beavers in order to determine $\mathbf{BB(n)}$ for $n \geq 4$. The reason why this is not quite feasible is the existence of Turing machines, whose behavior is not decisively ascertainable. An example is the Turing machine M that we described in the previous chapter. Let n be the number of states of M. As the decision of whether M halts or runs forever is independent of $\mathcal{ZFC}$ axiomatic set theory, then the value of $\mathbf{BB(n)}$ is subsequently unprovable in $\mathcal{ZFC}$. Suppose that the latter is provable, then $\mathcal{ZFC}$ can easily prove the behavior of M. Such proof will consist of running M and keeping track of the entirety of its transitions. If it halts, then the memorized computation history of the machine is the halting proof. If it doesn't, then after exceeding the threshold of $\mathbf{BB(n)}$ steps and by definition of $\mathbf{BB}$, we have proved that M will loop endlessly.

Therefore, the machine M (also called Gödel Machine) suggests another perspective of approaching the Busy Beaver contest. The question of proving all Busy Beaver values becomes the question of how many of them are knowable using a fixed axiomatic arithmetic-encoding theory as a basis for our computations. In other words, from which n the value $\mathbf{BB(n)}$ eludes modern mathematics. Adam Yedidia, Scott Aaronson, and Stefan O'rear considered this dilemma as a powerful motivation for their works of finding the highest upper bound, for which the Busy Beaver value is provable in $\mathcal{ZFC}$ set theory. [YA16]

Theorem 1.1 *If T is a computable and arithmetic-encoding consistent axiomatic theory. Then, there exists a constant $n_T \in \mathbb{N}$, so that for all $n \geq n_T$, the value of $\mathbf{BB(n)}$ cannot be proved in T.*

Proof. Let M be our Turing machine discussed above that, on the all-blank symbol input, enumerates the axioms and provable theorems and conclusions of T and terminates iff it finds a contradiction. However, since we assumed that T is consistent, we know in advance that M will never halt. Nevertheless, T cannot prove that M will never halt. If it could, then we have $\mathbf{ZF} \models \mathbf{Con(ZF)}$, which is a direct violation of the second Gödel's incompleteness theorem. Now let n_T be the number of states in M, then, for all $n \geq n_T$, $\mathbf{BB(n)}$ is unprovable in T. If such proof existed, then T will be able to prove that M never halts, simply by simulation M for $\mathbf{BB(n)}$ steps. If M hasn't halted yet, then we know it will never halt because $\mathbf{BB(n)} \geq \mathbf{BB}(n_T)$. ■

In the next chapter, we will be discussing, among other topics, the achievements reached by Yedidia, Aaronson, and O'rear and the ideas implemented in the goal of finding the smallest possible n for which $\mathbf{BB(n)}$ is independent of $\mathcal{ZFC}$ set theory.

2. The Laconic Programming Language

Laconic is a strongly typed high-level programming language that was designed to be convenient and reassemble most common languages like Python and Java, and, at the same time, be volatile enough to be compiled down into 1−tape 2−symbol parsimonious Turing machines.

2.1. Concept and Compilation

Here, we present the purpose for developing Laconic and the main contributions of its compiler.

2.1.1. Motivation

An intuitive wondering immediately surges regarding the purpose of Laconic. Why would Yedidia develop a new programming language hunting for a goal that other popular languages could achieve in a highly efficient way?

Simply put, it has to do with the independence of the computational complexity of Turing machines compared to other models of computation. Using this probably least powerful model, one could unambiguously affirm that the complexity of the results to be interpreted is determined solely by the Turing machine. The more complex the chosen model of computation is, the more complexity of the algorithm is undertaken by the latter. Thus, misinterpretation might surface regarding the complexity of the implementation being influenced more by the choice of the model and less by the actual algorithm. Therefore, there can be no doubt that the choice of the programming language is artificially contributing to reducing the actual size of the Turing machine. In their publication [YA16] Yedidia and Aaronson describe Turing machines as follows:

> *Part of the charm of Turing machines is that they give us a "standard reference point" for measuring complexity, unencumbered by the details*

> *of more sophisticated programming languages... This is why we prefer Turing machines as a tool for measuring complexity; not because they are particularly special, but simply because they are so primitive that their specifics will interfere minimally with what we mean by an algorithm being "complicated."*

Besides, Laconic allows utilizing infinite memory, which corresponds to a tape of a Turing machine.

As one could reasonably conclude, the crucial thing to aim for in designing these Turing machines is **parsimony**. Yedidia and Aaronson's intention was to develop a Turing machine with the least amount of state possible that proves to be independent of modern mathematics, unconcerned by any time or space complexity. Similarly, parsimony expresses the principle of code-golfing through Turing machines.

To achieve parsimony, Yedidia and Aaronson [YA16] used two main concepts.

The first idea is referred to as **on-tape processing**. It is a way of encoding commands of a high-level language into a Turing machine parsimoniously. The main trick resides in replacing the multiplicative factor that results from representing each command through a set of states by firstly writing the commands on the tape using an efficient encoding and then creating a fixed set of states with the only purpose of understanding and interpreting the commands. Hence, sparing the colossal amount of states due to the multiplicative overhead and converting it into an additive constant. In Yedidia's implementation, the set interpreting the commands on the Turing machines (processor) comprises exactly 3860 states.

The second idea implements the efficient coding mentioned above that is used to write a binary string representing the high-level commands on the tape: **Introspection**. It means: Instead of using the state's write field to encode each character, which presents only 2 choices, one could take advantage of the number of states that a single state could transition to. Thus encoding information in each state's transition. Both of these concepts are explained pointedly in the next section.

2.1.2. From Laconic Code to Parsimonious Turing Machine

The following description of the compiler is introduced in Yedidia and Aaronson's work. [YA16]

A Laconic program is not directly translated into a 1−tape, 2−symbol Turing machine. Instead, Yedidia developed another high-level intermediary language which he named TMD (Turing Machine Descriptor). TMD describes multi-tape, 3−symbol Turing machines with a function stack. Each tape offers infinite memory in one

direction and allows the symbols: `_`, 1 and E. A regular expression of the content of each tape can be written as in the form $\_^*(1|E)^+\_^\infty$.

The idea behind employing TMD as a bridge between Laconic and a primitive Turing machine resides in the similarity between multi-tape Turing machines and variables. We directly allocate one tape for each variable. As a result, each instruction manipulating a particular variable in the Laconic program is compiled into a TMD command that modifies its corresponding tape accordingly.

TMD code is in reality not located in a single `.tmd` file, but is rather spread among multiple documents that constitute the compiled TMD directory. The latter consists of 3 types of files: The **functions** files which contains a list of all functions created and called by the TMD program, the **initvar** file, which contains the initialization of each variable, coded as a non-blank symbol in its appropriate tape. And all other files with the extension `.tfn` used to describe functions. For more details on the Laconic-to-TMD compiler and all components and operations stated in the following descriptions, take a look at the documents in Yedidia's GitHub repository. [Yed16]

At compilation time, the TMD program is transformed into a binary string which is written on the tape of final Turing machine upon its execution.

The latter comprises 3 main sub-machines:

- **The initializer** creates each register for each variable with a unique identifier. Each register's initial value is the value saved in the **initvar** file mentioned above. In the same way, it initializes the function stack and pushes to it the first entry stored in the **functions** file. The control flow is handled by counters.

- **The printer** writes down the compiled binary string of the TMD program on the tape of the Turing machine using an efficient way of encoding called **introspection**, which we will describe thoroughly later on.

- **The processor interprets** the written binary **on tape** after reading it again, thereby modifying the variable registers and the function stack during the machine execution.

The Turing's machine transition function proceeds from the initializer to the printer and then the processor. Each component comprises a bunch of states that achieve what each sub-machine is supposed to do. In other words, there are 6 types of transitions: From initializer states to initializer states, from initializer states to printer states, from printer states to printer states, from printer states to processor states, from processor states to processor states, or from processor states to the `HALT` state, where the machine stops. [YA16]

Introspection

The printer's labor is to write down the binary string of the TMD program on the tape of the Turing machine. A typical idea to achieve this is to encode each character of the binary with a dedicated state. Figure 2.1 shows an example of this inefficient model. However, in doing so, a considerable amount of bit space is wasted. As a matter of fact, each **a** transition in this naive model points to the next stage, whereas no **b** transition occurs as the machine will only read blank symbols. Eventually, only the write field is used to store information, that is one single bit. In reality, in an n-state Turing machine, each state may point to n states for each of the two transitions **a** and **b** plus one bit for which symbol to write, thereby possibly encoding $2 \cdot (\log_2(n) + 1)$ bits of information.

Introspection relies on this sparing idea and works as follows: If the binary string is k bit long, then we define the word size w as the largest value possible so that $w \cdot 2^w \leq k$. Then, the binary gets split in $n_w = \lceil \frac{k}{w} \rceil$ words of w bits each (pad with blank symbol if necessary). In the Turing machine, each word is represented by a data state. Then, in each data state, the **a** transition points to the data state that **represents** the next word, whereas the **b** transition points to the data state that **encodes** the next word. Figure 2.2 illustrates an example of an introspective encoding.

For the interpreter to process the high-level commands introspectively encoded on the tape, it needs to read them from the data states first. For this purpose, Yedidia built an extractor that comprises a certain number of states that is $10 \cdot w + 17$. The extractor queries each data state by removing one **b** at a time, thereby altering the content of the tape, then reverts it to its original form. A detailed explanation of the work scheme of the extractor can be found in [YA16].

In the example that depicts the introspective approach shown in figure 2.2, the program binary is 10 bits long, so we choose w that takes the largest value so that $w \cdot 2^w \leq 10$, which is 2. So we know the data states will encode words of length 2. Hence, the number of the words, and therefore data state is $n_w = \frac{k}{w} = \frac{10}{2} = 5$. As mentioned above, the **b** transition points to the data state that encodes the next word, whereas the **a** transition points to the data state representing the next word.

Limits of introspection

Exploiting this efficient encoding only brings benefits when the length of the program binary is quite large. The example shown in figure 2.2 with 5 data states results actually in an extractor of size $10 \cdot w + 17 = 10 \cdot 2 + 17 = 37$ states to only gain 5 states in

return. Therefore, even more states than in the naive model are needed.

Gains through introspection

Introspection lives up to its efficiency when utilized for large programs like encoding mathematical statements and conjectures. The programs mentioned in the following table are thoroughly explained in the next chapter.

Program	Binary Size	w	n_w	Extractor Size	#States (Naïve)	#States (Introspective)
ZFC	38,864	11	3,534	127	38,864	3,661 = 3,534 + 127
Lehmer	10,542	10	1,054	117	10,542	1,172
Scholz	11,650	10	1,165	117	11,650	1,282
Total Coloring	33,538	11	3,049	127	33,538	3,176
Erdős–Gyárfás	25,762	11	2,342	127	25,762	2,469

On tape processing

The final labor of the Turing machine is to process the written commands by the printer and modify the register variables and function stack. This interpretation takes place on the tape. On tape-processing is the immediate follow-up of introspection. As we were taught in the theoretical computer science lecture, every multi-tape Turing machine that runs in time $\mathbf{t}$ can be converted to a single-tape Turing machine that runs in time $\mathcal{O}(t^2)$ (polynomial). We do so simply by translating each transition in the higher-level machine into a set of states in the single-tape machine. This conduces the multiplicative overhead described in the motivation chapter. By encoding the processor with a fixed number of states that understand and execute the commands, an additive overhead of 3860 states replaces the multiplicative one. A step-by-step description of the processor's workflow is depicted in [YA16].

Final Costs

Program	Initializer States	Printer States	Processor States	Total States
ZFC	389	3,661	3,860	7,910
Lehmer	379	1,172	3,860	5,411
Scholz	387	1,282	3,860	5,529
Behzad	395	3,176	3,860	7,431
Erdős–Gyárfás	395	2,469	3,860	6,724

The resulting number of states validate the advantages brought by introspection and on tape processing in the task of achieving parsimony. On initial observation, one could deduce that optimizing the processor is the primary priority in improving the results. However, the number of processor states is constant. That is, it will not vary even if the program is significantly more demanding than those implemented in this paper. Whereas, the number of the printer states will grow linearly in the length of the program binary where handling larger problems.

Therefore, meliorating the printer in the first place, the processor in the second place, and with it, the Laconic programming language presented by Yedidia could be altogether promising in achieving fewer states.

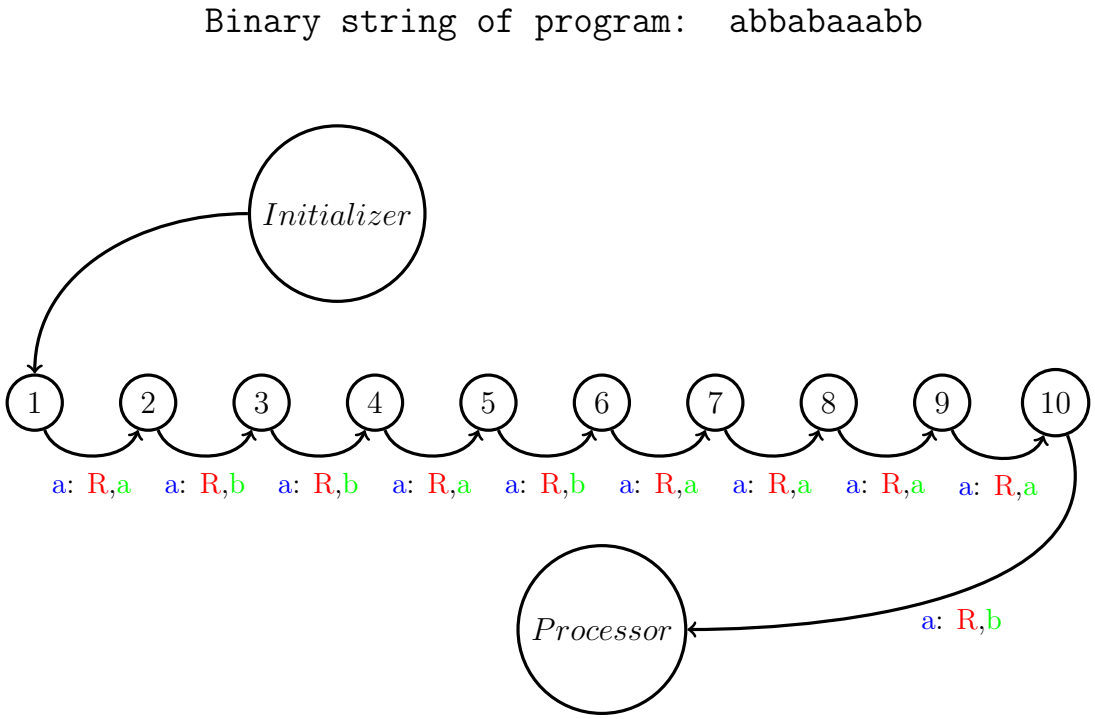

Figure 2.1.: naive states allocation of the printer for program binary

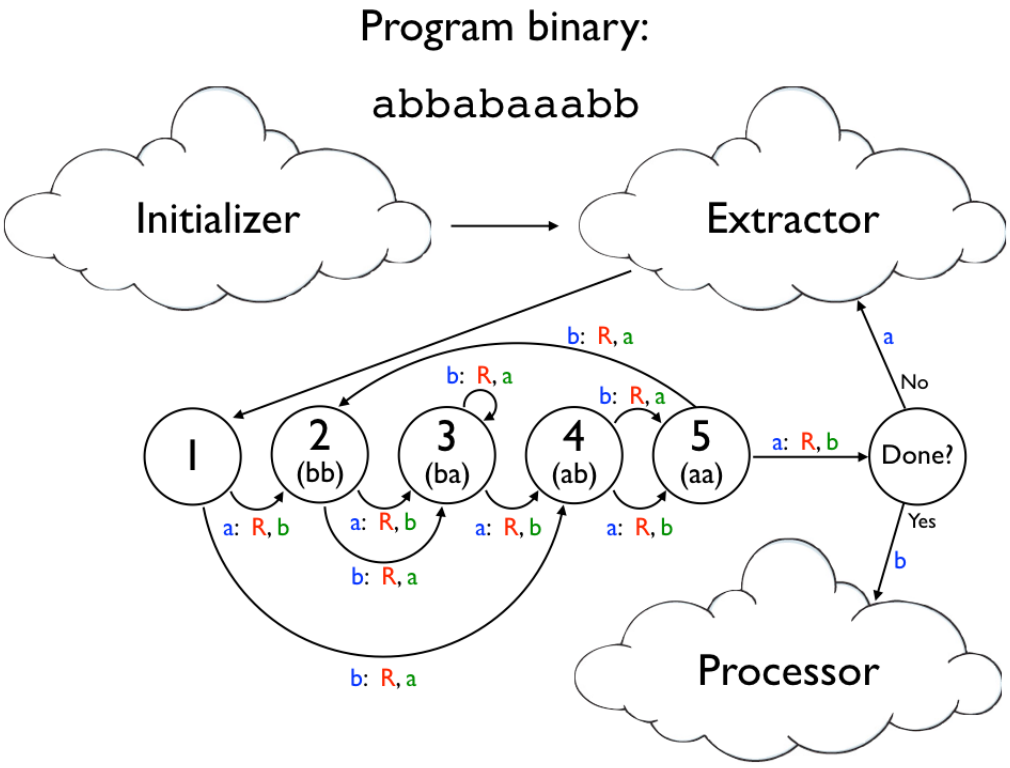

Figure 2.2.: an introspective implementation example of the printer with data states [YA16]

2.2. Application Examples

Here, we present the works of Adam Yedidia and Scott Aaronson using Laconic as well as individual application examples in graph theory and number theory.

2.2.1. The Friedman's Conjecture in Set Theory

In this section, we illustrate the approach and the implementation method of the Turing machine Z developed by Yedidia and Aaronson [YA16], whose behavior (whether it halts or loops endlessly) is independent of the $\mathcal{ZFC}$ set theory that we discussed in the above chapter. The fundamental goal of this Turing machine is to prove the lowest value n for which $\mathbf{BB(n)}$ is independent of $\mathcal{ZFC}$ set theory. In other words, what is the smallest number n for which the behavior of a n-state Turing machine can not be decisively determined using the axioms composing $\mathcal{ZFC}$, assuming $\mathcal{ZFC}$ is consistent?

An engaging approach to design the Turing machine Z is to encode the set theory in first-order logic. And then iterate over all provable statements using axioms of $\mathcal{ZFC}$ and the inference rules of first-order logic. The latter comprises many deductive systems (resolution, natural deduction, etc.) that allow inferring a formula from another formula as being a logical consequence.

However, Yedidia and Aaronson decide against this idea of enumerating all inferrable statements of $\mathcal{ZFC}$ because, despite its conceptual simplicity, it requires direct manipulation of the axioms and the deductive system rules of first-order logic. Hence, the resulting Turing machine would yield an enormous number of states.

Instead, they decide to take advantage of a statement devised by Friedman, whose truth is equivalent to the consistency of a set theory strong than $\mathcal{ZFC}$, which is $\mathcal{SRP}$. That imminently implies that $\mathcal{ZFC}$ is also consistent, as $\mathcal{SRP}$ is, in fact, an extended version of $\mathcal{ZFC}$ by some additional large cardinal axioms.

Before presenting the works, we explain some mathematical terms that will appear in Friedman's statement [Fri14]:

Technical Terms

Order invariant graph: An order invariant graph is a graph containing a countably infinite number of nodes. In particular, it has one node for each finite set of numbers.

In an order invariant graph, two nodes (a, b) have an edge between them if and only if each other pair of nodes (c, d), that is order equivalent with (a, b), has an edge between them.
Two pairs of nodes (a, b) and (c, d) are order equivalent if a and c are the same size and b and d are the same size and if for all $1 \leq i \leq |a|$ and $1 \leq j \leq |b|$, the i-th element of a is less than the j-th element of b if and only if the i-th element of c is less than the j-th element of d. [YA16]

$[\mathbb{Q}]^{\leq k}$: This operation refers to all the elements of the power set of $\mathbb{Q}$ whose cardinality is at most k. The power set $\mathcal{P}(S)$ of a set S is as follows:

$$\mathcal{P}(S) = \{U \mid U \subseteq S\}$$

The subsets of limited cardinality have a special notation

$$\mathcal{P}_k(S) = \{U \mid U \subseteq S \; and \; |U| \leq k\}$$

Therefore,

$$[\mathbb{Q}]^{\leq k} = \mathcal{P}_k(\mathbb{Q})$$

The vertices of a graph on $[\mathbb{Q}]^{\leq k}$ are elements of $[\mathbb{Q}]^{\leq k}$ and its edges are undirected, do not connect a pair of nodes twice, and cannot attach a node to itself. Each vertex of such a graph comprises a finite set of rational numbers.

Free set: A free set of nodes of a Graph $G = (V, E)$ comprises nodes so that there is no i and j, $1 \leq i, j \leq |V|$, for which $(v_i, v_j) \in E$ and $v_i, v_j \in V$. In other terms, no pair of vertices are connected by an edge.

Complexity of a number: A number of complexity c can be transformed in a fraction $\frac{a}{b}$, where a and b are integers with absolute values $\leq c$. A set has complexity c if and only if all its elements, which are rational numbers, are of complexity c.

The ush() function: ush() takes as input a set and returns a copy of that set with all non-negative numbers in that set incremented by 1.

$\underline{\leq_{rlex} \textit{ and reduction:}}$ In [Fri14], Friedman defines a way of ordering vertices of the graph, which is pretty similar to a lexicographic ordering. The last number of a vertex is the largest. Formally, for two nodes x and y, $x \leq_{rlex} y$ if and only if $x = y$ or $x_{|x|-i} < y_{|y|-i}$, where i is the smallest integer so that $x_{|x|-i} \neq y_{|y|-i}$. Using this reduction operation, we say a set of vertices X reduces another set of vertices Y, if for each vertex y in V, there exists a vertex $x \in X$ so that $x \leq_{rlex} y$ and x and y are connected by an edge.

With the use of the above concepts, Friedman [Fri14] asserts the following statement:

Statement 1 *for all* $k, n, r > 0$*, every order invariant graph on* $[\mathbb{Q}]^{\leq k}$ *has a free* $\{x_1, \ldots, x_r, ush(x_1), \ldots, ush(x_r)\}$ *of complexity* $(8knr)!$*. Each* $\{x_1, \ldots, x_{(8kni)!}\}$ *of the latter, for* $i > 0$ *and* $(8kni)! \leq r$*, reduces* $[x_1 \cup \cdots \cup x_i \cup \{0, \ldots, n\}]^{\leq k}$*. [Fri14]*

Hence, we ask Z to halt if and only if it finds a counterexample to statement 1 otherwise it keeps looping endlessly. Subsequently, this Turing machine's behavior is indubitably independent of $\mathcal{ZFC}$, because the falsehood of statement 1 is likewise independent of $\mathcal{ZFC}$, assuming the consistency of $\mathcal{SRP}$. [Fri14]

$$\mathcal{ZFC} \text{ inconsistent} \longrightarrow \mathcal{SRP} \text{ inconsistent} \longrightarrow Z \text{ halts.}$$
$$Z \text{ loops} \longrightarrow \mathcal{SRP} \text{ consistent} \longrightarrow \mathcal{ZFC} \text{ consistent.}$$
$$\mathcal{ZFC} \text{ cannot prove its consistency} \longrightarrow Z\text{'s behavior is independent of } \mathcal{ZFC}$$

Implementation method:

The following approach is thoroughly described in [YA16].

Encoding Friedman's statement is conceptually not complicated as each of its components can be implemented separately then combined in an iteration over all trios of numbers. Yet, this results in a considerably large laconic program. For full documentation of the code visit Yedidia's GitHub repository [Yed16]. I suggest looking primarily at the documented main loop of file *"friedman.lac"* to grab a better understanding of the method.

The main loop of the program starts by iterating over all possible tuples of numbers (k, n, r), and then generates a list N, in which all numbers of complexity $(8knr)!$ are stored. These numbers and others will represent the possible vertices of the graph. Numbers will be appended to the list if and only if they are written in the form $\frac{i}{j}$, where $-(8knr)! \leq i \leq (8knr)!$ and $1 \leq j \leq (8knr)!$. However, a problem surfaces as Laconic is not able to process float numbers. A solution is to multiply the candidate

fraction with a huge factor so that the resulting product is an integer. Yedidia and Aaronson settle for $((8knr)!)!$).

At the moment, N contains possible vertices that comprise only one number of complexity $(8knr)!$. Although, the statement compels a condition over graphs on $[\mathbb{Q}]^{\leq k}$. For this, we need to appends sets of numbers to N that represent subsets of $\mathbb{Q}$ with cardinality ranging from 2 to k. After the extension, N comprises now all the possible vertices of the potential order invariant graph for the particular tuple of integers (k, n, r). As a result, to ensure self-explanation, we rename it to V.

After preparing the possible set of vertices, the program iterates over all binary lists of length $|V|^2$. This list is a mapping of the possible edges of a graph $G = (V, E)$ as $E \subseteq V \times V$. In other words, every entry $E_{i|V|+j}$ in E represents whether the vertices V_i and V_j are connected (1) or not (0).

The aim is to find a binary list E, so that $G = (V, E)$ satisfies the three following conditions.

1. Check if G is undirected and does not contain selfloops.

2. Check that G is order invariant: Iterate over all pair of nodes (a,b). If a and b are connected, then verify that every other pair of nodes (c, d) that is order equivalent with (a, b) have an edge between them. In the same way, if a and b are not connected, then verify that every other pair of nodes (c, d) that is order equivalent with (a, b) does not have an edge between them.

3. G contains a free $\{x_1, \ldots, x_r, ush(x_1), \ldots, ush(x_r)\}$, where each $\{x_1, \ldots, x_{(8kni)!}\}$ from the latter, for $i > 0$ and $(8kni)! \leq r$, reduces $[x_1 \cup \cdots \cup x_i \cup \{0, \ldots, n\}]^{\leq k}$.

To verify the last condition, the program looks at each subset of V that has length r, then computes $\{ush(x_1), \ldots, ush(x_r)\}$, checks if it is in V and that $\{x_1, \ldots, x_r, ush(x_1), \ldots, ush(x_r)\}$ is a free set. Thereafter, we iterate over each subset $\{x_1, \ldots, x_{(8kni)!}\}$, for $i > 0$ and $(8kni)! \leq r$ and check if it reduces $[x_1 \cup \cdots \cup x_i \cup \{0, \ldots, n\}]^{\leq k}$ using the lexicographic ordering $\leq_{rlex}$ developed by Friedman. [Fri14]

For each list of nodes V, if there is no edges-list E so that $G = (V, E)$ is valid according to the previous criteria, then the program and its corresponding Turing machine halt, and therefore Friedman's statement is refuted.

The resulting Turing machine has 7910 states. Hence, it is inconceivable to prove the value of $\mathbf{BB(7,910)}$ without simultaneously proving or disproving the consistency of $\mathcal{SRP}$ and therefore $\mathcal{ZFC}$. As explained above, the behavior of this Turing machine Z is independent of $\mathcal{ZFC}$. Therefore, the least value n for which $\mathbf{BB(n)}$ is independent

of $\mathcal{ZFC}$ is at most 7910. This upper bound is not tight, and related works have been conducted to find a smaller upper bound. Section 2.3 presents a glimpse of these works.

2.2.2. The Lychrel Numbers

A natural number is said to be palindrome, if reversing its digits lead to the same number. In other words, there is a vertical axe, that splits the number into 2 symmetrical parts. [Wikc]

A natural number is a Lychrel number if **can not** turn into a palindrome after the iterative process of repeatedly reversing its digits and adding the results.

Example: 59 is not a Lychrel number in base 10 because it produces the palindrome 1111 after 3 iterations:

$$\begin{aligned} 59 + 95 &= 154 \\ 154 + 451 &= 605 \\ 605 + 506 &= 1111 \end{aligned}$$

We formally define the process through the Lychrel function $F_b : \mathbb{N} \longrightarrow \mathbb{N}$ for a base $b \geq 2$ as the following:

$$F_b(n) = n + \sum_{i=0}^{k-1} d_i b^{k-i-1}$$

where $k = \lfloor \log_b n \rfloor + 1$ is the length of the number in base b and,

$$d_i = \frac{n \mod b^{i+1} - n \mod b^i}{b^i}$$

The number n is therefore Lychrel if, for all $i \in \mathbb{N}$, $F_b^{i+1} \neq 2 \cdot F_b^i$. [Wikc]

Conjecture 1 *There exists no base-10 Lychrel number.*

This problem is, on several occasions, also referred to as the $196 - algorithm$, or the $196 - quest$. 196 is indeed the first number for which no palindrome has yet been determined. It is the lowest candidate Lychrel number and has therefore received the most attention. The first program ran for this purpose was conceived to last three years,

in which it computed 2.415.836 iterations and a number composed of 1.000.000 digits without reaching a palindrome. In 2015 Dolbeau's calculations attained a number with a billion digits without coming across a palindrome.

The Laconic program encoding the Lychrel problem compiles into a 5036-state Turing machine. The machine will halt if it ever finds a Lychrel number. Hence, it is unattainable to prove the value of $\mathbf{BB(5, 036)}$ without simultaneously solving the Lychrel problem. The Laconic program can be found in the appendices below.

2.2.3. The Scholz Conjecture

An interesting problem in number theory is to find the shortest addition chain given a number n- That is, how can we compute x^n with the fewest possible multiplications? The term *addition chain* rises from the perception that calculations involving multiplication and one variable x is highly similar to calculations involving addition and the natural 1.

Definition 2.1 *Let n be a natural number. An addition chain of n is defined as the following sequence of numbers:*

$$1 = a_0,\ a_1,\ \ldots,\ a_{r-1},\ a_r = n$$

with the property that:

$$a_i = a_j + a_k\ for\ j, k\ where j \leq k < i\ for\ all\ i \in \{1, \ldots, r\}\ [PDS81]$$

In other terms, an addition chain of n is given by a sequence of positive numbers starting with 1 and ending with n, where each element of the sequence is the sum of two previous ones. Note these latter might be the same.

Example: $(1, 2, 3, 6, 12, 24, 30, 31)$ represents an optimal addition chain of the natural 31, at it reaches the number after the minimum amount of additions. The length of an addition chain is equal to the number of sums needed to reach 31 which

is 1 less than the cardinality of the chain = 7:

$$\begin{aligned} 2 &= 1+1 \\ 3 &= 2+1 \\ 6 &= 3+3 \\ 12 &= 6+6 \\ 24 &= 12+12 \\ 30 &= 24+6 \\ 31 &= 30+1 \end{aligned}$$

Accordingly, this method allows computing the exponentiation of n using only 7 multiplications rather than the 30 multiplications of n by itself:

$$\begin{aligned} n^2 &= n \cdot n \\ n^3 &= n^2 \cdot n \\ n^6 &= n^3 \cdot n^3 \\ n^{12} &= n^6 \cdot n^6 \\ n^{24} &= n^{12} \cdot n^{12} \\ n^{30} &= n^{24} \cdot n^6 \\ n^{31} &= n^{30} \cdot n \end{aligned}$$

We define the generalized version of this problem: **The addition sequence problem**. Here, we do not seek an optimal addition chain that computes only one natural number. However, we look for a chain that optimally computes a sequence of positive integers $\{k_1, \ldots, k_n\}$. The sought optimum is the least number of additions to obtain all values of the sequence. [PDS81]

Treating the addition chain as a directed acyclic graph (dag) with a single leaf node having the value 1 and a single highlighted output node having the value n is preferable, as the order of the addition chains is irrelevant. Each non-leaf node has two successors, whose sum is the label (value) of the node. The length of the graph is the number of its non-leaf nodes. Accordingly, we can also present an addition chain for a sequence of positive integers the same way but with several distinguished output nodes labeled with each value of the sequel. We refer to a graph meeting these conditions as an

addition dag. [PDS81]

The left graph of figure 2.3 depicts a dag of 31, whereas the graph on the right depicts a dag for the sequence $\{1, 2, 4, 8, 15\}$.

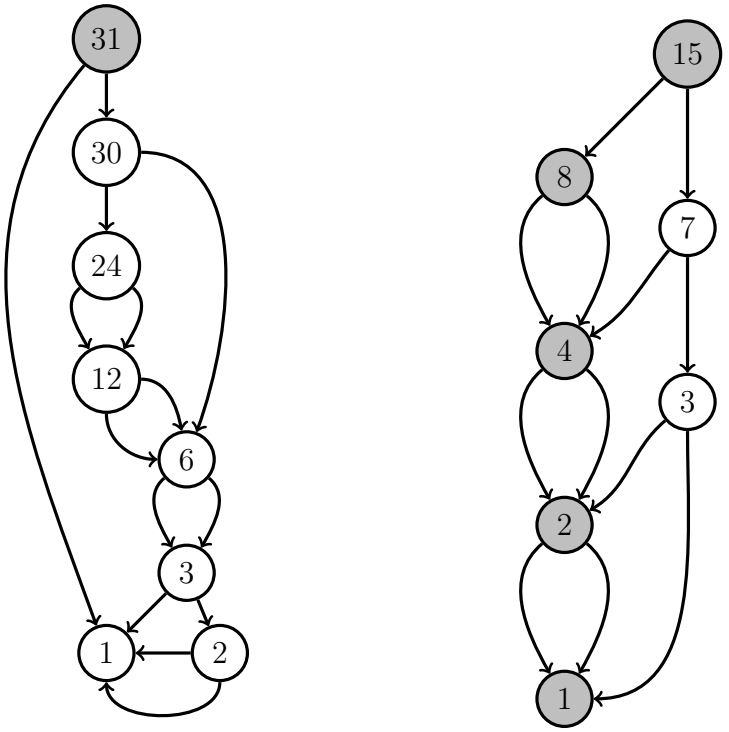

Figure 2.3.: non leaf nodes represent partial addition results

Downey, Leong, and Sethi presented a proof [PDS81] that the addition sequence problem is *NP-Complete* through a beautiful reduction over the famous Vertex Cover problem using the directed acyclic graph as a representation of a possible solution. Strangely enough, it has not still been proven that the particular problem of single-value addition chains is at least harder than the hardest problems in **NP**. Furthermore, there is no efficient algorithm or expression that computes the least number of additions needed for an addition chain of a natural n. Even though its **NP**-hardness is not yet confirmed, we still present a verifier that proves that the addition chain problem is in **NP**.

Prior to that, let's define the problem:

Definition 2.2 *$AC = \{\langle n, L\rangle \mid n, L \in \mathbb{N}$ and there is an addition dag whose output node is labeled with n and has length $\leq L\}$.*

A generous upper bound of the complexity of this verifier is $O(|G|^3)$. Therefore it runs in polynomial complexity and $AC \in$ NP.

Algorithm 2 Verifier for AC

Input: $\langle n, L, G = \langle V, E \rangle \rangle, n, L \in \mathbb{N}$ and G is a directed graph formed from vertices V and edges E. All nodes are labeled with positive integers.

1: check if $|V| - 1 \leq L$ ▷ runs in $O(|V|)$
2: check if root node of G has label n ▷ runs in $O(1)$
3: check if G has only one leaf node with label 1 ▷ runs in $O(1)$
4: **for** every node v $\neq$ leaf node in G **do** ▷ overall loop runs in $O(|V| \cdot (|V| \cdot |E| + |E|) \subseteq O(|V|^2 \cdot |E|)$
5: check if v has exactly 2 successors ▷ runs in $O(E)$
6: check if label of v is sum of its successors ▷ runs in $O(|V| \cdot |E|)$
7: **end for**
8: if always true then **accept** else **reject**

Enough description of the hardness of addition chains problems, now we present the actual Scholz conjecture:

Conjecture 2

$$l(2^n - 1) \leq n - 1 + l(n) \quad \forall n \in \mathbb{N}$$

where $l(n)$ presents the length of the shortest addition chain producing n.

Interestingly, results on this conjecture have already been achieved: Clift had proved that the inequality is actually an equality for all positive integers $\leq$ 64 [Cli10]. Moreover, he showed that the inequality is true for all naturals $\leq$ 5784689.

As presented above, finding the shortest addition chain is a tricky problem for which no feasible approach has been found. Besides, we cannot rely on approximations algorithms, because they do not provide an optimal solution, which is crucial in our case. That is, we might end up rejecting the conjecture with a poor solution even though an optimal solution would still verify it. As a result, we write a brute force algorithm that creates all possible addition chains until the first addition chain that ends with n is found. It is guaranteed that this first chain is indeed optimal because we append the chains with one value at every step. Hence, if the first solution we found has length k, we can safely affirm that any other solution will have at least length k. This method is far from efficient, though, as explained above, we do not focus here on efficiency but rather on parsimony. The Turing machine compiled from its corresponding Laconic Program has 5529 states. Hence, it is impossible to prove the value of $\mathbf{BB(5, 529)}$ without simultaneously proving or disproving the Scholz conjecture. The Laconic program can be found in the appendices below.

2.2.4. The Lehmer's Totient Problem

We present a 5411-state Turing machine that encodes the Lehmer's totient problem. Determining if this machine would ever halt is equivalent to determining whether the Lehmer's totient problem holds.

The Lehmer's totient problem asks whether there is any composite number n, in other words, any number that is not prime and not equal to 1, such that the Euler's totient function or Euler's function $\varphi(n)$ divides $n-1$. The latter is an unsolved problem as no such number has been found yet.

The statement holds trivially for prime numbers: It is known that $\varphi(n) = n-1$ if and only if n is prime [Hol19]. It follows that for every prime number p we have $\varphi(p) = p-1$ and, therefore, $\varphi(p)$ divides $p-1$.

The Euler's Phi function is defined in [Hol19] as follows:

Definition 2.3 *$\forall n \in \mathbb{N}, n \geq 2$, let $n = \prod_{i=1}^{k} p_i^{e_i}$ be the prime factorisation of n. Then the following applies:*

$$\varphi(n) = \prod_{i=1}^{k} p_i^{e_i - 1}(p_i - 1)$$

We encode this problem into a Laconic program that iterates over all composite numbers, calculates each Euler's function value $\varphi(n)$, and checks if it divides $n-1$. The program will stop if it ever finds such a number. Accordingly, its corresponding Turing machine will halt.

Hence, it is unachievable to prove the value of $\mathbf{BB(5, 411)}$ without simultaneously proving or disproving Lehmer's totient Problem. The Laconic program can be found in the appendices below.

2.2.5. The Total Coloring Conjecture

A total coloring represents the combination of vertex-coloring and edge-coloring. In graph theory, vertex coloring represents a way of coloring a graph's vertices, such that no adjacent vertices are assigned the same color. Similarly, edge coloring represents a way of coloring a graph's edges, such that no two incident edges are designated the same color. [Wike]

A total coloring is proper if no adjacent edges and no edge and its end vertices are assigned the same color. In the following, every coloring is considered proper.

While the chromatic number χ defines the minimum number of colors needed to obtain a vertex coloring of a graph, the total chromatic number χ'' of a graph G equals the fewest colors needed in any total coloring of G. [Wikb]

The following inequality provides a significant lower bound of the total chromatic number of a graph G:

$$\chi''(G) \geq \Delta(G) + 1$$

where $\Delta(G)$ is the maximum degree of G (a node has degree n iff it has n adjacent vertices). However, the task of finding a maximum degree-related upper bound is a strenuous problem that shook off mathematicians for many years.

Nonetheless, a collection of works on this matter have been achieved: Hind, Molloy, and Reed were able to develop a polynomial-time algorithm that finds a total coloring of a graph G with maximum degree Δ using at most $8 \log^8 \Delta$ colors. [HHR98]

However, an aim for a smaller upper bound was consistently present, as many graphs such as the complete bipartite graphs of the form $K_{n,n}$ needed exactly $\Delta(G)+2$ colors. No graph has ever been found that needed more colors. Thus the assumption that the total chromatic number of any graph G is either $\Delta(G) + 1$ or $\Delta(G) + 2$.

This conjecture was introduced by Behzad during his Ph.D. studies in 1965 and is, until today, never to be proven. [Beh65]

Conjecture 3

$$\chi''(\mathbf{G}) \leq \mathbf{\Delta(G)} + \mathbf{2}$$

Now, consider the following definitions.

Definition 2.4 *The total graph $T = T(G)$ of a graph G is a graph whose vertices correspond to the vertices and edges of G, and two vertices are adjacent in T if and only if their corresponding elements are either adjacent or incident in G. [Wike]*

Definition 2.5 *A total coloring of G is a vertex coloring of T where $T = T(G)$ is its corresponding total graph.*

By utilizing these definitions we can facilitate our undertaking and reformulate Behzad's conjecture to:

$$\chi(\mathbf{T(G)}) \leq \mathbf{\Delta(G)} + \mathbf{2}$$

We define the following optimisation problem:

Definition 2.6 ***MinVCP***

Problem: Minimum Vertex Coloring Problem (MinVCP)

Instances: $I = \{\langle (d_{i,j})_{1\leq i,j\leq n}\rangle \mid n, d_{i,j} \in \{0,1\}, 1 \leq i,j \leq n\}$. *We write*

$$D \coloneqq (d_{i,j})_{1\leq i,j\leq n},\ (d_{i,j}) = 0 \text{ for } i = j \text{ and } (d_{i,j}) = (d_{j,i}).$$

Solutions: $s(D)$ *is the set of all proper color labelings of* D *such that no two adjacent vertices in* D *are assigned the same label. Let* ξ *be a proper labeling of* D*, then* $\xi(i) \neq \xi(j)$ *for all* i, j *where* $i \neq j$ *,*$(d_{i,j}) = 1$ *and* $1 \leq i,j \leq n$*.* ξ *can be defined as follows :* $\xi\colon \{\mathbf{1},\ldots,\mathbf{n}\} \longrightarrow \mathbf{C}$*, where* C *is the set of colors used for the labeling of* D*.*

Measure: $m(D,\xi) = |C|$

Target: Min

We define its associated decision problem:

Definition 2.7 $MinVCP_D = \{\langle (d_{i,j})_{1\leq i,j\leq n}, K\rangle | d_{i,j} \in \{0,1\}\ \ i,j,K \in \mathbb{N}$*, and there is a total computable function* $\xi\colon \{\mathbf{1},\ldots,\mathbf{n}\} \longrightarrow \mathbf{C}$ *such that* $\xi(i) \neq \xi(j)$ *for all,* i,j $i \neq j$ *,*$(d_{i,j}) = 1,\ 1 \leq i,j \leq n$ *and* $|C| \leq K\}$.
$MinVCP_D$ *is none other than* $VCP =$ *Vertex Coloring Problem.*

An algorithm that solves an optimization problem is called an optimization algorithm. Similarly, the minimization and maximization problems are more precisely referred to as the minimization or maximization algorithms. Such algorithm is polynomial and computes an arbitrary best solution whose measure is the lowest possible if the target is Min. Knowing that MinVCP $\in$ NPO the existence of such an algorithm will imply $MinVCP \in$ PO.

Lemma 1 $P \in \mathrm{PO} \Rightarrow P_D \in \mathrm{P}$. *Proof in [Mei20]*

Given $MinVCP_D = VCP$, we conclude that VCP is in **P** and **P** = **NP** because VCP is *NP-complete*, thereby solving the most daunting problem in mathematics. As it is not the case, such algorithm is not yet to be determined.

Another seemingly appealing way to approach the problem is to use the so-called approximation algorithms, which are polynomial algorithms that compute an arbitrary solution that does not have to be optimal. In other terms, the number of needed colors for proper labeling is not minimal.

As we are trying to prove that the minimum number of colors needed for a vertex coloring of the total graph of G is tightly upper bounded, we certainly cannot rely on approximation algorithms. These could provide valid solutions that might exceed our upper bound, whereas more rigorous solutions exist.

Therefore, the most trustworthy way to approach the matter is to use the decision problem because it comes with a threshold. So, instead of creating an algorithm that searches for the minimum number of colors for a vertex coloring, we develop an algorithm, that given a graph G and an integer k, checks if it is feasible to achieve a vertex coloring using at most k colors. If it is not, then a vertex coloring of G needs necessarily at least $k + 1$ colors.

It is more accessible to validate an input than to generate it. Even though the algorithm's complexity might not be polynomial, it is still more solid than developing an efficient optimization algorithm. As explained in section 2.1.1, our aim is not to build efficient algorithms but instead to generate parsimonious Turing machines that simulate the algorithm with as few states as possible.

The task of generating all graphs of n vertices through brute force is by itself pretty slow, as it grows exponentially: Given n vertices, the maximum number of edges an undirected Graph G can have is $\frac{n(n-1)}{2}$. We define X_{C_i} as the total number of graphs with n vertices and i edges $\forall i \in \{0, \ldots, X\}$.

Consequently: #different graphs of n vertices $= \sum_{i=0}^{X} X_{C_i}$. Simplifying the expression leads to $2^X = 2^{\frac{n(n-1)}{2}}$ different graphs. This straightforwardly means that the number of operations needed also grows exponentially. Whenever we increase n by one, we add $n-1$ edges into consideration. In other words, we add 2^{n-1} operations to the run-time. So, for example, if the algorithm is efficient enough to generate all graphs of 6 vertices in seconds, then generating all graphs of 7 vertices will shift the run-time from seconds to minutes, for $N = 8$ it will result in hours and for $N = 9$ in months, etc. Even though time complexity is not as important as parsimony in this work, we still propose an average complexity of the algorithm, assuming it might halt after it finds a counterexample in its first iteration. [Kha20]

Finding a counter example in the first iteration will lead in the worst case to a time

complexity in $\mathcal{O}\left(2^{n\cdot(n-1)}\cdot n^2 + 2^{\frac{n(n-1)}{2}}\cdot n)\right)$.

The graph coloring method processes disconnected graphs as well. It treats each connected component as an independent graph and performs the labeling accordingly. For instance, if a disconnected component has only 1 vertex, then this vertex will be assigned an arbitrary color from any vertex in a connected component.

Finally, the implementation of the following algorithm in Laconic compiles into a 7431-state Turing machine. Hence, it is inconceivable to prove the value of $\mathbf{BB}(\mathbf{7}, \mathbf{431})$ without simultaneously proving or disproving the total coloring conjecture. The Laconic program can be found in the appendices below.

Algorithm 3 Total Coloring Conjecture

```
1:  totalColoringConjecture ← true
2:  n ← 2                                              ▷ Number of vertices
3:  while totalColoringConjecture do
4:      graphs ← generateGraphsOfNVertices(n)
5:      for G in graphs do
6:          if totalColoringConjecture then
7:              totalGraph ← generateTotalGraph(G)
8:              labeling ← {0}                  ▷ No color is assigned to any vertex
9:              threshold ← 0                       ▷ The number of colors to check
10:             found ← false                ▷ True if proper vertex coloring is found
11:             while !found do
12:                 threshold ← threshold + 1
13:                 graphColoring(G, labeling, threshold, found) ▷ Threshold enough?
14:             end while
15:             minimumValidColors ← threshold
16:             maximumDegree ← maxDegree(G)
17:             upperBound ← maximumDegree + 2
18:             if minimumValidColors > upperBound then      ▷ Verify the conjecture
19:                 totalColoringConjecture ← false
20:             end if
21:         end if
22:     end for
23:     n ← n + 1
24: end while
```

2.2.6. The Erdős–Gyárfás Conjecture

In 1995 the creative mathematician Paul Erdős and his collaborator András Gyárfás devised a property on simple cubic graphs [Wika]:

Conjecture 4 *Every cubic graph contains a simple cycle whose length is a power of 2. (cubic graphs are* 3*-regular- That is, each vertex has exactly degree* 3*)*

Before jumping into the implementation methods, we present some preliminaries on terms concerning graphs. A graph $G = (V, E)$ is composed of a set of vertices V and a set of edges E where $E \subseteq V \times V$. A graph is simple if and only if it comprises no self-loops and no multiple edges between the same vertices. We define a simple path in a graph G as a sequence of nodes $P = \{v_1, \ldots, v_n\}$ where $v_i \neq v_j$ if $\{i, j\} \neq \{1, n\}$ for all $i, j \in \{1, \ldots, n\}$ and $(v_i, v_{i+1}) \in E$ for all $i \in \{1, \ldots, n-1\}$, $n \in \mathbb{N}$ and $n \leq |V|$. The length of P is the number of edges connecting every two consecutive vertices. In the following, we refer to v_1 as head and v_n as a tail. If the head and the tail of the simple path are identical, we are talking about a simple cycle. Moreover, we refer to simple cycles of length k as k-cycles.

The implementation procedure is divided into two parts: **Generating all cubic graphs of** n **given vertices** and **iterating through the simple cycles of each graph**.

Generating Cubic Graphs Through Randomization

Even though the focus of this paper is, as mentioned above, mainly on parsimonious Turing machines, we still prefer to approach programming challenges efficiently if accorded the opportunity. Opting for brute force like in the total coloring conjecture to generate cubic graphs is certainly not a way to adopt, because, on closer examination, the number of cubic graphs of n nodes is meager compared to the number of all graphs of n nodes. As a result, we end up with an awful amount of computation and states. Several algorithms generating random k-regular graphs uniformly or close to uniformly at random were developed. Here, we present a method devised by Steger and Wormald [SW98] based on the first algorithm for generating d-regular graphs conceived by Bollobas, where he employs the concept of pairing [Bol80]. Let n be the number of vertices of the d-regular graph $G = (V, E)$ we are about to create. The description of the algorithm from [SW98] is as follows:

1. Begin with $n \cdot d$ nodes $\{1, 2, ..., n \cdot d\}$ ($n \cdot d$ has to be even, otherwise there will be no such graph as $\sum_{v \in V} deg(v) = n \cdot d = 2 \cdot |E|$) in n groups. Let $U = \{1, 2, \ldots, nd\}$, U denotes the set of unpaired points.

2. Choose two random vertices i and j from U. If i and j are suitable then pair i with j and delete i and j from U (two nodes are suitable if they lie in different

groups and no currently existing pair contains points in the same two groups). Repeat the previous until no two suitable vertices are found.

3. Create G through adding an edge from node q to node p if and only if there is a pair comprising nodes in the $q.th$ and $p.th$ groups. Afterwards check if G is d-regular. If so, then output it else repeat from the first step.

The authors [SW98] proved that the run-time of the second step is polynomial and bounded by $O(n \cdot d^2 + d^4)$. Besides, they also show that the distribution of the generated d-regular graphs is close to uniform. In our case, $d = 3$ is fixed as we are generating cubic graphs.

The introduced algorithm produces a single random graph. However, in order for us to attempt to prove or disprove the Erdős–Gyárfás Conjecture, we need to iterate over all cubic graphs. A related issue is the number of k-regular graphs. A formula or an expression that gives the exact value is not yet determined. Nonetheless, Bollobas [Bol80] affirms that the numbers of d-regular graphs of n nodes are asymptotically

$$\sqrt{2}e^{\frac{1}{4}}(\lambda^{\lambda}(1-\lambda^{\lambda}))^{\binom{n}{2}}\binom{n-1}{d}^{n}$$

where $\lambda = \frac{d}{n-1}$. In our case, we just replace d by 3. Even though this upper bound considerably overgrows the actual number of d-regular graphs for a growing number of vertices n, it is certainly a more convenient threshold then the number of all graphs of n vetrices. Hence, we execute the random graph generator as many times as the asymptotic function affirms for a specific n and store merely different generated 3-regular graphs.

Unfortunately, implementing this in Laconic is inconceivable. This is because Laconic does not provide a way to generate random or pseudo-random numbers. One could create a random number generator like the linear congruential generator. However, this will always lead to a predictable cycle of random numbers as the seed itself cannot be assigned randomly in Laconic. In the end, unfortunately, we had to use the brute force method.

Enumerating Cycles in a Graph

The second part of the implementation is to search for a cycle whose length is a power of two in the generated cubic graph. Looking for a valid cycle through iteration overall potential ones is neither efficient nor parsimonious. Since our central focus is golfing, we adopt an algorithm introduced by Liu and Wang [LW06] that is relatively efficient but considerably parsimonious. The latter is simple and gets implemented with few

variables and statements using Laconic. The algorithm relies on the idea that paths of length k can be effortlessly generated from paths of length $k-1$. We directly attach an edge from the tail to another vertice that does not appear in the path. In like manner, a k-cycle comprises a path of length $k-1$ and an edge connecting the tail to the head.

We slightly alter the presented algorithm by adding a break condition that stops the iteration over the found paths whenever a cycle whose length is a power of two is constructed, and by conditioning on vertices rather than edges. Figure 2.4 illustrates a detailed flowchart of the algorithm.

The resulting Turing machine of the Laconic program comprises 6724 states. Hence, it is impossible to prove the value of $\mathbf{BB}(\mathbf{6}, \mathbf{724})$ without simultaneously proving or disproving the Erdős–Gyárfás conjecture. A link to the Laconic program is in the appendices below.

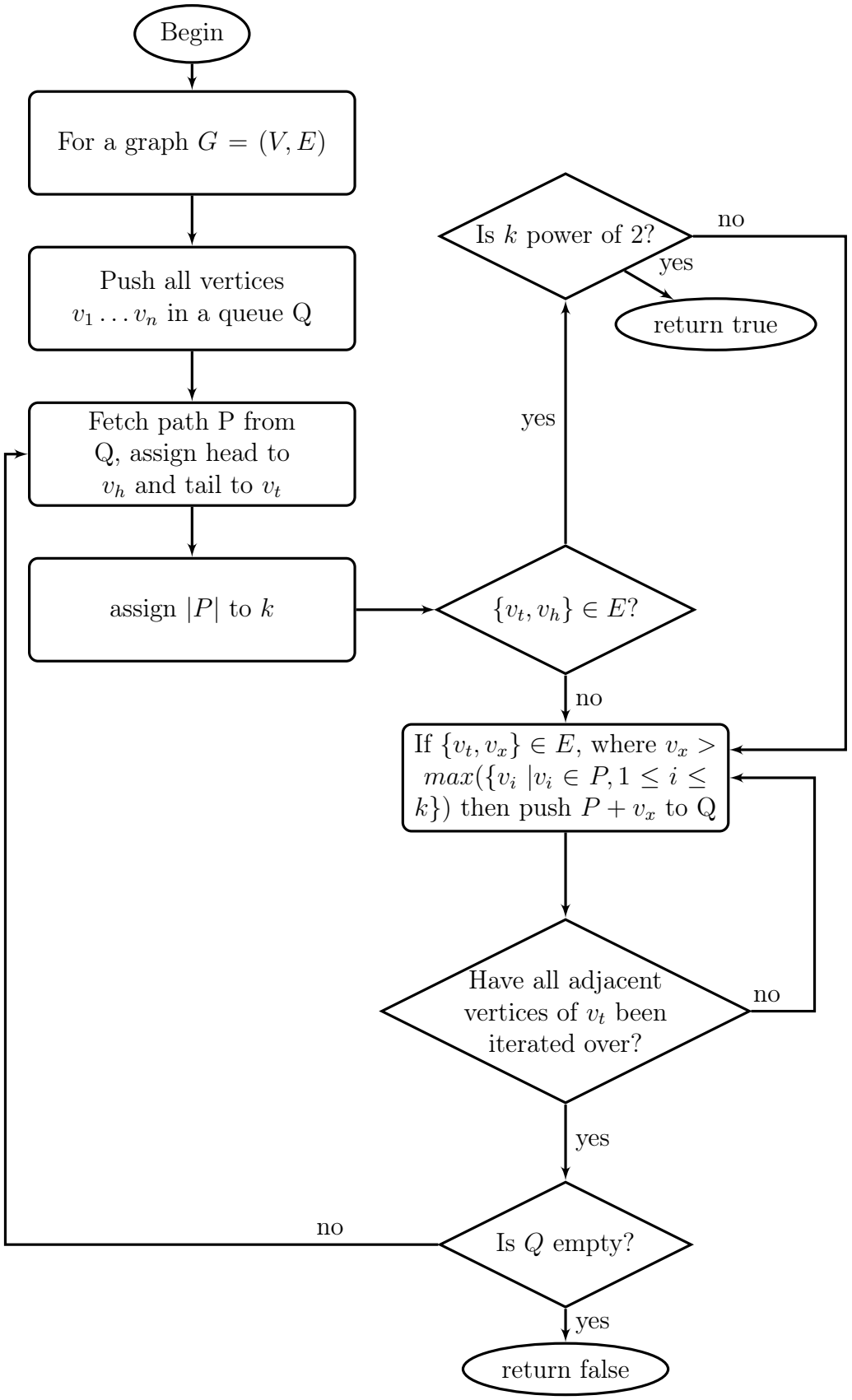

Figure 2.4.: Flowchart of enumerating k-cycles algorithm

2.3. Related Work: Not Quite Laconic

In this section, we briefly present the enhancements obtained by Stefan O'rear in searching for a stronger lower bound to the lowest value n, for which **BB(n)** is independent of $\mathcal{ZFC}$ set theory.

Motivated by the achievements of Scott Aaronson and Adam Yedidia in determining an upper bound on the first **BB** value eluding $\mathcal{ZFC}$, Stefan O'rear took advantage of Laconic to develop NQL (Not Quite Laconic), with which he directly searches for contradictions in a system equivalent to $\mathcal{ZFC}$. Thereby, he eliminates the reliance on Friedman's statement as well as on $\mathcal{SRP}$. NQL offers less functional diversity than Laconic as it has no native support for lists, negative numbers and handles procedures rather than functions. However, it compensates through optimized performance and better state count. More details about NQL are to be found in Stefan's Github repository. [O'r17a]

State count is affected strongly by logic but much less so by the lengths of axioms. Because of that, Stefan moved as much complexity as possible into axioms and out from side conditions. Furthermore, theorems inferred from stronger axioms were not explicitly iterated over as they will be proved sometime during the execution. The implementation of Stefan discards the axiom of choice as it does not affect the soundness of *Con($\mathcal{ZFC}$)*. The program uses the axiomatization for all universally valid sentences of predicate logic with identity conceived by Tarski. [TAR64]

The final set of axioms representing first-order logic and $\mathcal{ZF}$ is the following:

$$\textbf{Predicate Calculus} = \begin{cases} \text{Modus Ponens} : \varphi \ \& \ (\varphi \to \psi) \Rightarrow \psi \\ \text{Generalization} : \varphi \Rightarrow \forall x \varphi \\ \text{B1} : (\varphi \to \psi) \to ((\psi \to \theta) \to (\varphi \to \theta) \\ \text{B2} : (\neg\varphi \to \varphi) \to \varphi \\ \text{B3} : \varphi \to (\neg\varphi \to \psi) \\ \text{B4} : \forall x(\varphi \to \psi) \to (\forall x \varphi \to \forall x \psi) \\ \text{B6} : \varphi \to \forall x \varphi \\ \text{B7} : \exists x \ x = y \\ \text{B8} : x = y \to (\varphi \to \psi) \end{cases}$$

where x, y and z range over variables, and φ, ψ and θ range over well-formed formulas. In $B6$, x does not appear syntactically in φ. φ and ψ are atomic formulas in

$B8$, and ψ is obtained from φ by replacing a single instance of x with y. Therefore, in order to remove these enclosed conditions, Stefan differentiates 3 cases for each axiom.

$$\textbf{Zermelo-Fraenkel} = \begin{cases} \text{Extensionality} : \forall v_2(v_2 \in v_0 \leftrightarrow v_2 \in v_1) \rightarrow v_0 = v_1 \\ \text{Replacement} : \forall v_3 \exists v_1 \forall v_2(\forall v_1 \varphi \rightarrow v_2 = v_1) \rightarrow \exists v_1 \forall v_2 \\ (v_2 \in v_1 \leftrightarrow \exists v_3(v_3 \in v_0 \wedge \forall v_1 \varphi)) \\ \text{Power Sets} : \exists v_1 \forall v_2(\forall v_3(v_3 \in v_2 \rightarrow v_3 \in v_0) \rightarrow v_2 \in v_1) \\ \text{Union} : \exists v_1 \forall v_2(\exists v_3(v_2 \in v_1 \wedge v_3 \in v_0) \rightarrow v_2 \in v_1 \\ \text{Infinity} : \exists v_1(v_0 \in v_1 \wedge \forall v_0(v_0 \in v_1 \rightarrow \exists v_2(v_2 \in v_1 \wedge \\ \forall v_1(v_1 \in v_2 \leftrightarrow v_1 = v_0)))) \end{cases}$$

where v_1, v_2 and v_3 range over variables and φ over well formed formulas.

In his implementation [O'r17a], Stefan encodes formulas and proofs as integers. For this, he exploits the bijective Cantor pairing function, which takes pairs of non-negative integers and returns a unique identifier for each pair. He denotes with $(X \cdot Y)$ the Cantor value of the pair of integers (X, Y).

$$(X \cdot Y) = \frac{X + (X + Y) \cdot (X + Y + 1)}{2}$$

Applying this function allows a recursive numbering of formulas. For example:

- $\langle v_i = v_j \rangle \rightarrow (i \cdot j) \cdot 0$
- $\langle \varphi \rightarrow \psi \rangle \rightarrow (\langle \varphi \rangle \cdot |\psi|) \cdot 2$

where $\langle\ \rangle$ denotes the encoded value.´

The following explanation is pointedly cited in Stefan's $\mathcal{ZF}$-enumerator NQL file [O'r17b]. The formula with value 0 is $(v_0 = v_0)$ which is an obvious theorem of $\mathcal{ZF}$. The formula $(v_0 \in v_0)$ has value 1 and is not a theorem, as it states that all sets contain themselves. The latter serves as a contradiction, thereby the Turing machine will halt whenever a proof of 1 is found.

A proof will then consist of a list that contains chunks of 4 integers. Each group contains an axiom code followed by 3 parameters like $(ACa\ P1a\ P2A\ P3a; ACb\ P1b\ P2b\ P3b; ...)$ that represent variables or formulas. The assignment of the codes to each axiom is listed in [O'r17b]. Each proof manipulates a stack of already proved theorems. Each time an inference rule is applied, one or two theorems are popped from the stack, and after every step, a theorem is pushed.

If invalid steps occur, then the trivial theorem $(v_0 = v_0)$ is pushed. The execution and with it Turing machine will halt if at any point a proof pushes the contradiction formula $(v_0 \in v_0)$.

The resulting Turing machine achieves drastic improvements compared to the machine devised by Yedidia and Aaronson. It comprises 748 states, which is an order of magnitude smaller than the 7910-state Turing machine that relies on Friedman's hypothesis. Therefore, the following theorem:

Theorem 2.1 *There is a* 748*-state Turing machine whose behavior is independent of* $\mathcal{ZF}$*. It halts iff* ZF *is inconsistent. Hence, assuming* $Con(\mathcal{ZF})$*,* $\mathcal{ZF}$ *cannot prove the value of* $\mathbf{BB}(\mathbf{748})$ *.*

3. P vs. NP and Formal Independence

The $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ problem is one of the most significant open issues in modern mathematics and theoretical computer science. It figures among the Clay Math Institute's list of million-dollar prize problems and distinguishes itself as the most prominent and captivating one. This is due to its versatile nature that goes beyond just being a math problem to reaching everything from mathematical reasoning to philosophy to practical computation. Numerous researches were conducted during the past 50 years in pursuit of a definite answer. In light of the indecisive results, a question was raised, whether $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ could somehow be independent of the standard axioms systems of mathematics such as Zermelo-Fraenkel set theory. Several works have already been achieved, each approaching the problem from a particular point of view and producing observations that only affirm its utter harshness. Ineluctably, this problem cannot be addressed in a sole publication let alone a modest part of it. Therefore, we will only be briefly discussing two obstacles that compelled researchers to doubt the qualification of the currently exploited techniques in the journey towards proof of independence. But before that, we succinctly present the $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ problem. The following definitions and concepts are thoroughly explained in the official problem description provided by the Clay Math Institute. [COO71]

Let **P** be the following class of languages.

Definition 3.1 *$\boldsymbol{P} = \{L \mid L = L(M)$ for some deterministic Turing machine M that runs in polynomial time $(n^{\mathcal{O}(1)})\}$.*

The **NP** class of languages only differs from **P** in its non-determinism, as the Turing machine M that accepts a language L could have more than one transition for each configuration. However, **NP** is generally defined with an equivalent statement using the notion of polynomial checking. The following definitions are presented in [Mei20].

Definition 3.2 *For some language L over an alphabet Σ, $L \in \boldsymbol{NP}$ iff L is polynomially verifiable. That is, there is exits an algorithm V so that for all inputs w:*

$w \in L$ iff there exists x so that V on input $\langle w, x \rangle$ accepts. The running time of V on input $\langle w, x \rangle$ is limited by a polynomial in $|w|$. If V accepts on input $\langle w, x \rangle$, then x is called certificate for w.

In other words, **NP** is the class of languages for which we can feasibly (polynomially) check if a candidate solution is proper. Therefore, the $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ problem can be paraphrased into an intuitive question: Could the rapidly checkable languages also be rapidly solvable? That is, conceiving a solution from scratch with a comparable effort to what was devoted during the checking process.

The relevance of finding a definite answer to this problem grows out of the **NP-Completeness** notion devised by Cook and Levin in 1971 and out of the modern complexity-based cryptography, needless to mention the groundbreaking practical consequences $\mathbf{P} = \mathbf{NP}$ might yield.

Definition 3.3 *A language L is NP-Complete iff L is in **NP**, and $L' \leq_p L$ for every language L' in **NP**. $L' \leq_p L$ iff there is a polynomial-time computable function $f : \Sigma_1^* \to \Sigma_2^*$ such that $x \in L' \Leftrightarrow f(x) \in L$, for all $x \in \Sigma_1^*$.*

Properties.

- If $L_1 \leq_p L_2$ and $L_2 \in \mathbf{P}$ then $L_1 \in \mathbf{P}$.
- If L_1 is *NP-Complete*, $L_2 \in \mathbf{NP}$, and $L_1 \leq_p L_2$, then L_2 is *NP-Complete*.
- If L is *NP-Complete* and $L \in \mathbf{P}$, then $\mathbf{P} = \mathbf{NP}$.

As one could already have concluded, *NP-Complete* problems are the hardest in the **NP** class, where coming up with a polynomial-time algorithm for one of them would immediately collapse **P** and **NP**. One such problem is Satisfiability: Given a formula φ in propositional calculus, determine whether there exists a satisfying assignment for φ. A prominent recurring variety of Satisfiability is **3-SAT**, an *NP-Complete* problem whose instances are formulas in conjunctive normal form with only three literals in each clause. For instance, the formula

$$(x \vee y \vee z) \wedge (\bar{x} \vee y \vee \bar{z}) \wedge (x \vee \bar{y} \vee t) \wedge (\bar{x} \vee \bar{z} \vee \bar{t})$$

is in **3-SAT** since the assignment J satisfies the formula, where $J(x) = J(y) = J(t) =$ *True* and $J(\mathrm{z}) =$ *False*.

Since the rise of the concept, numerous *NP-Complete* languages have been discovered, comprising many graph problems (given a graph G and a natural k, is there a subset of k nodes, where all vertices are connected? Can G be colored with k colors, so that

each vertex gets assigned a color that is different from its neighbors? etc.), also the famous Traveling Salesman problem and several others. As far as many theoretical computer scientists are concerned, a strong reason for the $\mathbf{P} \neq \mathbf{NP}$ hypothesis is the thousands of *NP-Complete* problems that have been identified, but, at the same time, the thousands of problems that are shown to be in $\mathbf{P}$. It would have sufficed, that only one *NP-Complete* is at the same time element of $\mathbf{P}$, to immediately collapse $\mathbf{P}$ and $\mathbf{NP}$. Subsequently, the $\mathbf{P} \neq \mathbf{NP}$ hypothesis had thousands of chances to be falsified through the hardness proof reductions, yet, it manages every time, somehow, to avoid the deterministic polynomial-time algorithms.

It's high time we got straight into the topic of the yet unsolved potential formal independence of the $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$. For further readings about $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$, its notability and background, the Clay Math Institute provides an assiduous description [COO71]. We present some of the barriers that make the independence proof harder than we could imagine.

3.1. The Relativization Barrier

A statement is said to relativize or be relative to some oracle A if it holds in respect to A. In this context, an oracle A is a set of natural numbers or strings, for which there is an oracle Turing machine that can simulate all the customary operations of a standard Turing machine and provides A with an instance (natural number). The oracle decides afterward whether the given natural is an element of A. Thereby, the oracle Turing machine will write the characteristic function of A on a distinguished tape, called the oracle tape, for each queried number. In this way, we could define the relativized complexity class C^A for a complexity class C. We say a language L is in C^A if there exists an oracle Turing machine with oracle A, that decides L and in mostly the highest time in C. The approaches discussed in this section were surveyed by Scott Aaronson. [Aar03]

Through oracles, Baker, Gill, and Solovay prove the following. [TBS75]

Theorem 3.1 *There exists at least one oracle* A *such that* $\mathbf{P^A} = \mathbf{NP^A}$*, and at least one other oracle* B *such that* $\mathbf{P^B} \neq \mathbf{NP^B}$*.*

Thanks to the previous result, Hartmanis and Hopcroft show the following.

Theorem 3.2 *There exists a Turing machine* M *that halts on every input, such that neither* $\mathbf{P^{L(M)}} = \mathbf{NP^{L(M)}}$ *nor* $\mathbf{P^{L(M)}} \neq \mathbf{NP^{L(M)}}$ *is provable in* $\mathcal{ZF}$*, assuming* $\mathcal{ZF}$ *is consistent.*

Proof. The approach starts by considering an oracle A relative to which $\mathbf{P} = \mathbf{NP}$ [1], and another oracle B relative to which $\mathbf{P} \neq \mathbf{NP}$ [2]. The existence of such computable oracles follows from the above theorem by Baker, Gill, and Solovay. Now, consider a standard enumeration $P_1, P_2, \ldots$ of $\mathcal{ZF}$ proofs. Let M be a Turing machine that takes an integer y as input and lands on a halting state if either

1. $y \in B$ and there is a proof among $P_1, P_2, \ldots, P_y$ that $\mathbf{P^{L(M)}} = \mathbf{NP^{L(M)}}$, or

2. $y \in A$ and there is a proof among $P_1, P_2, \ldots, P_y$ that $\mathbf{P^{L(M)}} \neq \mathbf{NP^{L(M)}}$.

The construction of M comprises an encoding trick that allows the implication of two contradictory statements. If we take a look at conditions 1 and 2, we'll see that as soon as a proof is found of $\mathbf{P} = \mathbf{NP}$ relative to $L(M)$, the oracle $L(M)$ equals B "from that point forward," as M will accept on input $y \in B$, and B (or anything equal to B except on some finite prefix) makes $\mathbf{P} \neq \mathbf{NP}$. Likewise, as soon as a proof is found that $\mathbf{P} \neq \mathbf{NP}$ relative to $L(M)$, the oracle $L(M)$ equals A from that point forward, as M will accept on input $y \in A$ and A makes $\mathbf{P} = \mathbf{NP}$. Since both possibilities are contradictory, the only way out is that $\mathbf{P} = \mathbf{NP}$ relative to $L(M)$ is neither provable nor disprovable, which is what we wanted to show (and hence, $L(M)$ is the empty oracle, although $\mathcal{ZFC}$ can't prove that). [Aar03]

Intuitively, one would want to generalize and ask: Is it conceivable to create a computable oracle $\mathcal{O}$, relative to which $\mathbf{P} = \mathbf{NP}$ is independent of $\mathcal{ZF}$, regardless of the Turing machine that accepts $\mathcal{O}$?

Hartmanis [HH76] shows that such an oracle exists. The idea is summarized as follows. $\mathcal{O}$ is constructed so that for the majority of input lengths $\mathbf{P}^{\mathcal{O}} = \mathbf{NP}^{\mathcal{O}}$ holds. But, for some extensively distant input lengths, like the output of a total computable function f, $\mathbf{P}^{\mathcal{O}} \neq \mathbf{NP}^{\mathcal{O}}$ applies. And that is enough to decisively affirm $\mathbf{P}^{\mathcal{O}} \neq \mathbf{NP}^{\mathcal{O}}$ since there are infinitely many of these separated input lengths. Now, the problem resides in the fact that f is an extremely rapidly-growing function, so that $\mathcal{ZF}$ is unable to prove its total property or that its definition domain is actually infinite. Therefore, $\mathbf{P}^{\mathcal{O}} \neq \mathbf{NP}^{\mathcal{O}}$ is independent of $\mathcal{ZF}$.

[1] Considering complexity classes hierarchy, we could take any **PSPACE**-Complete language A. It is not hard to see that A collapses $\mathbf{P^A}$ and $\mathbf{NP^A}$ to **PSPACE**, as $\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE}$.

[2] B can be a random oracle, for which we define a special language L, that allows the separation between **P** and **NP**.

3.2. The Natural Proofs Barrier

Since relativization techniques were inherently weak to solve hard questions like $\mathbf{P} \stackrel{?}{=} \mathbf{PSPACE}$, $\mathbf{NP} \subseteq \mathbf{P/poly}$ and $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$, more studies involving combinatorial techniques from the vantage of Boolean circuits cropped up. The idea consists in proving strong lower bounds for complexity classes. The aim is to prove, for any **NP** problem, for instance, **SAT**, that it does not have any polynomial-size circuits. For this, Razborov and Rudich [RR96] introduce the notion of natural proofs and demonstrate, assuming a widely believed cryptographic hardness assumption, that proofs of this kind are unable to prove super-polynomial lower bounds for general circuits, thereby, refuting any strong lower bound proof that could naturalize. The notions and concepts in this part are introduced and thoroughly described in [RR96].

Let's start by defining natural proofs according to Razborov and Rudich. Natural proofs are formalised by a natural combinatorial property, which represents a set of Boolean functions $\{C_n \mid C_n \subseteq F_n, n \in \omega$ and F_n is the set of all Boolean functions on n variables$\}$. Thus, a function f_n will have then property C_n only if f_n is an element of the set C_n. Now, the property C_n is called natural if it contains a subset C_n^* that satisfies the following criteria:

- **Constructivity:** The membership of f_n in C_n is polynomially provable in the length of the truth table of f (was empirically shown to be plausible. Details in [RR96]).
- **Largeness:** $|C_n^*| \geq \frac{1}{2^{n \cdot k}} \cdot |F_n|$.

A combinatorial property is useful against *P/poly* if the following applies.

- **Usefulness:** For any sequence of functions $f_n \in C_n$, the circuit size is super-polynomial.

The largeness condition could be fathomed as the density of C_n^*- That is, a random function $f_n \in F_n$ has a relatively considerable chance of possessing the property C_n.

To better understand the practical significance of the above definitions, consider a natural proof that some explicit function h_n cannot be computed with polynomial-size circuits. Let C_n be the natural combinatorial property used during the proof. Then, the latter will show that every function having the property C_n, including h_n, also does not have polynomial-size circuits, which is equivalent to super-polynomial. That implies the usefulness of C_n. Therefore, if h_n was to be in **NP**, then this constitutes a proof that $\mathbf{P} \neq \mathbf{NP}$. Specifically, such a proof for $\mathbf{P} \neq \mathbf{NP}$ proceeds from the following strategy.

1. Consider a complexity measure for Boolean functions λ that returns high values for certain functions and low values for others. Formalize this notion to a combinatorial property C_n that asserts true for functions with high measure outputs.

2. Show inductively that for any random function f^* computable by polynomial-size Boolean circuits, the complexity measure λ returns a low value. In this way, we show that C_n is useful, because any function in C_n would necessarily need super-polynomial circuits.

3. Show that λ yields a high measure for any **NP** problem. For example, **SAT**, which means that **SAT** has property C_n.

If all steps of the approach are carried out, then we conclude that $\mathbf{P} \neq \mathbf{NP}$. The catch is that Razbov and Rudich give evidence that such a proof strategy is not consistent, as it will prove two contradictory statements simultaneously. They show that a combinatorial property C_n that is natural and useful against $\mathbf{P/poly}$ could be exploited to distinguish any polynomial-time pseudo-random generator from random. That is, it would contravene the generally accepted hypothesis that there exists a pseudo-random generator of hardness $2^{n^{O(1)}}$.

As described by Razborov, Rudich, and Aaronson [Aar03], the idea to show that such a proof is self-crushing is the following: A natural proof that a random function f is not in the class $\mathbf{P/poly}$ would have an algorithm. However, because the proof needs to distinguish f from pseudo-random functions computable by $\mathbf{P/poly}$, the proof algorithm would have then broken the (pseudo)-random function f. The proof is therefore self-crushing, in the sense that it aimed to prove the hardness of f but ended up yielding a solution to solve a hard problem. So, in the end, the algorithm showed the function to be harder than $\mathbf{P/poly}$, and simultaneously gave an algorithm that made it easier!

Within this frame of reference, several fragments of bounded arithmetic formal systems used the same assumption applied in natural proofs. The latter assumes the existence of pseudo-random generator that requires Boolean circuits of size $\Omega(2^{n^\varepsilon})$ to break for some $\varepsilon > 0$. These theories give a lower bound on the length of a proof for Circuit_n where

$$\text{Circuit}_n = \text{"}\mathbf{SAT}_n \text{ requires circuits of size } n^{\log n}\text{"}$$

where $\mathbf{SAT}_n$ comprises **SAT** instances on n variables. Razabov showed that proofs of these theories, which are considerably weaker than $\mathcal{ZF}$, would naturalize.

Consequently, such lower bounds are independent of arithmetic bounded theories (assuming the cryptographic hardness hypothesis).

Regardless of the previous concepts, let's try to conceive an independence proof in a technically mathematical way. For this, let Π_1 be the set of sentences of the form "$\forall\ x$, $Q(x)$," where Q is a recursive predicate or function, whose recursiveness can be proven in Peano Arithmetic. (Peano Arithmetic $\mathcal{PA}$ is a strong theory that encodes arithmetic, it is equivalent to $\mathcal{ZF}$ without the axiom of infinity). In the same way, Π_2 is the set of sentences that have the form "$\forall\ x$, $\exists y$, $Q(x, y)$," and so forth.

Clearly, the **P** vs. **NP** problem can be arranged as a Π_2 sentence: **For all** Turing machines M and polynomials p, **there exists** a **SAT**-instance φ such that the number of steps needed for M to halt with a true answer to φ exceeds $p(n)$. n represents the size of φ (number of variables).

However, Ben David and Halevi show that some Π_1 sentences are equivalent to **P** vs. **NP** [BDH91], despite the latter being a Π_2 statement. Consider the following affirmation A: "**SAT** is uncomputable with Boolean circuits of size $n^{\log n}$". A is thus a Π_1 sentence as we only need to range over all possible values of n and stop if we encounter circuits of size $n^{\log n}$ that compute **SAT**. Obviously, A entails $\mathbf{P} \neq \mathbf{NP}$.

Next, let $\mathcal{PA} + \mathbf{\Pi_1}$ be the theory comprising Peano Arithmetic expanded by the set of all valid Π_1-sentences. Even though this theory is notably strong, it is unable to prove some Π_2 sentences, as in Goodstein's Theorem [SAS87]. Paris and Harrington show that the latter is independent of $\mathcal{PA} + \mathbf{\Pi_1}$, and subsequently of $\mathcal{PA}$.

Now, here is the trick. Suppose **P** vs. **NP** were independent of $\mathcal{PA} + \mathbf{\Pi_1}$- That is, the assertion A devised above is unprovable within this theory. Then A would simply not hold, and **NP** would have Boolean circuits of size $n^{\log n}$. For if it did, then we must have a a Π_1-sentence among $\mathcal{PA} + \mathbf{\Pi_1}$ that implies $\mathbf{P} \neq \mathbf{NP}$, which makes **P** vs. **NP** provable in $\mathcal{PA} + \mathbf{\Pi_1}$, thereby contradicting the assumption. [Aar03]

To summarize, the deeper one digs in the quest of solving this problem, the more knife-twisting the results appear to be, and the clearer a formal independence proof of **P** vs. **NP** seems to be out of reach of the current techniques.

4. Summary and Outlook

With the definition of the Busy Beaver Game in 1961, Tibor Radó initiated a competition that is still relevant today. In order to win this game, the obstacle of uncomputability must be overcome. This hurdle challenges computer scientists time and again to find creative approaches for identifying the Busy Beavers. However, in light of this quest, an equally challenging but less prominent problem emerges: Could the Busy Beaver values, at some point, elude the foundations of moderns mathematical theories? This general concept of unknowability is significantly more intriguing than the prospect of pinning down Busy Beaver values. In this paper, we have presented a programming language devised by Adam Yedidia, with which two Turing machines were developed, whose behaviors cannot be proven using the axioms of a certain theory, assuming its consistency- that is, the corresponding Busy Beaver values of these machines are independent of set theory. In other words, we have given an upper bound on the highest provable Busy Beaver value in $\mathcal{ZF}$, which is $\mathbf{BB(747)}$. Besides that, we used Laconic to prove the undecidability of several other Busy Beaver values. In the end, we touched on an astronomically hard problem that dwells around the formal independence of the **P** vs. **NP**. There, we altercated two barriers that only but accentuate the utter harshness of an independence proof. Nevertheless, one could ironically affirm that it is believable that the independence results achieved so far will eventually subsidize $\mathbf{P} \neq \mathbf{NP}$, not because of what they conclude, but because of the new techniques developed throughout the process.

Even though the achievements of Stefan O'rear present a stupendous improvement upon the works of Yedidia and Aaronson, the presently obtained value n for which $\mathbf{BB(n)}$ is independent of set theory, namely 748, is still 2 orders-of-magnitude higher than the largest n, namely 4, for which $\mathbf{BB(n)}$ is known to be determinable. It is, therefore, legitimate to try pinning down the Gödelian boundaries of ZF- the axiomatic system underpinning almost all modern math. Is it conceivable that the smallest n for which $\mathbf{BB(n)}$ is independent of set theory is far smaller than 748? Could it be $\mathbf{BB(20)}$ or even $\mathbf{BB(6)}$? In his frontier about Busy Beaver, Scott Aaronson [Aar03] sticks his neck out and conjectures that $\mathbf{BB(20)}$ and $\mathbf{BB(10)}$ are unprovable in $\mathcal{ZF}$ and $\mathcal{PA}$, respectively. Whether near or far, such thresholds of unknowability definitely exist.

This restriction is what Gödel's incompleteness theorems inflicted to the all axiomatic systems encoding arithmetic. It might turn out, as well, that showing whether $\mathcal{ZF}$ proves $\mathbf{BB}(\mathbf{20})$ would be an unimaginably harsher task than solving the $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ problem, which is already one of the most challenging mathematical problems.

A. Appendices

Most implemented programs are available through the presented link to a Github repository. Yet, the Scholz conjecture's code is explicitly given as an example.

A.1. Laconic Program: The Scholz Conjecture

```
 func zero(x){
    x = 0;
    return;
}

func one(x){
    x = 1;
    return;
}

func incr(x){
    x = x + 1;
    return;
}

func modulus (x , y , out ) {
    out = x;
    while ( out >= y ) {
        out = out - y;
    }
    return;
}

func pow(b, e, result,i){

    one(result);
    if (e == 0) {
        result = 1;
        return;
    }
    if ( e== 1){
        result = b;
        return;
    }
    one(i);
    while ( i <= e){
```

```
        result = result * b;
        incr(i);
    }
    return;
}

func contains(l, value, i, found, length, element){

    zero(i);
    zero(found);
    length = #l;
    while (i < length){
        element = l@i;
        if ( value == element){
            one(found);
            return;
        }
        incr(i);
    }
    return;
}


func additionChain(n, allChains, chain, i, j, k, h, sum, found,foundLength,
finalLength, lengthFoundChain, lengthChain,isContained, bool, element,
lastElement, chainLength, length, firstSummand,
secondSummand, chainToAppend,tmp) {

    allChains = :[1,2]:;
    chain = [];
    one(bool);
    zero(k);
    while (bool & !found){
        chain = allChains@*k;
        chainLength = #chain;
        zero(i);
        while (i < chainLength){
            firstSummand = chain@i;
            j = i + 1;
            tmp = chainLength-1;
            if (i == tmp){
            j = i;
            }
            while ( j < chainLength){

                chainToAppend = chain;
                secondSummand = chain@j;
                lastElement = chain@tmp;
                sum = firstSummand + secondSummand;
                zero(found);
                /*print sum;*/
                contains(chain, sum, h, found, length, element);
                /*print found;*/
                if ( !found ){
                    zero(found);
```

```
                    if (sum>lastElement) {
                    chainToAppend = chainToAppend^sum;
                    allChains = allChains^*chainToAppend;
                    if (sum == n){
                        finalLength = #chainToAppend - 1;
                        return;
                    }
                    }
                }
                incr(j);
                }
            incr(i);
        }
    incr(k);
    }
    return;
}

int n;
list2 allChains;
list chain;
int i;
int j;
int k;
int h;
int y;
int sum;
int found;
int greaterFound;
int bool;
int element;
int lastElement;
int chainLength;
int length;
int firstSummand;
int secondSummand;
int chainToAppend;
int tmp;
int foundLength;
int lengthFoundChain;
int lengthChain;
int isContained;
int conjectureVerified;
int finalLength;
int lengthOfAdditionChainOfN;
int lengthOfAdditionChainOfPowerOfN;
int result;
int rightPart;

y = 2;

n = 3;
one(conjectureVerified);
while (conjectureVerified) {
    print n;
```

```
    additionChain(n, allChains, chain, i, j, k, h, sum, found ,foundLength, finalLength,
        lengthFoundChain, lengthChain,isContained, bool, element,lastElement,
        chainLength, length, firstSummand, secondSummand, chainToAppend,tmp);

    lengthOfAdditionChainOfN = finalLength;
    print lengthOfAdditionChainOfN;
    pow(y, n, result,i);
    result = result - 1;
    print result;

    additionChain(result, allChains, chain, i, j, k, h, sum, found ,foundLength,
        finalLength, lengthFoundChain, lengthChain,isContained, bool, element,
        lastElement, chainLength, length, firstSummand, secondSummand,
        chainToAppend,tmp);

    lengthOfAdditionChainOfPowerOfN = finalLength;
    print lengthOfAdditionChainOfPowerOfN;
    rightPart = n - 1 + lengthOfAdditionChainOfN;
    print rightPart;

    if (lengthOfAdditionChainOfPowerOfN > rightPart){
        zero(conjectureVerified);
    }
    print conjectureVerified;
    incr(n);
}
halt;
```

A.2. Laconic Program: The Lychrel Numbers

This is a link to the Laconic program encoding the Lychrel conjecture.

A.3. Laconic Program: The Lehmer's Totient Problem

This is a link to the Laconic program encoding the Lehmer's totient problem.

A.4. Laconic Program: The Total Coloring Conjecture

This is a link to the Laconic program encoding the total coloring conjecture.

A.5. Laconic Program: The Erdős–Gyárfás Conjecture

This is a link to the Laconic program encoding the Erdős–Gyárfás conjecture.

List of Figures

2.1. naive states allocation of the printer for program binary 15
2.2. an introspective implementation example of the printer with data states [YA16] 15
2.3. non leaf nodes represent partial addition results 23
2.4. Flowchart of enumerating k-cycles algorithm 33

Bibliography

[Aar03] Aaronson, Scott: Is P Versus NP Formally Independent? (2003). https://www.scottaaronson.com/papers/indep.pdf

[Aar20] Aaronson, Scott: The Busy Beaver Frontier. (2020). https://www.scottaaronson.com/papers/bb.pdf

[BDH91] Ben-David, S. ; Halevi, S.: On The Independence of P Versus NP. (1991). https://www.cs.technion.ac.il/users/wwwb/cgi-bin/tr-get.cgi/1991/CS/CS0699.revised.pdf

[Beh65] Behzad, Mehdi: Graphs And Their Chromatic Numbers. (1965). https://www.proquest.com/docview/302172694?pq-origsite=gscholar&fromopenview=true

[Bol80] Bollobás, Béla: A Probabilistic Proof of an Asymptotic Formula for the Number of Labelled Regular Graphs. In: *European Journal of Combinatorics* 1 (1980), 311-316. https://www.sciencedirect.com/science/article/pii/S0195669880800308

[Cli10] Clift, Neill M.: Calculating optimal addition chains. (2010). https://doi.org/10.1007/s00607-010-0118-8

[COO71] COOK, STEPHEN: THE P VERSUS NP PROBLEM. (1971). https://www.claymath.org/sites/default/files/pvsnp.pdf

[Fri14] Friedman, Harvey M.: Order Invariant Graphs And Finite Incompleteness. (2014). https://cpb-us-w2.wpmucdn.com/u.osu.edu/dist/1/1952/files/2014/01/FIiniteSeqInc062214a-v9w7q4.pdf

[HH76] Hartmanis, J. ; Hopcroft, J.E.: Independence Results In Computer Science. (1976). https://ecommons.cornell.edu/handle/1813/6063

[HHR98] Hugh Hind, Micheal M. ; Reed, Bruce: TOTAL COLORING WITH + poly(log) COLORS. (1998). https://epubs.siam.org/doi/abs/10.1137/S0097539795294578?journalCode=smjcat

[Hol19] Holm, Thorsten: *Diskrete Strukturen für Studierende der Informatik*. lecture slides, 2019

[Kha20] Khakhalin, Arseny: *Draw all graphs of N nodes*. https://github.com/sorear/metamath-turing-machines, 2020. – [Accessed: 2021-07-30]

[Lia11] Lian, Tony: Fundamentals OF ZERMELO-FRAENKEL Set Theory. (2011). https://www.math.uchicago.edu/~may/VIGRE/VIGRE2011/REUPapers/Lian.pdf

[LW06] Liu, Hongbo ; Wang, Jiaxin: A new way to enumerate cycles in graph. In: *Advanced Int'l Conference on Telecommunications and Int'l Conference on Internet and Web Applications and Services (AICT-ICIW'06)*, 2006, S. 57–57

[mat10] mathworld: *Magnitude of Graham's Number?* https://mathoverflow.net/questions/11934/magnitude-of-grahams-number, 2010. – [Accessed: 2021-07-25]

[Mei20] Meier, Arne: *Komplexität von Algorithmen*. Preparationsheet, 2020

[Met] Metamath contributors: *Metamath Proof Explorer Home Page*. http://us.metamath.org/mpeuni/mmset.html

[O'r17a] O'rear, Stefan: *metamath-turing-machines*. https://github.com/sorear/metamath-turing-machines, 2017

[O'r17b] O'rear, Stefan: *metamath-turing-machines*. https://github.com/sorear/metamath-turing-machines/blob/master/zf2.nql, 2017

[PDS81] Peter Downey, Benton L. ; Sethi, Ravi: COMPUTING SEQUENCES WITH ADDITION CHAINS. (1981). https://epubs.siam.org/doi/pdf/10.1137/0210047

[RAD61] RADO, T.: On Non-Computable Functions. (1961). https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6769603

[RR96] Razborov, Alexander A. ; Rudich, Steven: Natural Proofs. (1996). http://www.cs.umd.edu/~gasarch/BLOGPAPERS/natural.pdf

[SAS87] Stuart A.Kurtz, Michael J. ; S.Royer, James: How to prove representation-independent independence results. (1987). https://www.sciencedirect.com/science/article/pii/0020019087901918

[SW98] STEGER, A. ; WORMALD, N.C.: Generating Random Regular Graphs Quickly. (1998). https://www.cambridge.org/core/journals/combinatorics-probability-and-computing/article/generating-random-regular-graphs-quickly/49CE556102BCF231EBC489CB2C2A7E65

[TAR64] TARSKI, ALFRED: A SIMPLIFIED FORMALIZATION OF PREDICATE LOGIC WITH IDENTITY. In: *Archiv für mathematische Logik und Grundlagenforschung* 7 (1964), 61–79. https://link.springer.com/article/10.1007/BF01972461

[TBS75] THEODORE BAKER, JOHN G. ; SOLOVAY, ROBERT: RELATIVIZATIONS OF THE P =? NP QUESTION. (1975). http://cse.ucdenver.edu/~cscialtman/complexity/Relativizations%20of%20the%20P=NP%20Question%20(Original).pdf

[TUR36] TURING, A. M.: ON COMPUTABLE NUMBERS. (1936). https://www.cs.virginia.edu/~robins/Turing_Paper_1936.pdf

[Vol20] VOLLMER, Heribert: *Grundlagen der Theoretischen Informatik*. lecture slides, 2020

[Wika] WIKIPEDIA CONTRIBUTORS: *Erdős–Gyárfás conjecture*. https://en.wikipedia.org/wiki/Erd%C5%91s%E2%80%93Gy%C3%A1rf%C3%A1s_conjecture. – [Online; accessed 25-July-2021]

[Wikb] WIKIPEDIA CONTRIBUTORS: *Graph coloring*. https://en.wikipedia.org/wiki/Graph_coloring. – [Online; accessed 10-July-2021]

[Wikc] WIKIPEDIA CONTRIBUTORS: *Lychrel number*. https://en.wikipedia.org/wiki/Lychrel_number. – [Online; accessed 20-July-2021]

[Wikd] WIKIPEDIA CONTRIBUTORS: *Russell's Paradox*. https://en.wikipedia.org/wiki/Russell%27s_paradox. – [Online; accessed 1-August-2021]

[Wike] WIKIPEDIA CONTRIBUTORS: *Total Coloring*. https://en.wikipedia.org/wiki/Total_coloring. – [Online; accessed 10-July-2021]

[YA16] YEDIDIA, Adam ; AARONSON, Scott: A Relatively Small Turing Machine Whose Behavior Is Independent of Set Theory. (2016). https://arxiv.org/pdf/1605.04343.pdf

[Yed16] Yedidia, Adam: *parsimony*. `https://github.com/adamyedidia/parsimony`, 2016