

New Methods for Analyzing the Properties of Automatic Sequences

by

Mazen Khodier

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2026

© Mazen Khodier 2026

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners. I understand that my thesis may be made electronically available to the public.

Statement of Contributions

This thesis contains content from two papers of which I, Mazen Khodier, am a co-author. The first is an unpublished joint manuscript written with Gabriele Fici and Jeffrey Shallit [22], and the second is a joint work with Jeffrey Shallit and Luke Schaeffer, which has both an early arXiv version [30] and a conference proceeding version [31].

Sections 1.1 and 1.2 and Chapters 2 to 4 are based on [30, 31] and Chapter 6 and section 7.1 are based on [22] and portions of these works are extracted verbatim into this thesis.

Abstract

Automatic sequences and morphic words lie at the intersection of automata theory, logic, and combinatorics on words. Many of their structural properties can be formulated as logical predicates over integer representations and decided using automata. This thesis presents automata-based methods for efficiently constructing and verifying deterministic finite automata corresponding to such predicates, and builds on this foundation to analyze key combinatorial properties of morphic words, including the critical exponent and subword complexity.

In the first part of this thesis, Chapters 2 to 4, we introduce the notion of *self-verifying predicates*, which are logical predicates capable of verifying their own correctness. We show how this property enables verification of candidate automata through a small set of inductive conditions and allows the corresponding automata to be constructed deterministically rather than through heuristic guessing. Building on Angluin’s L^* learning algorithm, we demonstrate that for such predicates, the associated minimal automata can be generated in time polynomial in the size of both the automaton for the underlying sequence and the resulting automaton, thereby avoiding potentially extremely large intermediate automata that sometimes arise in Walnut. In particular, we give effective constructions for the *equality-of-factors* predicate, which is used extensively in the second half of the thesis, as well as for other *self-verifying predicates*, including periodicity of factors, addition relations for numeration systems, and summation of synchronized sequences.

The second part, Chapters 5 to 7, applies the previously constructed equality-of-factors predicate to investigate two central combinatorial measures of infinite words: the *critical exponent* and the *subword complexity*. Although binary 3-uniform morphisms are used as illustrative examples, the methods generalize naturally to all binary uniform morphisms. For the critical exponent, we present a decision procedure implemented in Walnut that detects whether the exponent is infinite and computes its exact rational value when finite. For subword complexity, we propose two complementary approaches: a constructive method that combines established concepts to produce exact formulas for $\rho(n)$, and a fully deterministic procedure that implements Frid’s approach using Walnut. The new results include explicit subword-complexity formulas for twelve morphisms, and critical-exponent values for ten morphisms.

All algorithms and implementations developed in this thesis are made publicly available on the Github repository [Cashew](#) as open-source code to support and facilitate further research in combinatorics on words, and automata theory.

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Jeffrey Shallit, for his guidance, support, immense patience, and for everything he has given me over the past two years of my graduate studies. His insight and generosity with his time have shaped every stage of this thesis, and I am truly grateful to have had the opportunity to learn from him.

I would like to acknowledge my co-authors Gabriele Fici, Luke Schaeffer, and Jeffrey Shallit, and I am grateful for their collaboration and the interesting ideas they shared that form the basis of the work presented here. I would also like to extend my appreciation to my readers, Luke Schaeffer and Jason Bell, for taking the time to review this thesis.

Finally but most importantly, I would like to thank God for everything, especially for my family, friends, and all my teachers throughout my life, whose constant support made this thesis possible.

Table of Contents

Author’s Declaration	ii
Statement of Contributions	iii
Abstract	iv
Acknowledgements	v
List of Figures	ix
List of Tables	x
1 Introduction	1
1.1 Büchi arithmetic	1
1.2 Walnut software	2
1.3 Thesis outline	5
2 Angluin’s algorithm	6
3 Self-verifying predicates	9
3.1 Equality of factors	9
3.1.1 Membership queries	10
3.1.2 Hypothesis queries	12

3.2	Equality of reversals of factors	14
3.3	Periods of factors	15
3.4	Creating an adder for a numeration system	16
3.5	Summation of synchronized sequences	17
4	Implementation details and results	21
4.1	Membership queries	21
4.2	Optimizing performance	22
4.3	Walnut considerations	23
5	Binary 3-uniform morphisms	25
5.1	Motivation	25
5.2	Formal definitions	25
5.3	Studied morphisms	26
6	Critical exponent	29
6.1	Checking for infinite critical exponent	29
6.2	Continued fraction search	31
7	Subword complexity	34
7.1	Heuristic approach	34
7.1.1	Upper bound using right-special factors	34
7.1.2	Drawing $\rho(n)/n$ using linear representation	36
7.1.3	Deduce a conjecture on the subword complexity	37
7.1.4	Proving the conjecture using Walnut	38
7.1.5	Proving the conjecture using linear representations	39
7.2	Frid's formula	42
7.2.1	Preliminary definitions	42
7.2.2	The algorithm and an example: the 3-adic word	45

8 Open problems	47
References	51

List of Figures

3.1	Equality of factors predicate for the Thue-Morse sequence	15
3.2	Partial sum automaton for the Thue-Morse sequence	19
3.3	Partial sum automaton for the Fibonacci word	19
7.1	Plot of $\rho_{3\text{-adic}}(n)/n$	37
7.2	First few novel factors of the 3-adic sequence using <code>tdcsubc</code>	40

List of Tables

4.1	Performance of the EqFac predicate on four automatic sequences.	23
4.2	Performance of the partial and rarefied sum predicates on different automatic sequences.	23
5.1	Table of aperiodic fixed points of binary 3-uniform morphisms.	27

Chapter 1

Introduction

1.1 Büchi arithmetic

Let $k \geq 2$ be a fixed integer, and let $V_k(n) = \sup\{k^i : k^i \mid n\}$. The first-order logical theory $\langle \mathbb{N}, +, <, 0, 1, V_k \rangle$ is commonly referred to as *Büchi arithmetic* [10]. This theory extends Presburger arithmetic and possesses sufficient expressive power to represent finite automata, while remaining algorithmically decidable [9]. It is important to note that while addition of integer variables is permitted within this framework, subtraction, multiplication and division of variables are not. However, a workaround to allow subtraction is to reformulate it into an addition operation. For example, since the subtraction operation $x-y$ is not a primitive one in Büchi arithmetic, we can reformulate $z = x-y$ into $z+y = x$. Of course, the result is meaningful only when $x \geq y$ in this case. Furthermore, multiplication and integer division by fixed constants are allowed, as they can be expressed through repeated additions or subtractions, respectively. For instance, multiplication by 2 is expressible as $n = x+x$ and for division by 3, we can use the predicate $\exists t (t < 3) \wedge x = n+n+n+t$.

Büchi arithmetic is quite useful in combinatorics on words, since it can be used to decide many claims about automatic sequences, provided the underlying numeration system is addable [39]. A sequence $(a(n))_{n \geq 0}$ is said to be *automatic* if it takes its values in a finite set, and there is a finite automaton that, on input n expressed in the numeration system \mathcal{N} , reaches a state with output $a(n)$; see [1]. An *addable numeration system* \mathcal{N} is one for which a finite automaton can compute the addition relation $x + y = z$, where x, y, z are natural numbers expressed in \mathbb{N} .

More precisely, the fundamental theorem of Büchi arithmetic is the following:

Theorem 1. *There exists an algorithm that takes a first-order logical formula φ about an automatic sequence in an addable numeration system, phrased in terms of addition and indexing of integer variables, and computes an automaton that accepts exactly those values of the free (unbound) variables that make φ true. If there are no free variables, the computed automaton is a single state that accepts everything (TRUE) or rejects everything (FALSE).*

Example 1. Given an automatic sequence $\mathbf{a} = (a(n))_{n \geq 0}$ defined on an addable numeration system, it is decidable whether \mathbf{a} is *squarefree*; that is, whether it contains no *squares*, which are two consecutive identical nonempty blocks. This follows because the property of not having a square is first-order expressible:

$$\neg \exists i, n (n \geq 1) \wedge \forall t (t < n) \implies a(i+t) = a(i+n+t).$$

However, the worst-case running time of the decision procedure is truly formidable; it is of the form

$$2^{2^{\dots 2^{p(N)}}},$$

where the number of 2's in the exponent is equal to the number of quantifier alternations in the formula, p is a polynomial, and N is the size of the logical formula. This is because the algorithm depends on repeated applications of the subset construction from automata theory [25], each one of which can potentially result in an exponential blowup in the number of states. Nevertheless, the procedure is, rather surprisingly, often very useful in practice despite this terrible worst-case running time [40, 17, 43, 3, 39].

In Büchi arithmetic, integers are represented as words x over a finite alphabet Σ^* . To represent j -tuples of integers (n_1, \dots, n_j) , where n_i is the integer whose representation is the projection of the i -th coordinate of x , we use the alphabet Σ^j . This may require padding shorter inputs with leading zeros. If $x \in (\Sigma^j)^*$, we let $[x]$ denote the tuple of integers represented by x . The canonical representation of an integer $n \in \mathbb{N}$ (that is, the one without leading zeros) is written (n) . This is generalized to j -tuples by writing (n_1, n_2, \dots, n_j) . Throughout this work, we assume representations of integers are read with the most significant digit first.

1.2 Walnut software

An automaton-based decision procedure for Büchi arithmetic, as well as certain related theories based on other kinds of representations, has been implemented in the free software

`Walnut` [33]. It has been used in over a hundred research papers and books to confirm old results, correct mistakes in the literature, and prove entirely new results [39].

The decision procedure for Büchi arithmetic implemented in `Walnut` supports integer representation in various addable numeration systems, including base- k for integers $k \geq 2$, the Fibonacci (Zeckendorf) numeration system, the Tribonacci numeration system, and others. `Walnut` also allows users to define their own custom numeration systems.

Given a first-order logical formula φ , `Walnut` produces a deterministic finite automaton (DFA) accepting the values of free variables that make φ true. Furthermore, we are guaranteed that the automata that `Walnut` computes are minimal. `Walnut` can also produce deterministic finite automata with output (DFAOs), where the output is a specified function of the final state reached.

In some cases, `Walnut` returns no response to a query because its limit on space is exceeded (roughly, that automata cannot have more than 2^{32} transitions.) This can be the case even if the final result would be small, because intermediate results can be extremely large. This can occur even with a single quantifier and a very simple formula.

To illustrate the kinds of space consumption that can occur, consider the following first-order logical formula for equality of factors of an infinite word \mathbf{x} :

$$\text{EqFac}(i, j, n) := \forall t (t < n) \implies \mathbf{x}[i + t] = \mathbf{x}[j + t].$$

It is true precisely when the length- n factor beginning at position i is the same as that beginning at position j . Note that in this predicate, and all others we discuss in this thesis, the domain of integer variables is assumed to be $\mathbb{N} = \{0, 1, 2, \dots\}$, the natural numbers. Moreover, the need for Büchi arithmetic, as opposed to the simpler Presburger arithmetic, arises because we need to be able to express the i 'th term of an automatic sequence $\mathbf{X}[i]$.

Nevertheless, since `Walnut` implements Büchi arithmetic, it can construct an automaton evaluating this logical formula for automatic sequences \mathbf{x} . For example, for the Thue-Morse sequence, the resulting automaton takes the inputs i, j, n in parallel, has 14 states, and the largest intermediate automaton formed during the computation has 408 states. To construct it, one can use the `Walnut` command

```
def tm_eqfac "At (t<n) => T[i+t]=T[j+t]":
```

However, when we carry out the same construction for the so-called Tribonacci word [16, 45] `tr` = 010201001020101020100102 \dots , using the `Walnut` command

```
def trib_eqfac "?msd_trib At (t<n) => TR[i+t]=TR[j+t]":
```

the largest intermediate automaton has 323,831,403 states (!), while the final result has only 26 states.¹

Therefore, it is desirable to have alternative methods to compute some of these basic and useful automata. In this thesis, we develop a new technique for this, of both theoretical and practical interest. It uses the observation that some of the most useful predicates are “self-verifying” in a certain sense. In some cases, we can prove that our new algorithm runs in time polynomial in the size of the automaton computing the original sequence \mathbf{x} and the size of the final result. In particular, the idea is based on Angluin’s algorithm, discussed in Section 2.

While the concept of using self-verifying predicates to verify automata has been explored in previous work [35, 41], the novelty of our approach lies in eliminating the need to “guess” the automaton to be verified. Instead, the automaton is, so to speak, “built from the ground up” in a completely deterministic fashion. More precisely, our main result is the following:

Theorem 2. *If \mathbf{x} is an automatic sequence and P is a self-verifying property, then we can construct a minimal automaton A_P that recognizes the predicate corresponding to the property P in time polynomial in the size of the automaton for \mathbf{x} and the automata for the predicates that prove the self-verifying property P .*

For the second part of our thesis, we have two main results related to the critical exponent and the subword complexity of the fixed points of binary uniform morphisms. These results are proved in Sections 6 and 7, and can be stated as follows:

Theorem 3. *For any binary uniform morphism h , with a fixed point \mathbf{x} , we can deterministically check whether the critical exponent $\alpha = \text{ce}(\mathbf{x})$ is infinite. If it is finite, then it has a rational critical exponent $\alpha = \frac{a}{b}$, and we have an algorithm to calculate α using $\text{poly}(\log(\max(a, b)))$ predicates. Moreover, using **Walnut**, we can compute the subword complexity of this fixed point $\rho_{\mathbf{x}}(n)$ for all values of n .*

The final contribution of this thesis is the open-source release of all methods developed and applied here, provided in the GitHub repository [Cashew](#) to support reproducibility and future research.

¹For this particular example, some reformulations of the logical predicate result in much smaller intermediate automata.

1.3 Thesis outline

The current chapter introduced the background and motivation for this work, including Büchi arithmetic and the `Walnut` software system, which form the foundation for the entire work.

Chapter 2 gives an overview of Angluin’s (L^*) learning algorithm and provides essential context for the methodology of automata construction used to build predicates that possess the property of “self-verification.”

Chapter 3 explores those self-verifying predicates, demonstrating how some automata can be used to prove their own correctness. Using this methodology, we construct automata for testing equality of factors, equality of a factor and the reversal of another factor, periods of factors, and other useful predicates.

Chapter 4 discusses implementation details of this approach, presents experimental results, and outlines optimization strategies. Building on these foundations, the remainder of the thesis demonstrates how some of the automata we constructed, in combination with `Walnut`, can be used to derive interesting properties of infinite words.

Chapter 5 introduces the two properties of infinite words in which we are interested for the remainder of the thesis, alongside the necessary formal definitions needed to understand them. These properties are the critical exponent of an infinite word and its subword complexity. The chapter also provides a complete description of both properties for all of the fixed points of the relevant binary 3-uniform morphisms.

Chapter 6 provides an in-depth discussion of the critical exponents of automatic sequences, including a deterministic algorithm for computing them for fixed points of uniform morphisms using `Walnut`.

Chapter 7 discusses subword complexity and outlines a practical approach with the steps required to do rigorous analysis that helps in computing the subword complexity for infinite automatic sequences using either `Walnut` or linear representations. It also discusses Anna Frid’s method [18] which provides a complete description of the subword complexity of fixed points of uniform morphisms.

Finally, Chapter 8 presents relevant open problems and directions for future research that arise from the results and methods discussed in this thesis.

Chapter 2

Angluin's algorithm

Dana Angluin developed an algorithm [4], sometimes called the “ L^* algorithm”, that allows a *learner* to infer a finite automaton for a regular language $L \subseteq \Sigma^*$ from examples and counterexamples. In her algorithm, the learner interacts with a *teacher* in two different ways.

The learner can present a word $w \in \Sigma^*$ to the teacher and ask if $w \in L$; this is called a *membership query*. The learner can also present an hypothesized automaton A for the language L and ask if $L(A) = L$. If the teacher answers positively, the learner has now found an automaton for L and the algorithm terminates. If, on the other hand, the teacher answers negatively, they provide a counterexample to the learner; that is, a member of the symmetric difference $(L \setminus L(A)) \cup (L(A) \setminus L)$. We call all of this a *hypothesis query*. If the minimal automaton for L has N states, then the total number of membership queries and hypothesis queries, and their size, is bounded by a polynomial in N . More precisely, her result is the following:

Theorem 4 ([4]). *There exists an algorithm L^* that learns a regular language L by performing membership and hypothesis queries. If all negative hypothesis queries return counterexamples of length at most m and the regular language L has a minimal DFA of n states, then the total running time is polynomial in m and n .*

The algorithm maintains an *observation table* (S, E, T) , where S is a finite prefix-closed set of words, E is a finite suffix-closed set of words, and $T : (S \cup S\Sigma) \cdot E \rightarrow \{0, 1\}$ is a map that records the answers to membership queries in L . *Prefix-closed* means every prefix of S is in S , and similarly for the suffix-closed property of E . Thinking of T as a table with rows labeled by $S \cup S\Sigma$ and columns labeled by E , the table T is said to be closed if for

each $t \in S\Sigma$ there is an $s \in S$ such that the row labeled s equals the row labeled t . It is called *consistent* if whenever two rows are equal, labeled s_1 and s_2 , say, then the row labeled s_1a equals that labeled by s_2a , for all $a \in \Sigma$.

If both properties hold, the observation table (S, E, T) defines a deterministic finite automaton $(Q, \Sigma, q_0, F, \delta)$ in a canonical way. Each distinct row of the table corresponds to a state of the automaton. Formally, the state set is $Q = \{\text{row}(s) : s \in S\}$. The initial state is the row corresponding to the empty word, $q_0 = \text{row}(\epsilon)$. A state $\text{row}(s)$ is accepting and in F if and only if $T(s) = 1$. Finally, for each $s \in S$ and $a \in \Sigma$, the transition from the state $\text{row}(s)$ on input a is given by $\delta(\text{row}(s), a) = \text{row}(sa)$.

Starting from the empty word and the letters of the input alphabet, the algorithm repeatedly repairs violations of closedness or consistency by extending S or E , respectively, until a valid DFA can be constructed. This DFA is then submitted as a conjecture. If a counterexample is returned, it is incorporated into the table and the process continues. The algorithm terminates once the conjectured automaton is accepted by the teacher, at which point it is guaranteed to be minimal. The pseudocode for this process is provided in Algorithm 1.

Algorithm 1 Angluin's L^* Algorithm

Input: Membership and equivalence query access, and the alphabet Σ .
Output: Minimal DFA for the target regular language.

- 1: Initialize $S \leftarrow \{\Sigma \cup \epsilon\}$, $E \leftarrow \{\epsilon\}$
- 2: **repeat**
- 3: **while** table (S, E, T) is not closed or not consistent **do**
- 4: **if** table is not closed **then**
- 5: Find $s \in S$, $a \in \Sigma$ such that sa has a new row
- 6: Add sa to S and update T
- 7: **end if**
- 8: **if** table is not consistent **then**
- 9: Find $s_1, s_2 \in S$, $a \in \Sigma$, $e \in E$ witnessing inconsistency
- 10: Add ae to E and update T
- 11: **end if**
- 12: **end while**
- 13: Construct DFA M from (S, E, T)
- 14: Submit M as an equivalence query
- 15: **if** a counterexample w is returned **then**
- 16: Add all prefixes of w to S and update T
- 17: **end if**
- 18: **until** M is accepted
- 19: **return** M

Normally the teacher and the learner are separate entities. However, for certain kinds of languages, corresponding to certain logical formulas about automatic sequences, we can use Theorem 1 to construct an algorithm that plays the role of *both* learner and teacher, as we will see in the next section. Currently we have no general theory characterizing exactly for which kinds of logical formulas our technique works. Nevertheless, the general idea is widely applicable and adaptable to a number of situations, as we intend to show in the following sections.

Chapter 3

Self-verifying predicates

Informally, we say that a predicate $P(n, x, \dots, z)$ is *self-verifying* if one can prove its correctness using P itself by induction on n , together with proofs of a finite collection of statements. These auxiliary statements serve as the basis for the induction proof, and they involve small values of n in addition to easily checkable Boolean conditions on the variables x, y, \dots, z . If the idea is not immediately clear, the reader will be able to deduce the conceptual meaning from our first example, the equality of factors predicate.

3.1 Equality of factors

One of the most useful of all predicates for understanding the properties of an infinite word \mathbf{x} is EqFac, the *equality of factors* predicate we saw in Section 1.1; that is, given integers i, j, n , determine whether the length- n factors beginning at positions i and j are the same. If we want to understand aspects of \mathbf{x} such as factor complexity (aka subword complexity), the number of distinct blocks of length n appearing in \mathbf{x} , then finding an automaton for this EqFac is a critical building block.

However, because of the appearance of the \forall quantifier, the best estimate we can find for the complexity of finding the automaton using Theorem 1, without more detailed analysis, is an exponential upper bound on the size of the resulting automaton. We can potentially see exponential blowup in intermediate automata, even if the final result is small.

In this section, by combining Angluin's algorithm with a self-verifying predicate, we prove the following result:

Theorem 5. *If \mathbf{x} is an automatic sequence, we can construct a minimal automaton A for the EqFac predicate in time polynomial in the size of the automaton for \mathbf{x} and the size of A .*

Of course, by “polynomial time” we mean polynomial in the number of bits of the integers involved. Also, it is important to distinguish this measure from the *state complexity* measure, which is only concerned with the number of states in the minimal automaton. In contrast, our use of polynomial time refers to the computational effort required to *construct* the minimal automaton which, by extension, also includes the time needed to generate any intermediate automata of varying sizes. For instance, as previously mentioned in Section 1.2 while Walnut outputs minimal automata, it may generate very large intermediate automata during computation. Hence, our notion concerns not only state complexity but also computational complexity.

For the equality of factors problem, the language L in Angluin’s algorithm is

$$L = \{x \in \mathcal{V}_3 : [x] = (i, j, n) \text{ and } \text{EqFac}(i, j, n)\},$$

where \mathcal{V}_j is the set of all valid representations of integer j -tuples in the underlying numeration system. For example, if we are representing integers in base 2, then $\mathcal{V}_3 = (\{0, 1\}^3)^*$. In particular, we note that a standard convention is that if an input is accepted, then so are all inputs that start with an arbitrary number of leading zeros. We enforce this by insisting that from the initial state, a transition on the input $[0, 0, \dots, 0]$ always leads back to the initial state.

3.1.1 Membership queries

We now explain how the membership queries for L required by Angluin’s algorithm can be carried out in polynomial time for EqFac.

Given x , the first thing we check is that we have used only legal representations in the underlying numeration system. For example, if we are working in the Fibonacci numeration system, we have to check that the projection of x into the binary strings representing three integers i_0, j_0, n_0 have no two consecutive 1’s. If any of them do have two consecutive 1’s, then $x \notin L$.

Now we know that the input is a legal representation of a triple of integers (i_0, j_0, n_0) . We want to determine if $\mathbf{x}[i_0..i_0 + n_0 - 1] = \mathbf{x}[j_0..j_0 + n_0 - 1]$.

A small difficulty is that given an automaton for \mathbf{x} that maps n to $\mathbf{x}[n]$, it is not immediately clear how many states are needed for $(i, n) \rightarrow \mathbf{x}[i + n]$. However, recently in

[15], the following result was proved: given an automaton A for \mathbf{x} that maps n to $\mathbf{x}[n]$, one can construct an automaton for $(i, n) \rightarrow \mathbf{x}[i + n]$ in time quadratic in the number of states of A .

However, when considering $\mathbf{x}[i + c]$ where c is some fixed constant, we could nevertheless get an automaton whose size depends linearly on c . For example, suppose we wish to check in the Thue–Morse sequence T whether the $(i + 1000)$ -th symbol T_{i+1000} is equal to 1. We can use generate an automaton to achieve that using one of the following two Walnut predicates:

```
[Walnut]$ def p1 "T[i+1000]=@1":
computed ~:1 states - 20ms
computed ~:2 states - 4ms
T[(i+1000)]=@1:1252 states - 87ms
Total computation time: 159ms.
```

```
[Walnut]$ def p2 "T[j]=@1 & j=i+1000":
T[j]=@1:2 states - 0ms
j=(i+1000):17 states - 3ms
(T[j]=@1&j=(i+1000)):33 states - 6ms
Total computation time: 12ms.
```

We see that `p2` produces an automaton with only 33 states, which is reasonable since:

```
[Walnut]$ def tmp1 "T[i+t]=@1":
T[(i+t)]=@1:4 states - 14ms
Total computation time: 47ms.
```

```
[Walnut]$ def tmp2 "j=i+1000":
j=(i+1000):17 states - 4ms
Total computation time: 8ms.
```

However, `p1` surprisingly has 1252 states! The reason for this large difference is that when Walnut processes the predicate `T[j]=@1 & j=i+1000`, it does not construct the same automaton as for `T[i+1000]=@1`; instead, it builds a new one that takes two inputs, i and j , rather than just one. Thus, the substitution of certain constants can, in some cases, cause a significant blow-up in the size of the automaton. Ideally, however, we would like the size to depend on the logarithm of the constant instead.

To achieve that and avoid the potential blow-up problem, we use De Morgan's law to rewrite a \forall query to a \exists formula. Furthermore, we avoid computing the DFA for $\exists x \varphi(x)$; instead we compute the automaton for $\varphi(x)$ and use breadth-first search (BFS) to determine whether any string is accepted, and if so, what a shortest such string is.

Thus, the first step is create the automaton $E(i, j, n, t, u, v)$ for the expression

$$(t < n) \wedge (u = i + t) \wedge (v = j + t) \wedge (\mathbf{x}[u] \neq \mathbf{x}[v]).$$

The automata for the first three conjuncts each have a constant number of states, and so do the automata for the intersection of the corresponding languages. The last conjunct can be expressed as an automaton using $O(N^2)$ states, where N is the number of states in the automaton for \mathbf{x} . Constructing the automaton for $E(i, j, n, t, u, v)$ needs to be done only once, at the very beginning of the algorithm.

Now we want to evaluate $\text{EqFac}(i_0, j_0, n_0)$ for some specific (i_0, j_0, n_0) . To do this, we create automata accepting $0^*(i_0)$, $0^*(j_0)$, and $0^*(n_0)$. These individually require only $O(\log \max(i_0, j_0, n_0))$ states. With additional intersections we create an automaton E' accepting only those 6-tuples (i, j, n, t, u, v) for which i matches i_0 , j matches j_0 , and n matches n_0 . We now perform breadth first search in E' , to see if E' accepts anything; that is, if an accepting state is reachable from the start state. This can be done in linear time in the size of E' , which is polynomial in $\log \max(i_0, j_0, n_0)$. If E' accepts anything, then $\text{EqFac}(i_0, j_0, n_0)$ is false; otherwise it is true.

Thus, for the equality of factors problem, we can perform membership queries in polynomial time.

3.1.2 Hypothesis queries

Now we explain how to implement hypothesis queries efficiently. Given an automaton A , we want to check whether $L(A) = L$. The crucial point is that EqFac is *self-verifying*; that is, provided certain logical formulas about A hold, they provide us with a proof by mathematical induction that the automaton A is correct. In the case of EqFac , an easy induction on n shows that a putative automaton A correctly decides EqFac if and only if all of the following assertions hold:

1. A accepts no illegal representations for i, j, n (representations that are not permissible in the given numeration system);
2. the initial state of A transits to itself on input $[0, 0, 0]$;

3. $\forall i, j A[i, j, 0]$;
4. $\forall i, j, n A[i, j, n + 1] \iff (A[i, j, n] \wedge \mathbf{x}[i + n] = \mathbf{x}[j + n])$.

Conditions 3 and 4 form the basis of a proof by induction on n that A is correct. Although it is perhaps not immediately obvious, we can check all of these conditions in polynomial time; in the case of conditions 3 and 4, by reformulating them with De Morgan's laws.

Condition 3 can be restated as

$$\neg \exists i, j, t (t = 0) \wedge (\neg A[i, j, t]).$$

This can be checked by complementing A , intersecting with the automaton accepting $(\Sigma^*, \Sigma^*, 0^*)$, and using BFS to check if there is a path to an accepting state. If there is none, then condition 3 holds; otherwise it fails and a counterexample $(i_0, j_0, 0)$ is given by the label of the path to any accepting state.

For Condition 4, we first rewrite it as

$$\begin{aligned} \forall i, j, n, t, u, v (t = n + 1 \wedge u = i + n \wedge v = j + n) \implies \\ (A[i, j, t] \iff (A[i, j, n] \wedge \mathbf{x}[u] = \mathbf{x}[v])) \end{aligned}$$

and then use de Morgan's law to replace the \forall with the disjunction of the following three statements:

$$\begin{aligned} &\neg \exists i, j, n, t (t = n + 1) \wedge (\neg A[i, j, n]) \wedge A[i, j, t] \\ &\neg \exists i, j, n, t, u, v (t = n + 1) \wedge (u = i + n) \wedge (v = j + n) \wedge (\neg A[i, j, t]) \wedge A[i, j, n] \\ &\quad \wedge \mathbf{x}[u] = \mathbf{x}[v] \\ &\neg \exists i, j, n, t, u, v (t = n + 1) \wedge (u = i + n) \wedge (v = j + n) \wedge \mathbf{x}[u] \neq \mathbf{x}[v] \wedge A[i, j, t]. \end{aligned}$$

In all three cases we can test the conditions by forming intersections of languages created by the direct product of automata, followed by BFS to check if an input is accepted. If no input is accepted, then A is the correct automaton. Otherwise, A accepts some word that provides the needed counterexample (i_0, j_0, n_0) . Actually, we do not know, a priori, whether it is $A[i_0, j_0, n_0]$ or $A[i_0, j_0, n_0 + 1]$ that is wrong, so some additional membership queries may be necessary to determine which value is incorrect.

We can now put everything together to prove Theorem 5.

Proof. As the analysis of Angluin's algorithm shows, if the minimal automaton A has N states, then the total number of membership and hypothesis queries is bounded by

a polynomial in N . Moreover, the construction of our automata for these membership queries ensures that their size is also polynomially bounded. Finally, we showed that each hypothesis about A is evaluated in polynomial time after reformulating the conditions using De Morgan’s laws and using breadth-first search. Using BFS also ensures that the length of the longest counterexample is bounded by a polynomial in N . Hence, the total running time is polynomial in N , and the size of the automaton for the automatic sequence \mathbf{x} . \square

Using this method, the equality of factors predicate for the Thue-Morse sequence was constructed as shown in Figure 3.1. For clarity, The dead state has been removed. Similarly, we can check formulas like

$$\forall t (t < n) \implies \mathbf{x}[i + t] = \mathbf{y}[j + t]$$

for two automatic sequences \mathbf{x} and \mathbf{y} defined over the same numeration system.

In what follows, we omit explicit mention of conditions 1 and 2 above, although we always need them to check that a hypothesized automaton A is correct.

3.2 Equality of reversals of factors

Another self-verifying predicate involves checking equality of a reversal of a factor. Let w^R denote the reversal of the word w so that, for example, $(\mathbf{drawer})^R = \mathbf{reward}$. Another useful predicate, EqRevFac, asserts that $\mathbf{x}[i..i + n - 1] = \mathbf{x}[j..j + n - 1]^R$. As a special case, we can use this predicate to test whether a factor is a palindrome.

We can use exactly the same technique as in the previous section, based on the fact that EqRevFac is self-verifying just the way EqFac is. More precisely, if A is a claimed automaton for EqRevFac, it is correct if and only if

1. $\forall i, j A[i, j, 0]$;
2. $\forall i, j, n A[i, j, n + 1] \iff (A[i + 1, j, n] \wedge \mathbf{x}[i] = \mathbf{x}[j + n])$.

We leave the details to the reader.

is very desirable to be able to find the corresponding automaton efficiently. We can write a first-order logic formula for this as follows:

$$\forall t (t + p < n) \implies \mathbf{x}[i + t] = \mathbf{x}[i + t + p],$$

and use Theorem 1 to find an automaton computing it. However, this may result in a large intermediate automaton, even when the final result is small.

Of course, we can also express $\text{Per}(i, n, p)$ using the EqFac predicate, but it is possible that the automaton for Per is smaller and hence easier to compute directly by our method.

Suppose A is an automaton that is claimed to compute the predicate $\text{Per}(i, n, p)$ that asserts that p is a period of $\mathbf{x}[i..i + n - 1]$. Then A is correct if and only if the following conditions hold.

1. $\forall i, p A[i, 0, p]$;
2. $\forall i, n, p A[i, n + 1, p] \iff ((p \geq n + 1) \vee (p \leq n \wedge A[i, n, p] \wedge \mathbf{x}[i + n] = \mathbf{x}[i + n - p]))$.

These can be translated into \exists predicates just as we did for the case of EqFac . Notice that the subtraction can be handled by introducing a new variable t and imposing the condition $t + p = i + n$.

3.4 Creating an adder for a numeration system

Using `Walnut` requires choice of a numeration system. For some numeration systems, such as base k for $k \geq 2$, the numeration system is built in.

One of the crucial requirements for `Walnut`'s algorithm to succeed is that the numeration system be *addable*; that is, there is a finite automaton recognizing the addition relation $x + y = z$.

For more exotic numeration systems, such as those based on a Pisot number [20, 21, 8], constructing the adder can be a bit of an onerous task. However, Angluin's algorithm allows it to be constructed "from the ground up" when it exists, because the adder itself is self-verifying. This was already pointed out in [23, Remark 2.1], and we simply reprise this below.

The first step is that we need an incremter; that is, an automaton `Incr` that accepts a pair of inputs (n, x) in parallel if and only if $x = n + 1$. This can be generated by

combining the automaton recognizing the legal representations in the numeration system with an automaton that compares two inputs lexicographically, as discussed in [36, p. 37].

Membership queries are more or less trivial and do not require `Walnut`.

Now suppose we have an automaton A that we hypothesize computes the addition relation. We claim A is correct if and only if both of the following conditions hold (in addition to the conditions on the validity of the representations and leading zeros).

1. $\forall y, z A[0, y, z] \iff y = z$;
2. $\forall x, y, z, t, u (\text{Incr}(x, t) \wedge \text{Incr}(z, u)) \implies (A[x, y, z] \iff A[t, y, u])$.

These two conditions provide the basis and the induction step, respectively, for an induction proof on x that A is correct.

These two conditions can be rearranged, as we did above, to consist of existential claims that can be verified efficiently with BFS. If they fail, short counterexamples can be computed with BFS, too.

For other recent discussions of “automatically” obtaining an adder for a numeration system, see [5, §4.1] and [34].

3.5 Summation of synchronized sequences

We say a sequence (or function) $(b(n))_{n \geq 0}$ from \mathbb{N} to \mathbb{N} is *synchronized* if there exists an automaton B that accepts the representation of those pairs $(n, x) \in \mathbb{N} \times \mathbb{N}$ for which $x = b(n)$. In this case we call B synchronized also. See [38] for more information about these sequences. Notice that it is possible that n is represented in one numeration system, while x is represented in some other numeration system. An example of this is the rarefied Thue-Morse sum automaton from [41], which accepts inputs (n, x) if and only if $x = \sum_{0 \leq i < n} (-1)^{t(3i)}$ where $t(i)$ is the i -th bit of the Thue-Morse sequence.. For this automaton n is represented in base 4 while x represented in base 3,

Often we would like to do partial summation on such a sequence, defining $c(n) = \sum_{0 \leq i < n} b(i)$. Unfortunately there are examples where a is synchronized but the partial sum sequence b is not. As an example, consider the synchronized sequence $\mathbf{p}_2 = 11010001 \dots$, which is 1 if $n + 1$ is a power of 2 and 0 otherwise. Then the partial sum sequence of \mathbf{p}_2 is given by $\lfloor \log_2 n \rfloor + 1$ for $n \geq 1$. However, this kind of growth rate is impossible for a synchronized sequence [38].

Nevertheless, in some cases the partial sum sequence is synchronized, and if it is, then we would like to construct the automaton for it. Luckily, the assertion that c is the partial sum sequence for b is self-verifying. In addition to the usual checks involving legal representations and leading zeros, in order to verify that an automaton C for c is correct, we only need to check that C satisfies the following:

$$(a) \quad \forall x \ C[0, x] \iff x = 0;$$

$$(b) \quad \forall n, t, u, y, z \ (u = n + 1 \wedge z = t + y \wedge B[n, t]) \implies (C[n, y] \iff C[u, z]).$$

Condition (a) provides the basis, and condition (b) the induction step for a proof by induction on n that $C[n, x]$ holds if and only if $c(n) = x$. These two conditions can be turned into existential claims that can be verified with BFS, as before.

There is also the issue of how to compute membership queries. Here we are presented with n and z and we want to determine whether $c(n) = z$. This can be done as follows: from the synchronized automaton B for b , first determine a linear representation for $b(n)$. Recall that a linear representation consists of a row vector v , a column vector w , and a matrix-valued morphism γ such that $b(n) = v \cdot \gamma(y) \cdot w$ for all representations y of n (including those with leading zeros). This linear representation can be trivially computed in linear time in the size of B , as explained in [39, §9.8].

Once we have the linear representation for b , we can compute a linear representation for c using a fairly simple transformation that only increases the size of the linear representation by a constant factor [42]. And from the linear representation we can compute $c(n)$ and check to see whether it equals z , in polynomial time in $\log n$.

Thus once again we can use Angluin’s algorithm to compute the automaton C for the partial sum sequence c , if it exists. Unfortunately, we have no general method to know *a priori* whether C exists. If it does not, Angluin’s algorithm will run forever. If it does halt, however, we are guaranteed that the computed automaton C is correct and that the running time is bounded by a polynomial in the size of B and the size of a minimal automaton for C . Using this method, the partial sum predicates for the Thue-Morse sequence and the Fibonacci word were constructed, and dead states were removed as can be seen in Figures 3.2 and 3.3 respectively. For clarification, the inputs to the automaton in Figure 3.2 are integers n and x , represented in base 2, and the automaton accepts if and only if $x = \sum_{0 \leq i < n} t_i$, where t_i denotes the i -th term of the Thue-Morse sequence.

In some cases `Walnut` can handle those sequences b that take negative integer values. For example, this is true for k -automatic sequences, which we can accept using an automaton

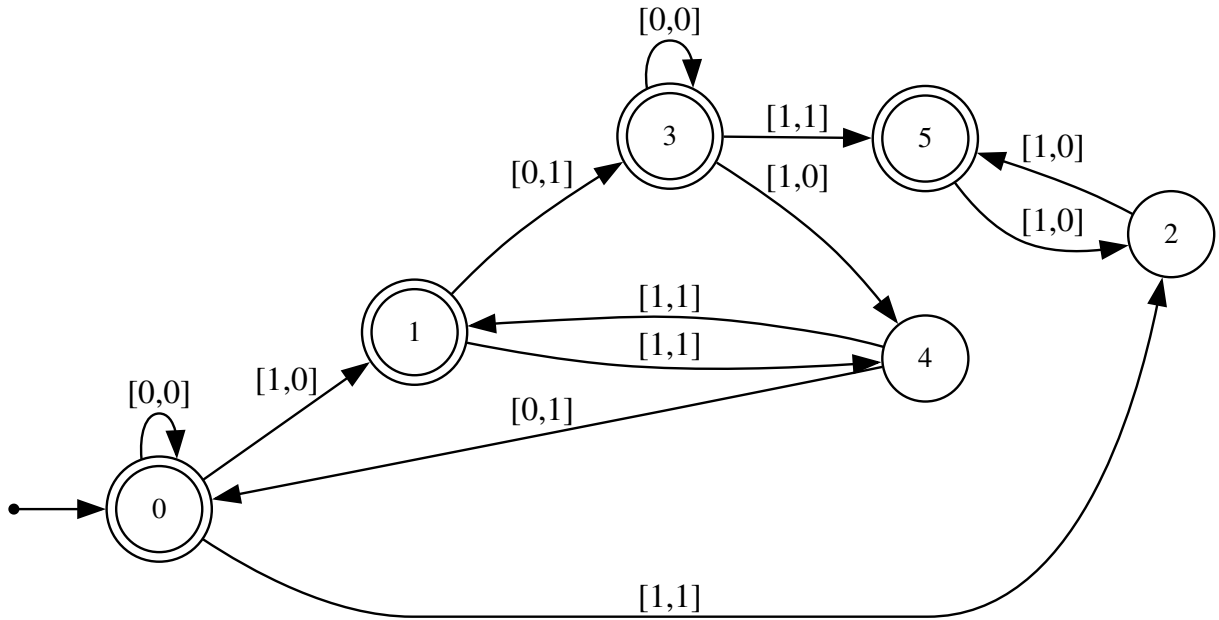


Figure 3.2: Partial sum automaton for the Thue-Morse sequence

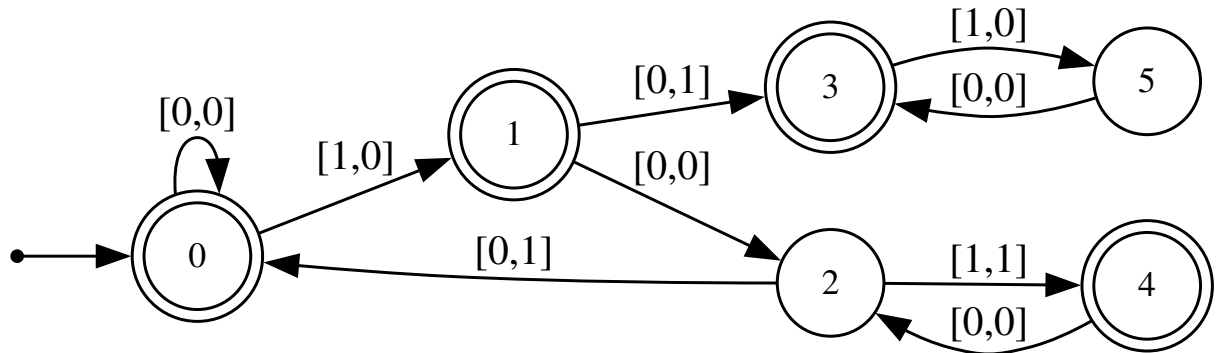


Figure 3.3: Partial sum automaton for the Fibonacci word

defined over base $-k$, and for Fibonacci-automatic sequences, which we can accept using an automaton in the negaFibonacci system. In some cases, then we can compute automata for partial sums $\sum_{0 \leq i < n} b(i)$ for *integer* sequences, not just sequences of natural numbers.

For more details about negative numbers in `Walnut`, see [26].

One of the most useful examples of this technique is the symbol-counting predicate $\text{Count}_a(n, x)$ which is true if and only if $x = |\mathbf{x}[0..n - 1]|_a$, the number of occurrences of the symbol a in the length- n prefix of \mathbf{x} . If Angluin's algorithm produces an automaton for $\text{Count}_a(n, x)$, then we can also compute $|\mathbf{x}[i..i + j - 1]|_a$ via subtraction. This is essential in understanding the abelian properties of an automatic sequence, such as the presence or absence of abelian powers. (Recall that a factor x is an abelian k 'th power if $x = x_1x_2 \cdots x_k$, with each x_i a permutation of x_1 .)

We implemented Angluin's algorithm to successfully deduce the automaton for the partial sums of the Thue-Morse word, Fibonacci word, and Tribonacci word as well as the rarefied Thue-Morse sum automaton from [41]. see Table 4.2 for a summary of the computation.

Chapter 4

Implementation details and results

4.1 Membership queries

In practice (as opposed to theory), there are various algorithmic engineering techniques to speed up the approach we have outlined in this thesis.

For example, for the equality of factors predicate, membership queries involve testing whether $\mathbf{x}[i..i+n-1] = \mathbf{x}[j..j+n-1]$ for some specific triple $(i, j, n) = (i_0, j_0, n_0)$. There are three ways to evaluate this, as follows:

First, we can evaluate this predicate directly from the automaton for \mathbf{x} . In general, this uses $\Theta(n)$ operations on integers of magnitude $\Theta(\max(i, j, n))$, which is clearly not polynomial time in $\log n$. Nevertheless, in practice this is likely to be extremely fast, so if n_0 is small (say $n_0 < 10^7$) then we can evaluate the query more efficiently in this way.

Second, we can evaluate this predicate by substituting the specific values $(i, j, n) = (i_0, j_0, n_0)$ in the predicate and evaluating the resulting predicate in **Walnut**. This could end up taking even more time, asymptotically, since it involves creating intermediate automata computing the functions $t \rightarrow \mathbf{x}[i_0 + t]$ and $t \rightarrow \mathbf{x}[j_0 + t]$, which, when combined by $\mathbf{x}[i_0 + t] = \mathbf{x}[j_0 + t]$, could theoretically result in an automaton of size $\Omega(\max(i_0, j_0)^2)$.

There is also another practical limitation here: in its current implementation, **Walnut** cannot process queries involving integers larger than $2^{31} - 1$. So it seems that this approach is unlikely to be competitive in practice with the first approach, except perhaps if i_0, j_0 are small (say $< 10^3$) and n_0 is large (say $10^7 < n_0 < 2^{31}$).

The third approach, previously outlined in Section 3.1.2, uses **Walnut** to build the “equality of factors” automaton without quantifiers once, and then intersects it with small

automata that accept only the specific values of the tuples we wish to test. This can be done in polynomial time.

Similarly, computing partial sums $\sum_{0 \leq i < n} b(i)$ of a synchronized sequence $(b(i))$ can be done either directly in $O(n)$ time, or via linear representations with $\log n$ multiplications of a vector times an $r \times r$ matrix, where r is the rank of the linear representation. This costs $\Theta(r^2 \log n)$. For small n the direct method will beat the linear representation, so some software engineering is needed to make sure membership queries are as efficient as possible.

Another way to compute the linear representation for the partial summation is as follows. If $b(n, x)$ is the synchronized automaton for b , then the `Walnut` command

```
def sumb "Ex $b(n,x) & i<n & j<x":
```

will directly compute the linear representation (as a `Maple` file) for $c(n) = \sum_{0 \leq i < n} b(i)$. Unlike the method of the previous paragraph, because of the presence of the \exists quantifier, we cannot rule out the possibility that this could result in exponential blowup (in the size of the automaton for b).

4.2 Optimizing performance

A primary bottleneck in our implementation of Angluin’s algorithm is the consistency test. Aside from the fact that every table entry needs to be queried once to be filled, the check for consistency requires querying strings that are not even in the table. In the worst case, to check for consistency we must perform $\Theta(|S| \cdot \Sigma \cdot |E|)$ membership queries. However, many membership queries could actually be looking at the same string but split at different points into S and E . Thus, we cached query results into a hash table. This simple optimization technique reduced the total runtime of the algorithm on the equality of factors predicate for the Thue-Morse sequence from 1672.2 s to 124.6 s. Other runtime statistics, after using a hash table, are shown for different words in Table 4.1. The execution times in the table are measured in seconds.

Table 4.1: Performance of the EqFac predicate on four automatic sequences.

Metric	EqFac			
	Thue-Morse	Baum-Sweet	Fibonacci	Tribonacci
Substitution method time	89.8	4402	323.1	2046.4
Intersection method time	439.5	20230.3	296.1	2687.1
# of unique queries	1672	75243	1032	4816
# of incorrect hypotheses	7	43	6	11
Longest counterexample	4	8	3	7
Longest queried string	8	15	6	11
Final # of states	15	130	12	27
Final $ S $	26	210	16	40
Final $ E $	9	51	9	17

Table 4.2: Performance of the partial and rarefied sum predicates on different automatic sequences.

Metric	Partial sums			Rarefied sums [41]
	Thue-Morse	Fibonacci	Tribonacci	Thue-Morse
Total time (in seconds)	1.4	32.2	3243.4	6156.2
# of unique queries	132	146	12932	3548
# of incorrect hypotheses	3	3	23	9
Longest counterexample	3	4	11	4
Longest queried string	6	7	18	9
Final # of states	7	7	89	17
Final $ S $	8	11	133	29
Final $ E $	5	4	32	11

4.3 Walnut considerations

Several engineering details related to Walnut, and outside the core algorithm, can affect performance. First, reusing a single `Walnut` process throughout the run avoids repeated startup overhead. Second, we found `Walnut`'s built-in BFS in the `test` command to be inefficient. So, instead we used Python implemented a simple BFS for finding shortest counterexamples. Also, as mentioned in Section 3.1.2, to ensure each membership test remains polynomial-time, we rewrote key predicates in order to avoid `Walnut` running into exponential behavior.

We also faced a problem when trying to reject invalid representations. For example, we need to use `?msd_fib` to let `Walnut` know that we are currently using the Fibonacci numeration system. However, in the case of the Fibonacci word if we write all of the counterexample predicates using `?msd_fib`, we won't actually be able to reject the invalid representations. This is due to the fact that `Walnut` automatically does not accept these

invalid representations, so they will never be suggested as counterexamples as they are not part of the test. To work around that, we added one predicate at the beginning for rejecting invalid representations and used `?msd_2` instead. This technique is needed no matter what numeration system we are dealing with. The reason is that `Walnut` is able to consider all binary sequences when using `?msd_2`, so we are able to isolate the invalid ones in this separate predicate.

Finally, a crucial consideration is the ordering of the counterexample predicates. Since the hypothesis tests are essentially doing an induction, we must ensure that the already accepted strings are in fact correct. Otherwise, we will be enforcing the algorithm to accept even more incorrect strings instead of rejecting them. For instance, the last hypothesis query mentioned in [3.1.2](#), was not written as a single test when programmed. Instead, each conjunction was used as a separate test and their orders were reversed. By using the third conjunction first, we ensure that we reject incorrect strings before building upon them as in the second conjunction.

Chapter 5

Binary 3-uniform morphisms

5.1 Motivation

In combinatorics on words, the properties of critical exponent and subword complexity play important roles in understanding the structure of words and sequences. These properties provide insights into the periodicity and repetition within words, The *critical exponent* measures the maximum ratio of a factor's length to its smallest period, with higher values indicating greater local periodicity, and has applications in areas such as string matching algorithms [32]. *Subword complexity* counts the number of distinct blocks of a given length within a word, and has different applications even in branches of cryptography [11]. Binary k -uniform morphisms, which map each letter of a binary alphabet to a fixed-length word of k symbols, provide a simple yet useful framework for studying both of these properties.

Our methods for determining the critical exponent and subword complexity of fixed points of binary 3-uniform morphisms rely heavily on the equality of factors and periods of factors predicates, both of which can be constructed in polynomial time by utilizing Angluin's algorithm as shown in Sections 3.1 and 3.3 respectively. Building on these foundations, we now turn to a detailed investigation of the critical exponent and subword complexity for binary 3-uniform morphisms for the rest of the thesis. However, before going into the technical details, we need to get some definitions out of the way.

5.2 Formal definitions

We start with some useful preliminary definitions.

Definition 1 (Integer Power, Rational Power). For a positive integer p and a non-empty word x , we define the p -th power of x , written as $x^p = xxx \cdots$, to be x concatenated with itself p times. Moreover, if q divides $|x|$, then the p/q -th power of x is the prefix of length $(p/q)|x|$ of the word $x^\omega = xxx \cdots$. For example, $(\mathbf{ent})^{7/3} = \mathbf{entente}$.

Definition 2 (Period). A nonempty finite word w has a period $p > 0$ if there is a prefix x of w such that $|x| = p$ and $w = x^\alpha$ where $1 \leq \alpha \in \mathbb{Q}^+$.

Definition 3 (Exponent). For a finite or infinite word w with p as its smallest period, the exponent of w is $\exp(w) = |w|/p$. Thus, for example, the French word $\mathbf{tentent}$ has periods 3, 6, 7 and exponent $7/3$.

Of course, if the exponent is irrational, it is not attained by any finite factor of w .

Definition 4 (Critical Exponent). The critical exponent of a word \mathbf{w} , written as $\text{ce}(\mathbf{w})$, is the supremum of $\exp(x)$ over all finite non-empty factors x of \mathbf{w} .

Informally, it determines an upper bound on the maximum repetition possible within a word, and as we will see in the upcoming sections, this bound is not necessarily attained in all infinite words.

For example, the word $\mathbf{lophophore}$ contains the string $\mathbf{ophopho}$ which has length 7, period 3 and exponent $7/3$. Thus, the critical exponent of $\mathbf{lophophore}$ is $7/3$.

Definition 5 (Subword Complexity). For a word w , we let $\rho_w(n)$ be the number of distinct factors of length n in w .

Definition 6 (Morphism). A morphism h is a mapping that satisfies $h(xy) = h(x)h(y)$ for all words x and y . If the alphabet consists of exactly two letters, then it is a binary morphism. A morphism is said to be k -uniform if it maps each letter of the alphabet to a word of length k .

Definition 7 (Fixed Point of a Morphism). An infinite word \mathbf{w} is called the fixed point of a morphism h if $h(\mathbf{w}) = \mathbf{w}$.

5.3 Studied morphisms

We note that there are 2^6 possible binary 3-uniform morphisms. There is a clear symmetry among the 64 morphisms because the roles of 0 and 1 can be swapped. Thus, to ensure

a fixed point starting with 0 for example, the image of 0 must itself start with 0. This reduces the number of distinct morphisms to 32 considering letter renaming. Moreover, all morphisms in which both the images of 0 and 1 map to the same string x result in the periodic word x^ω . In other words, these $2^2 = 4$ cases are considered trivial.

Furthermore, if in a morphism h the image of 0 contains no 1, in other words 0 maps to 000, then $h^\omega(0) = 0^\omega$. This gives us another 7 trivial cases that can be excluded. Finally, we claim that the fixed points of the morphisms $0 \mapsto 010, 1 \mapsto 101$ and $0 \mapsto 011, 1 \mapsto 111$ are also ultimately periodic. The former has a fixed point of $(01)^\omega$ while the latter has a fixed point of $0(1)^\omega$. Consequently, we are left with the 19 morphisms listed in Table 5.1.

Word	Morphism	OEIS	Critical Exp. [28]	Complexity $\rho(n)$	For $n \geq$
3-adic	$0 \rightarrow 001, 1 \rightarrow 000$	A182581	6	$\rho_{3\text{-adic}}(n)^*$	3
MK1	$0 \rightarrow 001, 1 \rightarrow 010$	A189628	$7/2 = 3.5$	$2n - 1^*$	2
Stewart choral [17]	$0 \rightarrow 001, 1 \rightarrow 011$	A116178	3	$2n$	1
MK2	$0 \rightarrow 001, 1 \rightarrow 100$	A189632	4^\dagger	$3n - 3^*$	3
Ferenczi	$0 \rightarrow 001, 1 \rightarrow 101$	A189640	3 [17]	$2n$	1
Mephisto–Waltz (MW)	$0 \rightarrow 001, 1 \rightarrow 110$	A064990	3^\dagger	$4n - 4^*$	2
Rote	$0 \rightarrow 001, 1 \rightarrow 111$	A189820	∞	$2n$	1
3-adicV2	$0 \rightarrow 010, 1 \rightarrow 000$	A356982	6	$\rho_{3\text{-adicV2}}(n)^*$	3
MK3	$0 \rightarrow 010, 1 \rightarrow 001$	A189664	$7/2$	$2n - 1^*$	2
$und_3 - 1$ [17]	$0 \rightarrow 010, 1 \rightarrow 011$	A080846	3	$2n$	1
MK4	$0 \rightarrow 010, 1 \rightarrow 100$	A189668	$7/2$	$2n - 1^*$	2
Noche [17]	$0 \rightarrow 010, 1 \rightarrow 110$	A189673	3	$2n$	1
Sierpiński	$0 \rightarrow 010, 1 \rightarrow 111$	A316829	∞	$2n - 1^*$	2
MK5	$0 \rightarrow 011, 1 \rightarrow 000$	A189816	$15/2 = 7.5$	$\rho_{\text{MK5}}(n)^*$	5
MK6 [17]	$0 \rightarrow 011, 1 \rightarrow 001$	A189706	3	$2n$	1
Sierpiński gasket [17]	$0 \rightarrow 011, 1 \rightarrow 010$	A156595	3	$2n$	1
MK7	$0 \rightarrow 011, 1 \rightarrow 100$	A189718	3^\dagger	$4n - 4^*$	2
MK8	$0 \rightarrow 011, 1 \rightarrow 101$	A189723	$7/2$	$2n - 1^*$	2
MK9	$0 \rightarrow 011, 1 \rightarrow 110$	A189727	4^\dagger	$3n - 3^*$	3

Table 5.1: Table of aperiodic fixed points of binary 3-uniform morphisms. A dagger signifies that the critical exponent is attained for this word. An asterisk means the entry is a new result of this thesis.

In our thesis, we will focus only on the 12 highlighted morphisms in Table 5.1, identified by asterisks. However, it is worth mentioning that the critical exponents of the fixed points of these morphisms have already been computed before [28, Table 5.1]. Moreover, the morphisms with critical exponent 3 and factor complexity $2n$ have been previously characterized in [17]. Finally, the last column of Table 5.1 indicates the value n , such that

for all n greater than or equal to this value, the subword complexity formula holds. In Section 7, we will describe the procedure used to compute the following subword complexities:

$$\rho_{3\text{-adic}}(n) = \rho_{3\text{-adicV2}}(n) = \begin{cases} 2n - 2 \cdot 3^r, & \text{if } 3 \cdot 3^r \leq n < 5 \cdot 3^r; \\ n + 3 \cdot 3^r, & \text{if } 5 \cdot 3^r \leq n < 3 \cdot 3^{r+1}; \end{cases}$$

$$\rho_{\text{MK5}}(n) = \begin{cases} 3n - \frac{1}{2}(7 \cdot 3^r + 3), & \text{if } \frac{3}{2} \cdot 3^{r+1} + \frac{1}{2} \leq n < \frac{13}{6} \cdot 3^{r+1} + \frac{1}{2}; \\ 2n + 3^{r+1} - 1, & \text{if } \frac{13}{6} \cdot 3^{r+1} + \frac{1}{2} \leq n < \frac{3}{2} \cdot 3^{r+1} + \frac{1}{2}. \end{cases}$$

Chapter 6

Critical exponent

The fixed points of uniform morphisms always have infinite or rational critical exponents [37]. Since every rational number can be represented by a continued fraction, we can search for this continued fraction instead. However, we first need to check whether the word has an infinite exponent or not.

6.1 Checking for infinite critical exponent

We can check if an infinite word \mathbf{x} has an infinite critical exponent as follows:

First, create an automaton A using Walnut that accepts the following language L :

$$L = \{(n, p) : \text{There is a factor of } \mathbf{x} \text{ with length } n \text{ and period } p, \\ \text{and } p \text{ is the least period over all factors of length } n \text{ of } \mathbf{x}\}.$$

Theorem 6. *Then the critical exponent is infinite iff the resulting automaton has N states, and there is a pair $y = (n, p)$ such that $\frac{n}{p} > k^N$.*

Proof. The forward direction of the claim is straightforward. For the reverse direction, we use two key facts. The first is that since $n > k^N p$, we know that $(p)_k$ starts with at least N “0”s and n starts with a nonzero digit. The second fact is that we can apply the pumping lemma on x because $|y| = |(n, p)_k| \geq k^N \geq N$ which is the number of states of the automaton A . Hence, by writing $y = uvw$ with $|uv| \leq N$ and $|v| \geq 1$, we have $wv^i w \in L$ for all $i \geq 0$. In other words, we can generate longer words accepted by A with

factors having the same period p , but with arbitrarily large length. In other words, we can generate words in $uv^i w \in L$ with arbitrarily large exponents $\exp(uv^i w) \rightarrow \infty$. Thus, the critical exponent of the word x is infinite. See [37, Theorem 8] and [39, §10.6.5]. \square

The following are the Walnut predicates implemented to achieve this on two different examples:

```
morphism g "0->001 1->010":
promote MK1 g:
# We define the required morphism as MK1 in Walnut
def haspmk1 "?msd_3 p>0 & p<=n & Aj (j>=i & j+p<i+n) => MK1[j]=MK1[j+p]":
# Does MK1[i..i+n-1] have period p?
# 28 states
def leastpmk1 "?msd_3 $haspmk1(i,n,p) & (Aq (q>=1 & q<p) =>
    ~$haspmk1(i,n,q))":
# Is the period p of MK1[i..i+n-1] minimal over all factors of length n?
# 42 states
def npmk1 "?msd_3 Ei $leastpmk1(i,n,p)":
# For a starting index i, does n and p belong to the language?
# 53 states
reg three53 msd_3 msd_3 "([0,0])*([1,0] | [2,0])([0,0] | [1,0] | [2,0]) ...
    ... ([0,0] | [1,0] | [2,0])([0,0] | [1,0] | [2,0]) |
    [0,1] | [1,1] | [2,1] | [0,2] | [1,2] | [2,2])*":
# Takes two inputs x and y and asks is x>=y*3^53+c? Where c>1
# "([0,0] | [1,0] | [2,0])" is written 53 times but truncated for visibility.
# 56 states
eval infmk1 "?msd_3 Ei,n,p ($leastpmk1(i,n,p) & $three53(n,p))":
# Is there a pair (n, p) with properties defined above (in L)
# such that n/p>3^53=k^N?
# i.e: Does MK1 have an infinite critical exponent? FALSE
```

So MK1 does not have an infinite critical exponent. Since we know that the morphism $0 \rightarrow 011, 1 \rightarrow 111$ has an infinite critical exponent, we can use the same approach for it as a sanity check. As discussed in 5, this morphism is ultimately periodic with a fixed point of $0(1)^\omega$.

```
morphism g "0->011 1->111":
```

```

promote MK g:
# We define the required morphism as MK in Walnut
def haspmk "?msd_3 p>0 & p<=n & Aj (j>=i & j+p<i+n) => MK[j]=MK[j+p]":
# 8 states
def leastpmk "?msd_3 $haspmk(i,n,p) & (Aq (q>=1 & q<p) =>
~$haspmk(i,n,q))":
# 6 states
def npmk "?msd_3 Ei $leastpmk(i,n,p)":
# For a starting index i, does n and p belong to the language?
# 4 states
reg three4 msd_3 msd_3 "([0,0])*([1,0] | [2,0])([0,0] | [1,0] | [2,0])
([0,0] | [1,0] | [2,0])([0,0] | [1,0] | [2,0])
([0,0] | [1,0] | [2,0])([0,0] | [1,0] | [2,0] |
[0,1] | [1,1] | [2,1] | [0,2] | [1,2] | [2,2])*":
# Takes two inputs x and y and asks is x>=y*3^5?
# 6 states
eval infmk "?msd_3 Ei,n,p ($leastpmk(i,n,p) & $three4(n,p))":
# Is there a pair (n, p) with properties defined above (in L)
such that n/p>=3^5>3^4=k^N?
# returns TRUE

```

6.2 Continued fraction search

A continued fraction [24, §10.1] can be represented as a list using abbreviated notation as follows:

$$\frac{415}{93} = 4 + \frac{1}{2 + \frac{1}{6 + \frac{1}{7}}} = [4, 2, 6, 7].$$

In this representation, increasing even-indexed partial quotients results in a larger value for the fraction, while increasing odd-indexed partial quotients has the opposite effect. For instance:

$$\frac{7}{2} = [3, 2] < [4, 2] = \frac{9}{2} > [4, 3] = \frac{13}{3}.$$

Our main goal then is to calculate the continued fraction of the critical exponent by iteratively calculating each partial quotient in the abbreviated notation [39, §10.6.5]. We begin by using exponential search to calculate the first partial quotient. We iteratively check, for each integer $i \geq 1$, whether all factors of our infinite word \mathbf{z} have exponent $\leq 2^i$,

and we stop as soon as the answer is TRUE. At this point, we know that \mathbf{z} has a critical exponent α where $2^{i-1} < \alpha \leq 2^i$.

In the next step, we use binary search in the interval $(2^{i-1}, 2^i]$ to determine the smallest integer t_1 such that all factors of \mathbf{z} have an exponent less than or equal to $[t_1] = t_1$. If there is a factor with exponent $[t_1]$ then $\alpha = t_1$ and we are done. Otherwise, we know that the first partial quotient in the continued fraction expansion of the rational number α is $t_1 - 1$, and $[t_1 - 1] < \alpha < [t_1]$.

The algorithm now proceeds by determining each successive partial quotient t_i in the same way; We use exponential search to determine an interval for the next partial quotient, and then use binary search to calculate t_i exactly. One caveat is that odd-indexed partial quotients have reversed intervals due to the nature of continued fractions. For instance, $[t_1 - 1, t_2] < \alpha < [t_1 - 1, t_2 - 1]$.

Finally, let $\alpha = \frac{a}{b}$ and let N denote the number of digits in $\max(a, b)$. Then the number of partial quotients in the continued fraction expansion of α is at most $\text{poly}(N)$, meaning the algorithm terminates in a finite number of iterations. Moreover, the value of each partial quotient is bounded by N [6, §4.5], and each iteration performs an exponential search requiring $\mathcal{O}(\log N)$ steps, where each step corresponds to evaluating a Walnut predicate. Thus, the algorithm takes at most $\text{poly}(N)$ steps in total.

We now illustrate this procedure on the specific example of MK1. The following are the Walnut predicates implemented to achieve the described procedure:

```
eval checkmk1 "?msd_3 Ai,n,p (n>=1 & $haspkm1(i,n,p)) => (p>=n)":
# Do all factors of MK1 have an exponent of 1 or less? FALSE
eval checkmk1 "?msd_3 Ai,n,p (n>=1 & $haspkm1(i,n,p)) => (7*p>=2*n)":
# Do all factors of MK1 have an exponent of 2 or less? FALSE
eval checkmk1 "?msd_3 Ai,n,p (n>=1 & $haspkm1(i,n,p)) => (4*p>=n)":
# Do all factors of MK1 have an exponent of 4 or less? TRUE
eval checkmk1 "?msd_3 Ei,n,p (n>0 & $haspkm1(i,n,p) & (4*p=n))":
# Is an exponent of 4 attained and thus 4 is the critical exponent? FALSE
eval checkmk1 "?msd_3 (Ai,n,p ($haspkm1(i,n,p) => n<4*p)) &
(Ec Am Ei,n,p ((c>0 & p>m & $haspkm1(i,n,p)) &
(n+c>=4*p)))":
# Is 4 the critical exponent of MK1 and not attained? FALSE
eval checkmk1 "?msd_3 Ai,n,p (n>=1 & $haspkm1(i,n,p)) => (3*p>=n)":
# Do all factors of MK1 have an exponent of 3 or less? FALSE
eval checkmk1 "?msd_3 Ai,n,p (n>=1 & $haspkm1(i,n,p)) => (7*p>=2*n)":
```

```

# Do all factors of MK1 have an exponent of 3.5 or less? TRUE
eval checkmk1 "?msd_3 Ei,n,p (n>0 & $haspmk1(i,n,p) & (7*p=2*n))":
# Is an exponent of 3.5 attained and thus it's the critical exponent FALSE
def checkmk1 "?msd_3 (Ai,n,p ($haspmk1(i,n,p) => 7*p>2*n)) &
      (Ec Am Ei,n,p ((c>0 & p>m & $haspmk1(i,n,p)) &
      (2*n+2*c>=7*p)))":
# Is 3.5 the critical exponent of MK1 and not attained? TRUE

```

Chapter 7

Subword complexity

We start by noting that [14] proved that the subword complexity function of a k -automatic sequence \mathbf{x} is k -synchronized. In other words, one can construct an automaton that takes (n, y) as input and accepts if and only if $y = \rho_{\mathbf{x}}(n)$. However, this approach has two limitations:

1. The automaton only provides the subword complexity for an input values of n , while we aim for a complete description of $\rho_x(n)$, at every n . This is non-trivial, as the subword complexity can be intricate in some cases, as illustrated for the 3-adic word in Table 5.1 which required a piecewise-linear function.
2. Using Walnut to implement this technique is computationally expensive, making it practical only for very small examples.

Because of these limitations, we explore two alternative methods for computing subword complexity. The first is a heuristic method applicable to any binary k -automatic sequence; it is not guaranteed to work in every case, but is effective in many practical situations. The second is a deterministic algorithm specifically designed for all binary uniform morphisms.

7.1 Heuristic approach

7.1.1 Upper bound using right-special factors

Our heuristic approach first uses right-special factors to establish an upper bound on the subword complexity. Then, we use a linear representation to graph $\rho(n)/n$ so we can

formulate a conjecture about $\rho(n)$. This conjecture is later proved or disproved also using linear representations once again or via Walnut predicates. The following sections discuss these steps in more detail, but we first start by defining right-special factors:

Definition 8 (Right-Special Factor). *A factor w of x is right-special if there exist two distinct letters a and b such that both wa and wb are factors of x as well. For example, the word `cameraman` contains the right-special factor `am`, which can be followed by `e` or `a`.*

Let $\text{RSF}_h(n)$ be the number of right-special factors of length n in the fixed point of morphism h . Moreover, let

$$\text{MRSF}_h(n) = \max_{n \in \mathbb{Z}^+} \{m \in \mathbb{N} : \text{RSF}_h(n) = m\}.$$

We now get an upper bound on the subword complexity at n of this infinite word by computing the maximum number of right-special factors of length n . From Definition 8 of a right-special factor, we can see that any non special-right factor of length n is counted once in factors of length $n + 1$. On the other hand, right-special factors are counted twice. Thus, if $\rho_{3\text{-adic}}(n)$ is defined as the subword complexity of the fixed point of the “3-adic” morphism at length n , $\rho_{3\text{-adic}}(n+1) - \rho_{3\text{-adic}}(n)$ is exactly the number of right-special factors of length n . Moreover, $\rho_{3\text{-adic}}(n) \leq n \cdot \text{MRSF}_{3\text{-adic}}(n)$ [14].

The following are the Walnut predicates implemented to obtain an upper bound on $\text{MRSF}_{3\text{-adic}}(n)$:

```
morphism g "0->001 1->000":
promote TDC g:
# We define the required morphism as TDC in Walnut
def eqtdc "?msd_3 Am (m<n) => TDC[i+m] = TDC[j+m]":
# Does TDC[i..i+n-1] = TDC[j..j+n-1]
# 7 states
def rspectdc "?msd_3 Ej $eqtdc(i,j,n) & TDC[i+n] != TDC[j+n]":
# Is TDC[i..i+n-1] a right-special factor?
# 4 states
def nftdc "?msd_3 Aj (j<i) => ~$eqtdc(i,j,n)":
# Is TDC[i..i+n-1] a novel factor (never appeared before)?
# 5 states
def nrtdc "?msd_3 $nftdc(i,n) & $rspectdc(i,n)":
# Is TDC[i..i+n-1] both a novel factor and a right-special factor?
# 3 states
```

```

def lesstdc "?msd_3 Em (m<n) & TDC[i+m]<TDC[j+m] & $eqtdc(i,j,m)":
# Is TDC[i..i+n-1] lexicographically less than TDC[j..j+n-1]?
# 17 states
def spec2tdc "?msd_3 En,i1,i2 $lesstdc(i1,i2,n) &
              $nrtdc(i1,n) & $nrtdc(i2,n)":
# Does the TDC have 2 unique right-special factors of length n for any n?
# 1 state
def spec3tdc "?msd_3 En,i1,i2,i3 $lesstdc(i1,i2,n) & $lesstdc(i2,i3,n) &
              $nrtdc(i1,n) & $nrtdc(i2,n) & $nrtdc(i3,n)":
# Does the TDC have 3 unique right-special factors of length n for any n?
# 1 state

```

We note that `spec2tdc` evaluates to `TRUE` while `spec3tdc` evaluates to `FALSE`. Consequently, we know that there are at most 2 special factors of every length, and for all n $\rho_{3\text{-adic}}(n+1) - \rho_{3\text{-adic}}(n) \leq 2$. Thus, $\rho_{3\text{-adic}}(n) \leq 2n$.

7.1.2 Drawing $\rho(n)/n$ using linear representation

We begin by writing a Walnut predicate to count the number of novel factors in TDC.

```
eval tdcsbc n "?msd_3 Aj (j<i) => ~$eqtdc(i,j,n)":
```

By running this predicate, Walnut generates a Maple file “`tdcsbc.mpl`” containing the linear representation in the “`Result`” directory. Adjusting the vector v to get v' by multiplying with $\gamma(0)$ multiple times [39, §9.8], we get the required linear representation for $\rho_{3\text{-adic}}(n)$:

$$v = v' = (1 \ 1 \ 0 \ 0 \ 0), \quad w^T = (1 \ 0 \ 1 \ 0 \ 1)$$

$$\gamma(0) = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \end{pmatrix}, \quad \gamma(1) = \begin{pmatrix} 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 2 & 1 & 0 \end{pmatrix}, \quad \gamma(2) = \begin{pmatrix} 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 2 & 1 & 0 \\ 0 & 0 & 3 & 0 & 0 \end{pmatrix}$$

With $\rho_{3\text{-adic}}(n) = v' \gamma((n)_k) w$, where $(n)_k$ is the base- k representation of n , and in our case $k = 3$. Finally, using this we can draw a graph of $\rho_{3\text{-adic}}(n)/n$ by substituting different values for n .

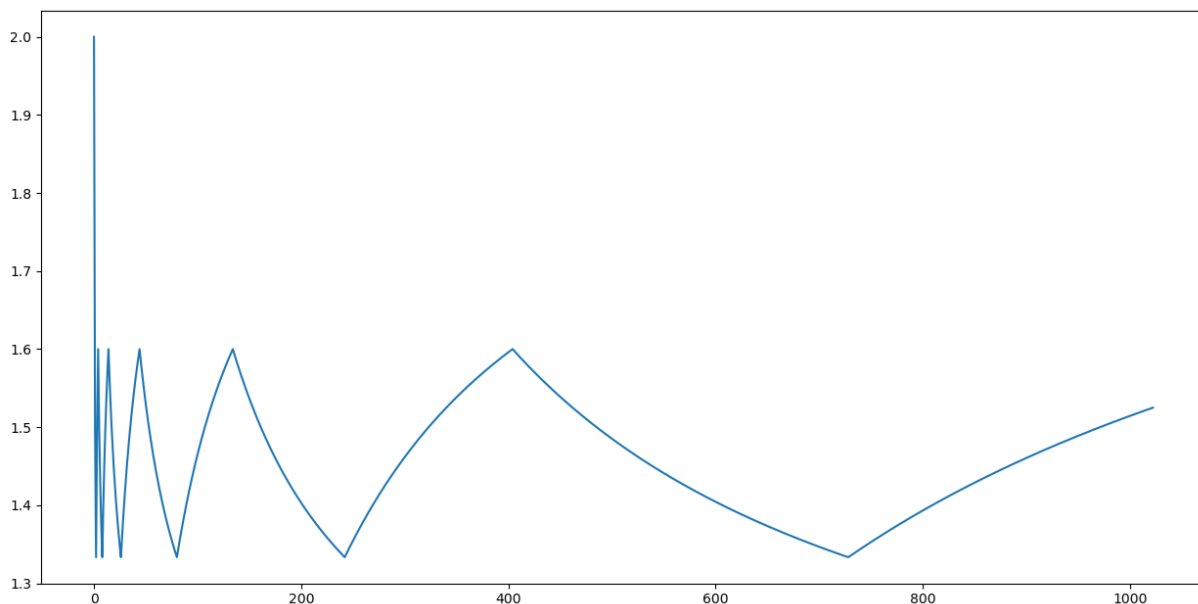


Figure 7.1: Plot of $\rho_{3\text{-adic}}(n)/n$

7.1.3 Deduce a conjecture on the subword complexity

In order to form a conjecture on the complete description of the subword complexity of the infinite word “3-adic”, we need to do two things:

- Get a description of the local maxima and minimum points of the graph in Figure 7.1 including the subword complexity at those lengths.
- Investigate the behavior of the subword complexity function between these points.

Looking at the graph in Figure 7.1, we note that the first few local minimum points occur at the following values of n : 3, 9, 27, 81, 243, 729. Meanwhile, the first few local maximum points occur at the following values of n : 5, 15, 45, 135, 405. So, we can conjecture that the local minima occur at $n = 3^{r+1}$ and local maxima at $n = 5 \cdot 3^r$ for $r \geq 0$. Since

$$\rho(3^{r+1}) = v\gamma(1)\gamma(0)\gamma(0)^r w = v'\gamma(0)^r w, \quad \text{where } v' := v\gamma(1)\gamma(0),$$

we can compute $\rho(3^{r+1})$ for all $r \geq 0$ using the minimal polynomial of $\gamma(0)$, which is $x(x-1)(x-3)$ [39, §9.9]. Thus, it follows that

$$\rho_{3\text{-adic}}(3^{r+1}) = c_0 \cdot 0^r + c_1 \cdot 1^r + c_2 \cdot 3^r = c_2 \cdot 3^r + c_1.$$

Using the linear representation, we compute $\rho(3) = 4$ and $\rho(9) = 12$. Substituting $r = 0$ and $r = 1$ yields the linear system

$$\begin{aligned} \rho_{3\text{-adic}}(3^{0+1}) &= c_2 + c_1 = 4, \\ \rho_{3\text{-adic}}(3^{1+1}) &= 3c_2 + c_1 = 12. \end{aligned}$$

Solving gives $c_2 = 4$ and $c_1 = 0$, and hence $\rho_{3\text{-adic}}(3^{r+1}) = 4 \cdot 3^r$. Similarly, since $\rho(5 \cdot 3^r) = v\gamma(1)\gamma(2)\gamma(0)^r w$, we can use the same argument to show that $\rho_{3\text{-adic}}(5 \cdot 3^r) = 8 \cdot 3^r$.

Finally, by calculating $D_{3\text{-adic}}(n) = \rho_{3\text{-adic}}(n+1) - \rho_{3\text{-adic}}(n)$ at different n , we note that $D_{3\text{-adic}}(n) = 2$ when n is increasing from a local minimum point to a local maximum point in Figure 7.1 while $D_{3\text{-adic}}(n) = 1$ when n is decreasing from a local maximum point to a local minimum point in Figure 7.1. This leads us to suggest the following conjecture:

$$\rho_{3\text{-adic}}(n) = \begin{cases} 4 \cdot 3^r + 2i, & \text{if } n = 3^{r+1} + i, 0 \leq i < 2 \cdot 3^r; \\ 8 \cdot 3^r + i, & \text{if } n = 5 \cdot 3^r + i, 0 \leq i < 4 \cdot 3^r. \end{cases}$$

The condition $n = 3^{r+1} + i, 0 \leq i < 2 \cdot 3^r$ translates to $3 \cdot 3^r \leq n < 5 \cdot 3^r$. Moreover, since $n = 3^{r+1} + i = 3 \cdot 3^r + i$, we have $i = n - 3 \cdot 3^r$, and hence

$$\rho_{3\text{-adic}}(n) = 4 \cdot 3^r + 2(n - 3 \cdot 3^r) = 2n - 2 \cdot 3^r.$$

Doing this for the second case as well and combining the two yields:

$$\rho_{3\text{-adic}}(n) = \begin{cases} 2n - 2 \cdot 3^r, & \text{if } 3 \cdot 3^r \leq n < 5 \cdot 3^r, \\ n + 3 \cdot 3^r, & \text{if } 5 \cdot 3^r \leq n < 3 \cdot 3^{r+1}. \end{cases}$$

7.1.4 Proving the conjecture using Walnut

Since we have proved that there are at most 2 special factors of every length, the novel factors in the “3-adic” word are grouped into at most 3 contiguous blocks [39, §10.8.19]. This can also be seen in Figure 7.2. Consequently, we can use the following Walnut predicates to prove our conjecture:

```

def tdcsub "?msd_3 E a2,a3,b1,b2,b3 (b1<=a2&a2<=b2&b2<=a3&a3<=b3) &
(Ai i<b1 => $nftdc(i,n)) & (Ai (a2<=i&i<b2) => $nftdc(i,n)) &
(Ai (a3<=i&i<b3)=>$nftdc(i,n)) & (Ai (i>=b1&i<a2) => ~$nftdc(i,n)) &
(Ai (i>=b2&i<a3)=> ~$nftdc(i,n)) & (Ai (i>=b3) => ~$nftdc(i,n)) &
(s = b1+(b2-a2)+(b3-a3))":
# Is the subword complexity of TDC equal to s?
# 7 states
reg power3 msd_3 "0*10*":
# Is the input a power of 3?
# 2 states
eval tdcsubcheck "?msd_3 Ax,n,y
                ((($power3(x) & 3*x<=n & n<5*x & $tdcsub(n,y)) =>
                y=2*n-2*x) & (($power3(x) & 5*x<=n & n<9*x &
                $tdcsub(n,y)) => y=n+3*x))":
# Is the subword complexity piecewise-defined as in our conjecture?
# 1 state

```

Finally, Walnut evaluates `tdcsubcheck` to TRUE which proves the conjectured subword complexity formula.

7.1.5 Proving the conjecture using linear representations

For some morphisms, the previous step might result in an exponentially large automaton that cannot be feasibly executed due to computational limitations. In such cases, we must prove our conjecture using an alternative method. For instance, assume that using the methodology described so far we have this conjecture for the subword complexity of the Mephisto–Waltz word $\rho_{\text{MW}}(n) = 4n - 4$ for $n \geq 2$. We use the following technique to prove it:

1. Get the linear representation for LHS and RHS of the conjecture.
2. Get the linear representation for the difference between them.
3. Minimize the difference using Berstel-Reutenauer minimization algorithm for linear representations [7], and check if it is equivalent to the zero representation.

The first step is straightforward. The only caveat is that we only need to check that the subword complexity formula holds for $n \geq 2$ as mentioned in Table 5.1. To achieve this, we can use the following Walnut commands:

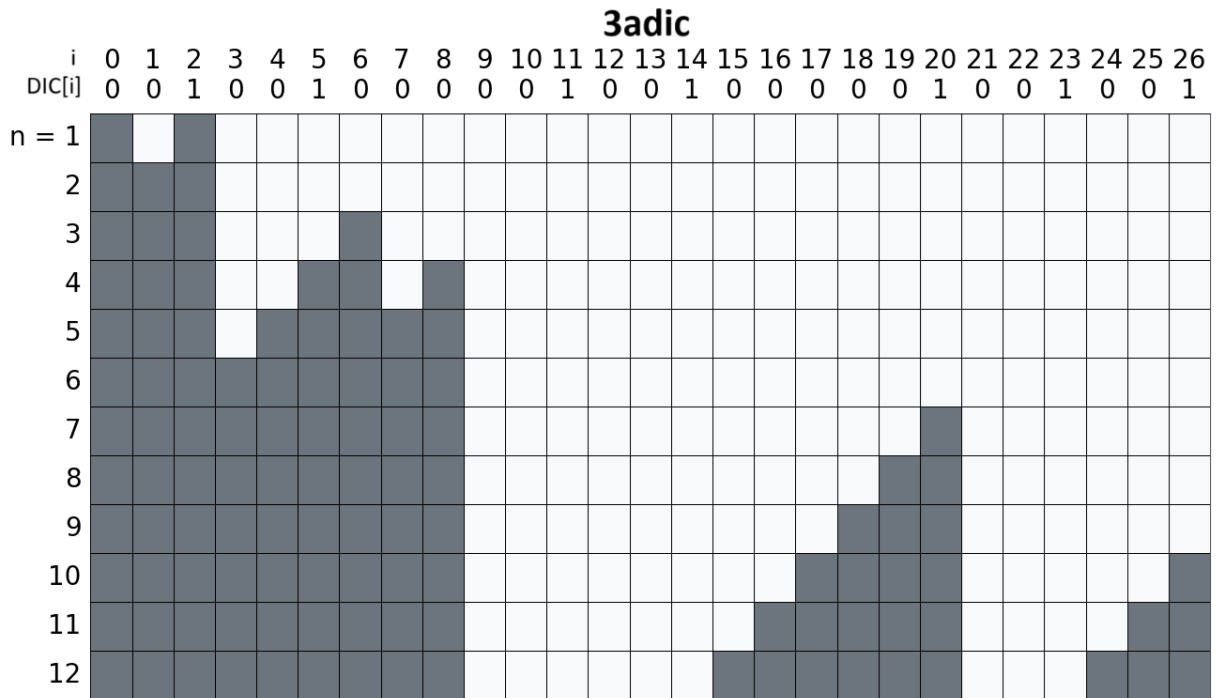


Figure 7.2: First few novel factors of the 3-adic sequence using `tdcsubc`

```
def eqmw "?msd_3 Am (m<n) => MW[i+m] = MW[j+m]":
# Are factors of length n starting at indices i and j equal?
# i.e: MW[i...i+n-1] = MW[j...j+n-1]?
# 13 States
def nfmw "?msd_3 Aj (j<i) => ~$eqmw(i,j,n)":
# Is MW[i...i+n-1] a novel factor?
# 9 States
def lhs n "?msd_3 n>=2 & $nfmw(i, n)":
# 9 States
def rhs n "?msd_3 n>=2 & i<4*n-4":
# 8 States
```

Thus, Walnut outputs `lhs.mpl` and `rhs.mpl` containing the linear representation

$$v_1, \gamma((n)_k)_1, w_1 \text{ and } v_2, \gamma((n)_k)_2, w_2.$$

Now we simply compute the difference $v_1\gamma((n)_k)_1w_1 - v_2\gamma((n)_k)_2w_2$ by defining:

$$v = [v_1 \quad -v_2], \quad w = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}, \quad \gamma((n)_k) = \left[\begin{array}{c|c} \gamma((n)_k)_1 & 0 \\ \hline 0 & \gamma((n)_k)_2 \end{array} \right].$$

The vertical and horizontal lines are added solely for clarity, to visually separate the different submatrices. For the sake of completeness we explicitly state these matrices (and vectors) for the Mephisto-Waltz word. Note that that v was multiplied with $\gamma(0)$ multiple times [39, §9.8], and the lines were added to clarify the different subvectors and submatrices used to construct the final one.

$$v = [1 \ 1 \ 1 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ | \ -1 \ -1 \ -1 \ 0 \ 0 \ 0 \ -1 \ 0],$$

$$w^T = [0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \ | \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 0],$$

$$\gamma((n)_0) = \begin{bmatrix} 1110000000|00000000 \\ 0000000100|00000000 \\ 0000000000|00000000 \\ 0000003000|00000000 \\ 0000102000|00000000 \\ 0000102000|00000000 \\ 0000003000|00000000 \\ 0000000000|00000000 \\ 0000000100|00000000 \\ 0000201000|00000000 \\ 0000000000|11100000 \\ 0000000000|00000010 \\ 0000000000|00000000 \\ 0000000000|00000300 \\ 0000000000|00010200 \\ 0000000000|00000300 \\ 0000000000|00000000 \\ 0000000000|00011001 \end{bmatrix}, \quad \gamma((n)_1) = \begin{bmatrix} 0001110000|00000000 \\ 0000000011|00000000 \\ 0000000020|00000000 \\ 0000003000|00000000 \\ 0000003000|00000000 \\ 0000003000|00000000 \\ 0000000100|00000000 \\ 0000100010|00000000 \\ 0000003000|00000000 \\ 0000000000|00021000 \\ 0000000000|11000001 \\ 0000000000|00100010 \\ 0000000000|00000300 \\ 0000000000|00000300 \\ 0000000000|00000300 \\ 0000000000|00000300 \\ 0000000000|00000000 \\ 0000000000|00000300 \end{bmatrix},$$

$$\gamma((n)_2) = \begin{bmatrix} 0000003000|00000000 \\ 0000101010|00000000 \\ 0000200010|00000000 \\ 0000003000|00000000 \\ 0000003000|00000000 \\ 0000003000|00000000 \\ 0000003000|00000000 \\ 0000100020|00000000 \\ 0000101010|00000000 \\ 0000003000|00000000 \\ 0000000000|00000300 \\ 0000000000|00002010 \\ 0000000000|10001001 \\ 0000000000|00000300 \\ 0000000000|00000300 \\ 0000000000|00000300 \\ 0000000000|01100010 \\ 0000000000|00000300 \end{bmatrix}.$$

Now we use the Berstel–Reutenauer minimization algorithm [§2][7] to determine whether the minimal linear representation of $v\gamma((n)_k)w$ is the zero representation. Our conjecture holds if and only if the minimized representation is identically zero. The minimization

yields a representation with $v = [1]$ and $w = [0]$, which implies that the difference is always zero for all n . Consequently, the conjecture is correct, and

$$\rho_{\text{MW}}(n) = 4n - 4 \quad \text{for all } n \geq 2.$$

This procedure was also used to establish the subword complexity formula for the MK7 word listed in Table 5.1.

7.2 Frid's formula

In this section, we outline a more rigorous method introduced by Anna E. Frid which provides an algorithm to compute the subword complexity of fixed points of binary k -uniform morphisms [18].

7.2.1 Preliminary definitions

A key idea in Frid's framework is the distinction between *circular* and *uncircular* morphisms words. The uncircular binary k -uniform morphisms can be defined as those adhering to one of the following four patterns:

- (i) $\begin{cases} h(0) = (01)^{x0} \\ h(1) = (10)^{x1} \end{cases} \quad \text{for } x = (k - 1)/2$
- (ii) $\begin{cases} h(0) = 0^k \\ h(1) \text{ is any binary word of length } k \end{cases}$
- (iii) $\begin{cases} h(0) = 01^{k-1} \\ h(1) \text{ is any binary word of length } k \end{cases}$
- (iv) $\begin{cases} h(0) \text{ is a length-}k \text{ binary word other than } 0^k, 01^{k-1} \\ h(1) = 1^k \end{cases}$

It is easy to see that the first two cases correspond to periodic words with $\rho(n) = 2$ for case (i) and $\rho(n) = 1$ for case (ii), while the case (iii) corresponds to an ultimately periodic word

with $\rho(n) = 2$. The fourth case corresponds to an uncircular unperiodic infinite words. The fixed point of every other binary k -uniform morphism is circular and unperiodic. Moreover, the block length b of a binary uniform morphism is defined as the sum of the lengths of the longest common prefix and the longest common suffix between the two images of the morphism. For example, the block length of the Stewart choral [17] morphism ($0 \rightarrow 001, 1 \rightarrow 011$) is $b = 1 + 1 = 2$. Finally, each morphism has a *synchronization delay*. In order to understand the definition of the synchronization delay L for an infinite circular word w , we first need to define the meaning of a synchronization point [12, 19]. Let $F(w_h)$ be the set of all factors of the fixed point w_h of the morphism h . If $u \in F(w_h)$, we say that (u_1, u_2) is a *synchronization point* of u on w_h if

$$u = u_1u_2, \text{ and } \forall v_1, v_2 \in \Sigma^*, \forall s \in F(w_h) \exists s_1, s_2 \in F(w_h) \\ [v_1uv_2 = \phi(s) \implies (s = s_1s_2, v_1u_1 = \phi(s_1), u_2v_2 = \phi(s_2))].$$

A word u is *circular* on w_h if it has at least one synchronization point on w_h . The infinite word w_h is *circular with synchronization delay L* if $\forall u \in F(w_h), (|u| \geq L \implies u \text{ is circular})$. For the uncircular unperiodic k -uniform morphisms, we define the synchronization delay as $L = \min\{l \leq k^2 : \forall n \geq l, 1^n \text{ is the only uncircular word of length } n \text{ in } w_h\}$ [18].

We can find the synchronization delay L for fixed points of k -uniform morphism using Walnut. This is due to the fact that in a k -uniform morphism, circularity of u on w means that all starting indices of occurrences u are congruent modulo k [18]. For example, to calculate the synchronization delay for the circular 3-adic word, we define the following predicates:

```
def mod3 "?msd_3 n=z+3*(n/3)":
# Is n = z (mod 3)?
# 3 states
def mod33 "?msd_3 E1,k ($mod3(i,1) & $mod3(j,k) & l=k)":
# Are i and j are congruent modulo 3?
# 2 states
def circulartdc "?msd_3 Aj ($eqtdc(i,j,n) => $mod33(i,j))":
# Is the factor TDC[[i..i+n-1]] circular on TDC?
# 5 states
def synchrotdc "?msd_3 (Ai,n ( n>=1 => $circulartdc(i,n)))":
# Are all factors with length 1 or greater circular on TDC?
# 4 states
test synchrotdc 1:
```

```
# Outputs "12" as the first accepted string of synchrotcd.
# The synchronization delay of TDC is "12" which is 5 in base 3.
```

For an uncircular unperiodic word, like the Rote word, we use the following predicates instead:

```
morphism g "0->001 1->111":
promote ROT g:
def eqrot "?msd_3 Am (m<n) => ROT[i+m] = ROT[j+m]":
# The equality of factors predicate for the Rote word
# 37 states
def circularrot "?msd_3 Aj ($eqrot(i,j,n) => $mod33(i,j))":
# Is the factor ROT[[i..i+n-1]] circular on ROT?
# 10 states
def onesrot "?msd_3 At (t<n) => ROT[i+t]=@1":
# Is the factor ROT[[i..i+n-1]] = 1^n?
# 7 states
def synchorot "?msd_3 l<=9 & (Ai,n (n>=1 & ~$circularrot(i,n)) =>
    $onesrot(i,n))":
# If n >= 1 >= 3^2, are the only uncircular factors of length n all "1"s?
# 5 states
test synchorot 1:
# Outputs "2" as the first accepted string of synchorot.
# The synchronization delay of ROT is "2" which is 2 in base 3.
```

With these basic definitions out of the way, the approach of Frid is outlined in Algorithm 2. It was adapted to directly compute $\rho(n)$ for a given input n , but it is straightforward to see how it can be modified to obtain the general piecewise-linear formula for the subword complexity. We also illustrate this with an example.

7.2.2 The algorithm and an example: the 3-adic word

Algorithm 2 Frid’s subword complexity method for binary uniform morphisms.

Procedure: FRIDFORM(H)
Input: Binary uniform morphism h , and a positive integer n .
Output: $\rho(n)$ for the fixed point of the binary uniform morphism h .

- 1: Calculate the buffer length b and the block length k of the morphism h .
- 2: Determine whether h is circular by comparing it with known cases of circular (unperiodic) k -uniform morphisms.
- 3: Find the synchronization delay L of the morphism (defined differently depending on circularity).
- 4: Compute the smallest integer t satisfying $k(t - 1) + b + 1 \geq L$.
- 5: Define the array $T = [t, \dots, k(t - 1) + b + 1]$, where T_i denotes the i -th element of T .
- 6: **for** each T_i in T except the last element **do**
- 7: Compute $\rho(T_i)$ using **Walnut** and the linear representation method (Section 7.1.2).
- 8: Compute $\Delta\rho_i = \rho(T_i + 1) - \rho(T_i)$.
- 9: **end for**
- 10: **if** $n \leq t$ **then**
- 11: Compute $\rho(n)$ using **Walnut** and the linear representation method.
- 12: **else**
- 13: **for** each T_i in T except the last element **do**
- 14: **if** $(T_i - 1) \cdot k^r + b \cdot \frac{k^r - 1}{k - 1} + 1 < n \leq T_i \cdot k^r + b \frac{k^r - 1}{k - 1} + 1$ for some constant p **then**
- 15: $\rho_{\text{circular}_i} \leftarrow \Delta\rho_i \cdot n + \Delta\rho_i \cdot \frac{b - k + 1}{k - 1} + k^r \cdot [\rho(T_i) - \Delta\rho_i \cdot (\frac{b}{k - 1} + T_i - 1)]$
- 16: $\rho_{\text{uncircular}_i} \leftarrow \rho_{\text{circular}_i}(n) - k^r + 1$
- 17: **Break**
- 18: **end if**
- 19: **end for**
- 20: **end if**
- 21: **return** ρ_{circular_i} if h is circular; otherwise $\rho_{\text{uncircular}_i}$

Now, we apply Frid’s Formula to the 3-adic morphism:

$$\begin{aligned} 0 &\mapsto 00.1. \\ 1 &\mapsto 00.0. \end{aligned}$$

The morphism does not follow one of the uncircular patterns of k -uniform binary morphisms, and thus it is a circular morphism. Also, we can immediately see block length

$b = 2 + 0 = 2$, the block length $k = 3$, and the synchronization delay $L = 5$ as we computed in the previous part.

Using simple algebraic manipulation, we get that the smallest integer t satisfying $k(t - 1) + b + 1 \geq L$ is $t = 2$. Indeed, $3 \cdot (2 - 1) + 2 + 1 = 6 \geq 5$.

Consequently, the array $T = [2, 3, 4, 5, 6]$. From Figure 7.2, we see that $\rho(2) = 3, \rho(3) = 4, \rho(4) = 6, \rho(5) = 8, \rho(6) = 9$. Furthermore, $\Delta\rho_0 = 4 - 3 = 1, \Delta\rho_1 = 2, \Delta\rho_2 = 2, \Delta\rho_3 = 1$. Now, we illustrate the computation of $\rho(n)$ for different values of i as follows:

1. At $i = 0$, we have $T_i = 2, \rho(T_i) = 3$, and $\Delta\rho_i = 1$. Then

$$\text{if } 2 \cdot 3^r \leq n < 3 \cdot 3^r, \quad \rho(n) = n + 3^r. \quad (7.1)$$

2. At $i = 1$, we have $T_i = 3, \rho(T_i) = 4$, and $\Delta\rho_i = 2$. Then

$$\text{if } 3 \cdot 3^r \leq n < 4 \cdot 3^r, \quad \rho(n) = 2n - 2 \cdot 3^r. \quad (7.2)$$

3. At $i = 2$, we have $T_i = 4, \rho(T_i) = 6$, and $\Delta\rho_i = 2$. Then

$$\text{if } 4 \cdot 3^r \leq n < 5 \cdot 3^r, \quad \rho(n) = 2n - 2 \cdot 3^r. \quad (7.3)$$

4. At $i = 3$, we have $T_i = 5, \rho(T_i) = 8$, and $\Delta\rho_i = 1$. Then

$$\text{if } 5 \cdot 3^r \leq n < 6 \cdot 3^r, \quad \rho(n) = n + 3 \cdot 3^r. \quad (7.4)$$

It is easy to see that intervals (7.2) and (7.3) can be grouped together, and in fact, we can also rewrite interval (7.1) and group it with interval (7.4) by substituting r with $r + 1$ in (7.4) as follows:

$$\text{If } 2 \cdot 3^{r+1} = 6 \cdot 3^r \leq n < 3 \cdot 3^{r+1}, \quad \rho(n) = n + 3^{r+1} = n + 3 \cdot 3^r. \quad (7.5)$$

$$\text{Thus, for } n > 2, \text{ we have } \rho_{3\text{-adic}}(n) = \begin{cases} 2n - 2 \cdot 3^r, & \text{if } 3 \cdot 3^r \leq n < 5 \cdot 3^r \\ n + 3 \cdot 3^r, & \text{if } 5 \cdot 3^r \leq n < 3 \cdot 3^{r+1}, \end{cases}$$

Chapter 8

Open problems

Open Problem 1. *Give a characterization for the k -automatic sequences that have an exponential blowup in size when constructing the “equality of factors” automaton. Currently no such class of examples is known.*

This issue has two distinct aspects. The first concerns the size of the minimal automaton for the equality of factors predicate relative to the size of the morphism. We believe that this relationship is exponential, but currently, no class of examples is known that proves this behavior. The second aspect involves determining the most efficient strategy for constructing this automaton, whether by guessing, using `Walnut` or some alternative method. Currently, we have no good bounds on how to do this efficiently.

As an example of the second issue, we refer back to the intuitive construction of the predicate $\text{EqFac}(i, j, n) := \forall t (t < n) \implies \mathbf{x}[i+t] = \mathbf{x}[j+t]$ using `Walnut`. As mentioned previously in Section 1.2, this approach is quite inefficient for constructing the equality of factors predicate for the Tribonacci word.

```
def trib_eqfac "?msd_trib At (t<n) => TR[i+t]=TR[j+t]":
```

The final automaton has only 26 states, while the largest intermediate automaton has 323,831,403 states, which is more than a 12.45 million-fold increase. Execution required over 300 GB of RAM and took 432,831,386 ms over several days.

However, there are possible alternative formulations that run much faster and in a reasonable amount of time. For instance, constructing the following equivalent predicate took only 16307 ms, and the largest intermediate automaton had 18,853 states:

```
def trib_eqfac_f "?msd_trib Au,v (u>=i & u<i+n & u+j=v+i) => TR[u]=TR[v]":
```

This is despite the fact that `trib_eqfac_f` is bounded by $2^{O(m^2)}$ states, where m is the number of states of the DFAO for the Tribonacci word `tr`. More generally, automata in base k constructed using a similar formulation for the equality of factors predicate are bounded by 2^{9m^2} states, where m is the number of states in the DFAO of the corresponding k -automatic sequence.

Open Problem 2. *Give a characterization for the logical formulas that are self-verifying predicates.*

So far, no general theory exists that specifies which kinds of logical formulas our method of self-verifying predicates can handle. Nevertheless, the idea behind it is widely applicable and adaptable to a number of situations, as we showed in Section 3:

1. Periods of factors
2. Equality of reversals of factors
3. Creating an adder for a numeration system
4. Summation of synchronized sequence

Open Problem 3. *Extend the methods of calculating the critical exponent and subword complexity to non-uniform morphisms.*

Although our method for calculating the critical exponent applies to all k -automatic sequences, our approaches to subword complexity are more restricted. The heuristic method works only for binary k -automatic sequences, whereas Frid's formula applies solely to binary k -uniform morphisms. Consequently, our techniques can be used for the fixed points of some, but not all, non-uniform morphisms, since every k -automatic sequence is the image of the fixed point of a non-uniform morphism [2]. Nevertheless, it is clear that a general deterministic procedure is needed to compute the critical exponent and subword complexity for fixed points of all non-uniform morphisms.

Open Problem 4. *Find an efficient algorithm for computing the synchronization delay of an infinite word or fixed points of non-uniform morphisms or k -automatic sequences.*

Aside from the algorithm in Section 7.2.2, we present another method for computing the synchronization delay of a binary uniform morphism using Walnut. This approach can, in principle, be adapted to handle some non-uniform morphisms as well. However, the problem is that it is already very inefficient even for uniform morphisms as we shall see. The idea is to reformulate the definition of synchronization delay in terms of indices and lengths, since Walnut can operate only on non-negative integers and not directly on binary words. To begin, we assign names to each binary word appearing in the definition:

```
s = X[i0 ... i0+l0-1], h(s) = X[k*i0 ... k*(i0+l0)-1],
s1 = X[i1 ... i1+l1-1], h(s1) = X[k*i1 ... k*(i1+l1)-1],
s2 = X[i2 ... i2+l2-1], h(s2) = X[k*i2 ... k*(i2+l2)-1],
u = X[i3 ... i3+l3-1], u1 = X[i4 ... i4+l4-1], u2 = X[i5 ... i5+l5-1],
v1 = X[i6 ... i6+l6-1], v2 = X[i7 ... i7+l7-1]
```

Of course, there are a few conditions we need to impose later on, which are $l1+l2 = 10$, $i4 = i3$, $i5 = i3+l3-l5$, and finally $l4+l5 = l3$. Now, we can use the following Walnut predicates to calculate the synchronization delay of the 3-adic word:

```
def eq0 "?msd_3 $eqtdc(i3, i4, l4) & $eqtdc(i3+l4, i5, l5) &
    l4+l5=l3 & i4=i3 & i5=i3+l3-l5":
# Check u = u1 u2
# 4 states
def eq1 "?msd_3 $eqtdc(i6, 3*i0, l6) & $eqtdc(i3, 3*i0+l6, l3) &
    $eqtdc(i7, 3*i0+l6+l3, l7) & l6+l3+l7=3*(i0+l0)-1":
# Check v1 u v2 = h(s)
# 3053 states
def eq2 "?msd_3 $eqtdc(i0, i1, l1) & $eqtdc(i0+l1, i2, l2) & l0=l1+l2":
# Check s = s1s2
# 92 states
def eq3 "?msd_3 $eqtdc(i6, 3*i1, l6) & $eqtdc(i4, 3*i1+l6, l4) &
    l4+l6=3*(i1+l1)-1":
# Check v1 u1 = h(s1)
# 291 states
def eq4 "?msd_3 $eqtdc(i5, 3*i2, l5) & $eqtdc(i7, 3*i2+l5, l7) &
    l5+l7=3*(i2+l2)-1":
# Check u2 v2 = h(s2)
# 291 states
def rhs "?msd_3 $eq2(i0,i1,i2, l0,l1,l2) & $eq3(i1,i4,i6, l1,l4,l6) &
```

```

    $eq4(i2,i5,i7, 12,15,17)":
# The consequent in the definition of synchronization delay
def synchpoint "?msd_3 $eq0(i3,i4,i5, 13,14,15) & (Ai6,16,i7,17 (Ai0,10 (
    Ei1,11,i2,12 ( ($eq1(i0,i3,i6,i7, 10,13,16,17)) =>
    ($rhs(i0,i1,i2,i4,i5,i6,i7, 10,11,12,14,15,16,17)) ))))":
# Is (u1, u2) is a synchronization point of u on TDC?
def circular "?msd_3 Ei4,14,i5,15 $synchpoint(i3,i4,i5, 13,14,15)":
# Is u circular on TDC?
def synchdelay "?msd_3 Ai3,13 (13 >= L => $circular(i3,13))":
# is TDC a circular word with some synchronization delay L?
test synchdelay 1:
# Output the first accepted string by the $synchdelay predicate (L)

```

Sadly, however, we were unable to generate the `rhs` automaton. The main reason is that the predicate involves 14 free variables, which results in a very large automaton that `Walnut` cannot construct without producing enormous intermediate automata. Even attempting to build the `rhs` automaton in two stages by splitting the conjunctions failed to terminate for us in a reasonable length of time. Therefore, the number of free variables must be significantly reduced to have any realistic chance of successfully generating this automaton, at least with the computers we have access to.

However, in theory, it is possible to modify this approach to work for non-uniform morphisms as well if they are k -automatic. The main challenge is that we cannot calculate the starting indices and lengths of $h(s)$, $h(s_1)$, and $h(s_2)$ directly by multiplying by a constant. One possible workaround is to use the symbol-counting predicate, mentioned at the end of Section 3.5, until the start of the factor s and then multiply each symbol count by the length of its image to obtain the starting index of $h(s)$. Similarly, the length $|h(s)|$ can be expressed in terms of $l_0 = |s|$ by using the starting and ending indices of s together with the symbol-counting predicate. The same procedure can be applied to s_1 and s_2 .

The main difficulty, of course, is that this not only introduces additional complexity and free variables to an already challenging problem, but we also do not know a priori whether the required symbol-counting automaton exists.

References

- [1] J.-P. Allouche and J. Shallit. *Automatic Sequences: Theory, Applications, Generalizations*. Cambridge University Press, 2003.
- [2] J.-P. Allouche and J. Shallit. Automatic sequences are also non-uniformly morphic. In Andrei M. Raigorodskii and Michael Th. Rassias, editors, *Discrete Mathematics and Applications*, pages 1–6, Cham, 2020. Springer International Publishing.
- [3] J.-P. Allouche and J. Shallit. Additive properties of the evil and odious numbers and similar sequences. *Functiones et Approximatio Commentarii Mathematici*, 70(1):55 – 69, 2024.
- [4] D. Angluin. Learning regular sets from examples and counterexamples. *Info. Comput.*, 75:87–106, 1987.
- [5] M. Rigo B. Mignoty, A. Renard and M.A. Whiteland. Automatic proofs in combinatorial game theory. ORBi preprint, University of Liège, 2024. Available at <https://orbi.uliege.be/handle/2268/323845>.
- [6] E. Bach and J. Shallit. *Algorithmic number theory*. MIT Press, Cambridge, MA, USA, 1996.
- [7] J. Berstel and C. Reutenauer. *Noncommutative rational series with applications*. Number 137 in Encyclopedia of Mathematics and Its Applications. Cambridge University Press, 2011.
- [8] V. Bruyère and G. Hansel. Bertrand numeration systems and recognizability. *Theoret. Comput. Sci.*, 181:17–43, 1997.
- [9] V. Bruyère, G. Hansel, C. Michaux, and R. Villemaire. Logic and p -recognizable sets of integers. *Bull. Belgian Math. Soc.*, 1:191–238, 1994. Corrigendum: vol. 1 (1994), 577.

- [10] J.R. Büchi. Weak second-order arithmetic and finite automata. *Zeitschrift für mathematische Logik und Grundlagen der Mathematik*, 6:66–92, 1960. Reprinted in S. Mac Lane and D. Siefkes (eds.), *The Collected Works of J. Richard Büchi*, Springer-Verlag, 1990, pp. 398–424.
- [11] J. Bult. The lattice of machine invariant sets and subword complexity. ArXiv preprint arXiv:cs/0502064 [cs.CR], 2005. Available at <https://arxiv.org/abs/cs/0502064>.
- [12] J. Cassaigne. An algorithm to test if a given circular HDOL-language avoids a pattern. In *IFIP Congress (1)*, pages 459–464, 1994.
- [13] J. Cassaigne and F. Nicolas. Factor complexity. In V. Berthé and M. Rigo, editors, *Combinatorics, Automata and Number Theory*, pages 163–247. Cambridge University Press, 2010.
- [14] L. Schaeffer D. Goč and J. Shallit. Subword complexity and k -synchronization. In *International Conference on Developments in Language Theory*, pages 252–263. Springer, 2013.
- [15] L. Schaeffer D. Moradi and J. Shallit. State complexity of linear subsequences of k -automatic sequences. Manuscript in preparation, October, 2025.
- [16] L. Bélanger E. Barcucci and S. Brlek. On Tribonacci sequences. *Fibonacci Quart.*, 42:314–319, 2004.
- [17] G. Fici and J. Shallit. Properties of a class of Toeplitz words. *Theoretical Computer Science*, 922:1–12, 2022.
- [18] A.E. Frid. The subword complexity of fixed points of binary uniform morphisms. In B. S. Chlebus and L. Czaja, editors, *FCT 1997*, volume 1279 of *Lecture Notes in Comp. Sci.*, pages 179–187. Springer, 1997.
- [19] A.E. Frid. On uniform DOL words. In M. Morvan, C. Meinel, and D. Krob, editors, *STACS 98*, volume 1373 of *Lecture Notes in Comp. Sci.*, pages 544–554. Springer, 1998.
- [20] C. Frougny. Representations of numbers and finite automata. *Mathematical Systems Theory*, 25:37–60, 1992.
- [21] C. Frougny and B. Solomyak. On representation of integers in linear numeration systems. In M. Pollicott and K. Schmidt, editors, *Ergodic Theory of \mathbb{Z}^d Actions*

- (*Warwick, 1993–1994*), volume 228 of *London Mathematical Society Lecture Note Series*, pages 345–368. Cambridge University Press, 1996.
- [22] M. Khodier G. Fici and J. Shallit. Critical exponent and subword complexity for binary 3-uniform morphisms. Unfinished Manuscript, 2025.
- [23] L. Schaeffer H. Mousavi and J. Shallit. Decision algorithms for Fibonacci-automatic words, I: basic results. *RAIRO Theoret. Inform. Appl.*, 50:39–66, 2016.
- [24] G.H. Hardy and E.M. Wright. *An introduction to the theory of numbers*. Oxford university press, 1979.
- [25] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.
- [26] S.L. Shan J. Shallit and K.H. Yang. Automatic sequences in negative bases and proofs of some conjectures of Shevelev. *RAIRO Theoret. Inform. Appl.*, 57:Paper 4, 2023.
- [27] D. Krieger. On critical exponents in fixed points of non-erasing morphisms. *Theoret. Comput. Sci.*, 376:70—88, 2007.
- [28] D. Krieger. *Critical Exponents and Stabilizers of Infinite Words*. PhD thesis, University of Waterloo, 2008.
- [29] D. Krieger. On critical exponents in fixed points of k -uniform binary morphisms. *RAIRO Inf. Theor. Appl.*, 43:41–68, 2009.
- [30] L. Schaeffer M. Khodier and J. Shallit. Self-verifying predicates in Büchi arithmetic. ArXiv preprint arXiv:2507.19717 [cs.FL], 2025. Available at <https://arxiv.org/abs/2507.19717>.
- [31] L. Schaeffer M. Khodier and J. Shallit. Self-verifying predicates in Büchi arithmetic. In G. Castiglione and S. Mantaci, editors, *Implementation and Application of Automata*, volume 15981 of *Lecture Notes in Computer Science*, pages 237–251. Springer Nature Switzerland, 2026.
- [32] O. Merkurev and A.M. Shur. Computing the maximum exponent in a stream. *Algorithmica*, 84(3):742–756, 2022.
- [33] H. Mousavi. Automatic theorem proving in Walnut. ArXiv preprint arXiv:1603.06017 [cs.FL], 2016. Available at <https://arxiv.org/abs/1603.06017>.

- [34] M. Delacourt O. Carton, J.-M. Couvreur and N. Ollinger. Linear recurrence sequence automata and the addition of abstract numeration systems. ArXiv preprint arXiv:2406.09868 [cs.FL], 2025. Available at <https://arxiv.org/abs/2406.09868>.
- [35] N. Rampersad and J. Shallit. Rudin–Shapiro sums via automata theory and logic. *Theory of Computing Systems*, 69(1):9, 2025.
- [36] L. Schaeffer. Deciding properties of automatic sequences. Master’s thesis, University of Waterloo, 2013. Available at <https://cs.uwaterloo.ca/~shallit/thesisLukeSept4.pdf>.
- [37] L. Schaeffer and J. Shallit. The critical exponent is computable for automatic sequences. *International Journal of Foundations of Computer Science*, 23(08):1611–1626, 2012.
- [38] J. Shallit. Synchronized sequences. In T. Lecroq and S. Puzynina, editors, *WORDS 2021*, volume 12847 of *Lecture Notes in Computer Science*, pages 1–19. Springer-Verlag, 2021. Available at https://doi.org/10.1007/978-3-030-85088-3_1.
- [39] J. Shallit. *The Logical Approach To Automatic Sequences: Exploring Combinatorics on Words with Walnut*, volume 482 of *London Mathematical Society Lecture Note Series*. Cambridge University Press, 2023.
- [40] J. Shallit. Proving properties of φ -representations with the Walnut theorem-prover. *Communications in Mathematics*, Volume 33 (2025), Issue 2 (Special issue: Numeration, Liège 2023, dedicated to the 75th birthday of professor Christiane Frougny), Sep 2024.
- [41] J. Shallit. Rarefied Thue-Morse sums via automata theory and logic. *J. Number Theory*, 257:98–111, 2024.
- [42] J. Shallit and M. Stipulanti. Algorithms for linear representations of k -regular sequences. Manuscript in preparation, 2025.
- [43] J. Shallit and A. Zavyalov. Transduction of automatic sequences and applications. In Benedek Nagy, editor, *Implementation and Application of Automata*, pages 266–277, Cham, 2023. Springer Nature Switzerland.
- [44] N.J.A. Sloane et al. The on-line encyclopedia of integer sequences. Available at <https://oeis.org>.

- [45] B. Tan and Z.-Y. Wen. Some properties of the Tribonacci sequence. *Europ. J. Combin.*, 28:1703–1719, 2007.