COMPUTABILITY OF UNAMBIGUOUS GENERATING SETS OF CODED SHIFTS

TAMARA KUCHERENKO AND BENJAMIN TUPPER

Abstract. In a constructive argument, Béal *et al.* demonstrate that any coded shift can be represented as an unambiguously coded shift. In this note, we show that such an unambiguous code is computable. More precisely, there exists an oracle Turing machine that, given access to a generating set, produces arbitrarily large finite approximations of the corresponding unambiguous code. Our algorithm determines how many words from the original set are required to compute a prescribed number of words in the unambiguous code. As an illustration, we implement this algorithm in the programming language Python.

1. INTRODUCTION

The concept of coded shifts, a broad class of shift spaces encompassing transitive shifts of finite type and sofic shifts, was first proposed by Blanchard and Hansel [1]. Consider a finite alphabet $\mathcal{A}$ and a set C of finite words over $\mathcal{A}$, which we refer to as a *generating set*. A shift space X is *coded by* C if X is the closure of the set of bi-infinite concatenations of words in C. Then X is a *coded shift*. Equivalently, X is a coded shift if there exists a set C such that the *language of* X, denoted by $\mathcal{L}(X)$, is the set of all *factors* of C^* [4]. Here, C^* denotes the set of all finite concatenations of words from C (including the empty word), and a word w is a *factor* of a word v if w appears as a contiguous subword of v.

It is not immediately apparent whether a finite concatenation of words from a given set C admits a unique factorization into elements of C, nor whether a bi-infinite concatenation of words from C has a unique such decomposition. We say that C is a *code* if every finite concatenation of words from C admits a unique decomposition into elements of C. A subshift X is *unambiguously coded* if there exists a code C such that every point $x \in X$ has at most one representation as a concatenation of words from C [4].

For finite blocks, a method exists to determine whether a given generating set C can produce a word $c_1 c_2 \cdots c_n$ with two distinct factorizations in C. An elegant and efficient algorithm for this problem was introduced by Sardinas and Patterson [10]. For bi-infinite sequences, the situation is more subtle. It was

Key words and phrases. Computability, symbolic dynamics, shift spaces, coded shifts.

shown partially by Fiebig and Fiebig [6] and later fully by Béal *et al.* [4] that every coded shift can be represented as an unambiguous one. In [4], Béal *et al.* extend the argument of Fiebig and Fiebig and give a constructive proof that produces an unambiguous code from a given generating set.

The objective of this paper is to implement the construction formulated in [4] as an explicit algorithm and to establish the computability of an unambiguous generating set. While the construction in [4] is constructive in a mathematical sense, it is not computable in the sense of classical computability theory. In particular, the original argument operates with infinite generating sets, bi-infinite concatenations, and implicitly assumes the existence of enumerations of the resulting code without specifying an effective procedure for producing them.

From a computability-theoretic perspective, these assumptions are nontrivial: a Turing machine can only process finite input and produce finite output in finite time. Consequently, a constructive existence proof does not automatically yield a computable algorithm. In [4], the construction of an unambiguous code relies on infinite objects and global choices that cannot be directly implemented on a finite computer, nor does it specify how to uniformly approximate the resulting code by finite data.

The main contribution of the present work is to bridge this gap between theory and computation. We develop an explicit algorithm that, given a finite subset of a (possibly ambiguous) code, produces any prescribed finite number of words of an unambiguous code. This is achieved by employing approximation techniques from computability theory, allowing the algorithm to work with finite representations while faithfully reflecting the infinite construction. Moreover, we provide a computable and stable enumeration of the resulting code: the initial words of the unambiguous code remain unchanged when a larger output is requested. This stability property is essential for practical computation and is not addressed in the theoretical construction of [4].

In this way, our work makes the construction of unambiguous generating sets fully effective and suitable for implementation, thereby extending its applicability beyond a purely theoretical setting.

Implementing and analyzing a real-word example in Python highlights several potential areas for future research. One key area of interest involves the convergence of coded shifts. Let ϕ_n be a set containing the initial n words of C as generated by the algorithm. The result demonstrates that $X(C)$ is the limit of the shift spaces coded by ϕ_n, that is, $X(C) = \lim_{n\to\infty} X(\phi_n)$. However, the convergence rate remains unclear. Future research could consider the entropy or pressure of the convergent coded shifts, situating this algorithm in the context of other potential candidate algorithms.

Several practical questions arise from this construction. The algorithm always returns an infinite unambiguous code. Even with a finite generating set,

the resulting code remains infinite with this construction. Do algorithms exist that produce more tractable or efficient unambiguous codes? Ideally, such an algorithm would generate a finite unambiguous code whenever possible. In cases where an infinite code is necessary, it would generate one that optimizes certain constraints, such as syntactic simplicity or minimality. Such an advancement would enhance our ability to explore coded shifts by providing researchers with practical and efficient tools.

The organization of this paper is as follows. In Section 2 we discuss preliminary concepts including shift spaces, coded shifts, and unambiguously coded shifts. We review graph presentations of shifts and their properties, as well as the concept of computability defined in terms of oracles approximations. Additionally, we view our algorithm in the context of computability. In Section 3 we outline the general structure of the constructive proof from [4] that our algorithm is based on. We then translate the specifics of the proof into Python and provide an example. We discuss the practical considerations involved in translating a theoretical framework into executable code. An implementation of our algorithm in Python is given in Appendix A and an illustrative example in Appendix B.

Acknowledgement. This work was substantially advanced during the 10-week Rich Summer Internship Program at the City College of New York. We thank Dr. Barnett and the Jean Hollander Rich Fund for their generous support.

We also thank the anonymous referee for their detailed and constructive comments, which which substantially improved the exposition and clarity of the paper.

2. PRELIMINARIES

A more detailed account of the concepts covered in this section can be found in [8], which serves as the primary reference for the material presented here.

2.1. **Shift spaces.** The fundamental object in our field of study is the topological *shift space* which is composed of *bi-infinite sequences.* Let $\mathcal{A}$ be a finite alphabet. We call the elements of $\mathcal{A}$ *letters* or *symbols.* While an infinite sequence of letters typically extends infinitely to the right, a bi-infinite sequence is a sequence of letters extending infinitely in both directions:

$$x = \dots a_{-2}a_{-1} \, . \, a_0a_1a_2\dots \qquad a_i \in \mathcal{A}$$

The period between a_{-1} and a_0 indicates the position of the zeroth element of the bi-infinite sequence (here it is zero-indexed to a_0). The set of all bi-infinite sequences over the alphabet $\mathcal{A}$ is called the *full-shift over* $\mathcal{A}$, denoted by $\mathcal{A}^{\mathbb{Z}}$. We define a metric on $\mathcal{A}^{\mathbb{Z}}$ as $d(x, y) = 2^{-|n|}$ where n is the smallest in absolute value such that $x_n \neq y_n$. This metric induces the product topology on $\mathcal{A}^{\mathbb{Z}}$.

The shift map, denoted by σ, shifts each letter of a sequence one position to the left, while σ^{-1} shifts letters to the right. Specifically,

$$\sigma: \ (\dots x_{-1} \ . \ x_0 x_1 x_2 \dots) \mapsto (\dots x_{-1} x_0 \ . \ x_1 x_2 \dots)$$

We may restrict the domain of σ to a subset $X \subset \mathcal{A}^{\mathbb{Z}}$ as long as X is closed under shifts, i.e., $\sigma^n(x) \in X$ for all $n \in \mathbb{Z}$. Then we say X is *shift invariant.* These two kinds of closure — under shifts and under the topology — give rise to the definition of a shift space. We say $X \subseteq \mathcal{A}^{\mathbb{Z}}$ is a *shift space* if and only if X is topologically closed and shift invariant. A shift space is sometimes called a *shift* or, to emphasize when $X \neq \mathcal{A}^{\mathbb{Z}}$, a *subshift.* The full-shift $\mathcal{A}^{\mathbb{Z}}$ is a shift space.

A *block* or *word* w is a finite string of letters from a finite alphabet $\mathcal{A}$. For example, if $\mathcal{A} = \{0, 1\}$, then $w_1 = 110$ and $w_2 = 1$ are both blocks. If a block w occurs in a sequence x we write $w \in x$. The set of all blocks of all sequences of a shift space X is the *language* of the shift space, denoted by $\mathcal{L}(X)$.

We can also characterize a shift space using *forbidden blocks.* Let $\mathcal{A}$ be an alphabet and $\mathcal{F} = \{w_1, w_2, \dots\}$ be a set of blocks. Let $\mathsf{X}_{\mathcal{F}}$ be the set of bi-infinite sequences that do not contain any block from $\mathcal{F}$, that is,

$$\mathsf{X}_{\mathcal{F}} = \{x \in \mathcal{A}^{\mathbb{Z}} \mid w \notin x \text{ for all } w \in \mathcal{F}\}.$$

It can be easily verified that $\mathsf{X}_{\mathcal{F}}$ is shift-invariant and closed under the product topology. Moreover, every shift space can be obtained in this way, i.e., $X \subseteq \mathcal{A}^{\mathbb{Z}}$ is a shift space if and only if $X = \mathsf{X}_{\mathcal{F}}$ for some set of *forbidden blocks* $\mathcal{F}$.

Example 1. Let $\mathcal{A} = \{0, 1\}$ and $\mathcal{F} = \{11\}$. Then $X = \mathsf{X}_{\mathcal{F}}$ is the set of all bi-infinite sequences not containing two adjacent 1's. This shift space is called the *golden mean shift.* The language of X is

$$\mathcal{L}(X) = \{0,\ 1,\ 00,\ 01,\ 10,\ 000,\ 001,\ 010,\ 100,\ 101,\ 0000, \dots\}$$

Example 2. Let $\mathcal{A} = \{0, 1\}$ and $\mathcal{F} = \{10^k 1 \mid k \text{ is odd}\}$. Then $X = \mathsf{X}_{\mathcal{F}}$ is the set of all bi-infinite sequences such that between any two 1's there is an even number of 0's. This is called the *even shift.* The language of X is

$$\mathcal{L}(X) = \{0,\ 1,\ 00,\ 01,\ 10,\ 11,\ 000,\ 001,\ 010,\ 011,\ 100,\ 110,\ 111,\ 0000, \dots\}$$

2.2. **Shifts of finite type.** We note that two distinct sets $\mathcal{F}$ and $\mathcal{F}'$ of forbidden blocks can define the same shift space. For example, if $\mathcal{F} = \{11\}$ and $\mathcal{F}' = \{11, 111, 1111, \dots\}$, then $\mathsf{X}_{\mathcal{F}} = \mathsf{X}_{\mathcal{F}'}$, namely the golden mean shift. A set of forbidden blocks may be finite (see Example 1) or infinite (see Example 2). We say that X is a *shift of finite type* (SFT) if there exists a finite set $\mathcal{F}$ such that $X = \mathsf{X}_{\mathcal{F}}$. The even shift is not of finite type, since any set of forbidden blocks defining it must be infinite; in particular, forbidding only finitely many

words cannot enforce the requirement that blocks of consecutive 0's have even length.

2.3. **Sofic shifts.** A shift space can be represented by a directed graph. Consider the graph $\mathcal{G}$ with vertices $V = \{a, b\}$ and edges labeled by $E = \{0, 1\}$ (Figure 1). For a directed edge, say e, we denote the source (initial) vertex of e as $i(e)$ and the target (terminal) vertex of e as $t(e)$. The label of an edge e is denoted by $L(e)$.

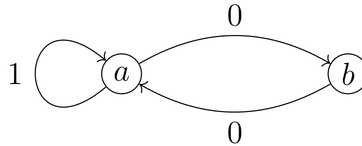

FIGURE 1. The graph $\mathcal{G}$ of the even shift X

We define a *path* π in a graph as a sequence of edges such that $t(e_i) = i(e_{i+1})$ for every consecutive pair of edges in π. We are interested in both finite and bi-infinite paths. The label of a path $L(\pi) = L(\dots e_{-1}e_0e_1e_2\dots)$ is the sequence of the labeled edges along the path:

$$L(\pi) = \dots L(e_{-1})L(e_0)L(e_1)L(e_2)\dots.$$

We say that a labeled graph $\mathcal{G}$ *represents* a shift space X if X is the closure of the set of labels of all bi-infinite walks on $\mathcal{G}$. Similarly, we say that $\mathcal{G}$ *recognizes* the language of X, denoted by $\mathcal{L}(X)$, if the set of labels of all finite paths in $\mathcal{G}$ coincides with the set of allowable finite words of X. Thus, recognizing a shift is a local, finite condition on words, whereas representing a shift is a global condition involving bi-infinite extensions.

The graph $\mathcal{G}$ shown in Figure 1 both represents the even shift and recognizes its language. In particular, no bi-infinite walk on $\mathcal{G}$ produces a block of the form 10^k1 with k odd. If one modifies the graph in Figure 1 by adding a dead-end path labeled 1 to vertex b, the resulting graph no longer recognizes the language of the even shift, since it admits a finite path labeled 101, which is not an allowable word. Nevertheless, this modified graph still represents the even shift, as the added path cannot occur as part of any bi-infinite walk.

On the other hand, one can easily construct a countable labeled graph that recognizes the language of the even shift but does not represent the shift itself. Consider a graph with a distinguished vertex a, and for each word a in the language of the even shift, attach a finite directed path starting at a whose sequence of edge labels is exactly w. By construction, the set of labels of finite paths in this graph coincides with the language of the even shift, so the graph recognizes the language. However, every path in the graph is finite and

originates at a, so the graph admits no bi-infinite walks. Consequently, the graph does not represent the even shift.

The notions of representing a shift space and recognizing its language do not coincide for arbitrary labeled graphs, as illustrated by the examples above. However, they agree under a standard connectivity assumption. Informally, a labeled graph is strongly connected if any vertex can be reached from any other vertex by following a directed path. If $\mathcal{G}$ has this property and admits at least one bi-infinite path, then every finite labeled path can be extended to a bi-infinite walk by inserting suitable connecting paths, using the fact that all vertices communicate with one another. Consequently, every finite word arising as the label of a path in $\mathcal{G}$ appears as a subword of some bi-infinite label sequence, and conversely every finite subword of a bi-infinite label sequence arises from a finite path. Hence the set of labels of finite paths coincides with the language generated by bi-infinite walks, and therefore $\mathcal{G}$ recognizes $\mathcal{L}(X)$ if and only if $\mathcal{G}$ represents X.

Remark. *Some authors use terms from the language of computation and refer to such graphs as an* automata [4] *or a* presentation [8]*. However, we will use the term* graph *and we say $\mathcal{G}$ is a graph of the shift space $X = X(\mathcal{G})$ or that $\mathcal{G}$ realizes X.*

Although the even shift cannot be defined by a finite set of forbidden words, it can be defined using a finite graph. This gives rise to a broader class of shift spaces that properly includes SFTs. Let $X \subseteq \mathcal{A}^{\mathbb{Z}}$ be a subshift. We say X is a *sofic shift* if there is a finite directed graph $\mathcal{G}$ such that X is the closure of the set of labels of bi-infinite walks on $\mathcal{G}$.

A sofic shift can alternatively be defined by its language $\mathcal{L}(X)$. We say a language is *regular* if it can be described by regular expressions. Equivalently, a language is regular if it is closed under union, intersection, concatenation, and Kleene star operations (where the Kleene star of a language denotes the set of all finite concatenations of words in that language, including the empty word). A shift X is a *sofic shift* if and only if its language $\mathcal{L}(X)$ is regular.

2.4. **Coded shifts.** An even more general type of shift is the *coded shift,* introduced by Blanchard and Hansel [1]. Let $C = \{c_1, c_2, c_3, \ldots\}$ be a (possibly infinite) set of words. Consider the set of bi-infinite sequences composed of concatenations of words from C, i.e.,

$$X' = \{\ (\ldots c_{i_{-1}} c_{i_0} c_{i_1} \ldots) \mid c_{i_k} \in C\}$$

The set X' is shift-invariant but may not be closed, so it is not necessarily a shift space. For example, consider the set

$$C = \{1, 100, 10000, \ldots, 1(00)^k, \ldots\}$$

and let X' be the set of all bi-infinite concatenations of words from C. Although X' closely resembles the even shift, it notably excludes the bi-infinite sequence $x_0 = 0^\infty$. However, x_0 is the limit of the sequence $x_1, x_2, x_3, \ldots$ where $x_k = (1(00)^k)^\infty \in X'$, with the origin chosen in the middle of one of the $1(00)^k$ blocks. Thus X' is not closed in a topological sense and therefore not a shift space. To ensure we have a shift space, we must take the closure $X = \mathrm{cl}(X')$, resulting in a even shift.

Let C be a set of words over an alphabet $\mathcal{A}$. The *coded shift generated by* C, denoted by $X(C)$, is the smallest subshift of $\mathcal{A}^\mathbb{Z}$ containing every bi-infinite concatenation of words from C. In other words, if X' is the set of all bi-infinite concatenations of words from C, then $X(C)$ is the closure of X'. We call C a *generating set*.

We now review some properties of generating sets. Consider the following sets C, D, and E :

$$C = \{a, ab, bc, c\}$$
$$D = \{01, 10\}$$
$$E = \{x, xy, xyz\}$$

An issue with the set C is that the word abc can be factored in two different ways: $abc = a \cdot bc = ab \cdot c$. If a sender transmits the string abc over a channel, the words of the message will appear ambiguous to a receiver. Thus we consider the following property for generating sets. We say a set of words C over an alphabet $\mathcal{A}$ is a *code* if every word over $\mathcal{A}$ has at most one factorization into words in C [4]. If a set C satisfies this property, it is also called *uniquely decipherable* [9]. Notably, every coded shift is generated by some code [1]. However, even if a generating set C is not a code, $X(C)$ is a coded shift.

The set D is a code since any finite word over $\mathcal{A}$ factors uniquely into words in D. However, D has a problematic characteristic: The shift generated by D, i.e., $X(D)$, contains a bi-infinite sequence with two factorizations: $(01)^\infty$ and $(10)^\infty$. Therefore, we consider a stronger property for coded shifts. Following [4], we say that X is *unambiguously coded by* C if every bi-infinite sequence $x \in X$ has at most one factorization into words in C. If there exists a set C that unambiguously codes a shift X, we say X is an *unambiguously coded shift.* It has been proven that every coded shift is unambiguously coded [4]. In this paper, we translate the proof of this result into an executable algorithm that generates such codes.

The set E is an example of a generating set that codes a shift space $X = X(E)$ which is both a uniquely decipherable and an unambiguous code.

In recent years, there has been a growing interest in the study of coded shifts, reflecting their importance in theory and practice. Notable contributions include works that explore computability, statistical properties and entropy, as

seen in the recent literature [2, 5, 9, 7]. We primarily follow the terminology of [7].

The set of points constructed from infinite concatenations of C is called the concatenation set $X_{\text{con}}(C)$; while the set of limit points of X not contained in $X_{\text{con}}(C)$ is called the residual set $X_{\text{res}}(C)$ [7]. Together, $X_{\text{con}}(C)$ and $X_{\text{res}}(C)$ partition X. It is possible for two generating sets to define the same shift but partition the shift differently. For example, the sets $C = \{1(00)^k \mid k = 0, 1, 2, \ldots\}$ and $D = \{1, 00\}$ both define the even shift, but $X_{\text{res}}(D)$ is empty while $X_{\text{res}}(C)$ is not; in particular, it contains 0^∞, among other points.

As mentioned earlier, every coded shift is recognized by some graph $\mathcal{G}$, although not necessarily by a finite graph. If a generating set C is finite, then $X(C)$ is recognized by a finite graph and is therefore sofic. In this case, every admissible sequence is already realized by a bi-infinite path in the graph, since any limit of concatenations of words from a finite set can still be represented by an infinite concatenation of those words. Thus taking the closure does not introduce any additional points, and hence the residual set is empty. On the other hand, if C is infinite, then it is possible that $X(C)$ is not sofic; that is, $X(C)$ is not represented by any finite graph and its language $\mathcal{L}\big(X(C)\big)$ is not regular. Thus, we have defined three classes of shifts nested in strict containment:

$$\text{Shifts of finite type} \;\subset\; \text{Sofic shifts} \;\subset\; \text{Coded shifts}$$

2.5. **Properties of shifts and graphs.** The proof strategy detailed in Section 3.1 involves first analyzing the properties of graphs, then applying this knowledge to the shift spaces they represent, and finally constructing a graph for any coded shift with the required properties. In this subsection, we review the most important and relevant properties of the relationship between shifts and graphs; we refer the reader to [4] for a comprehensive exposition.

Let $\mathcal{G}$ be a directed graph labeled by an alphabet $\mathcal{A}$. We say $\mathcal{G}$ is *strongly connected* or *irreducible* if, for every pair of vertices i and j, there exists a path in $\mathcal{G}$ starting from i and ending at j. This provides an alternative characterization for coded shifts: A shift X is a coded shift if it is recognized by a countable strongly connected graph [1, Proposition 2.1].

A graph is *unambiguous* if, for every word w and pair of states p, q, there is at most one path labeled w from p to q. A graph is *strongly unambiguous* if it has at most one bi-infinite path with a given bi-infinite label. We will see that, given the right assumptions, a strongly unambiguous graph corresponds to an unambiguously coded shift. This fact is essential to the proof.

A graph is *deterministic* if no vertex has two outgoing edges with the same label. A graph is *co-deterministic* if its reversal (formed by reversing every directed edge) is deterministic. A graph is *reversible* if it is both.

2.6. **Computability.** We address the challenges of translating the proof described in [4] into an algorithm that can be executed as a Python program. Given that coded shifts may be generated by infinite codes, we apply approximation techniques from computability theory to manage this complexity on a finite machine. Specifically, our algorithm acts as an oracle, which produces arbitrarily large blocks of data as requested, rather than attempting to compute infinite structures directly. This approach allows us to adapt the proof into an executable algorithm while maintaining a connection to the underlying theory. We review the theory and practice surrounding oracles and explore their application to the study of coded shifts.

An *oracle* in computability theory is a theoretical black box that provides solutions to specific computational problems, often decision problems. A mathematical object is *computable* if there exists an algorithm (Turing machine) that approximates that object to arbitrary precision [3]. We analyze this definition as applied to various mathematical objects and situate our algorithm in the context of computability theory.

In the following examples, computability is defined by the existence of an algorithm that can give increasingly precise approximations of a certain mathematical object. See [3] for more details.

Definition (Computability of a real number). *An* oracle approximation of a real number $x \in \mathbb{R}$ *is a function* ψ *such that on input* $n \in \mathbb{N}$, $\psi(n) \in \mathbb{Q}$ *satisfies* $|x - \psi(n)| < 2^{-n}$. *A real number* x *is said to be* computable *if there exists a Turing machine which is an oracle for* x.

Definition (Computability of a set). *Suppose that* $S \subseteq \mathbb{N}$ *is a set of natural numbers. An* oracle approximation of the set S *is a function* ψ *such that on input* $n \in \mathbb{N}$, $\psi(n) = 1$ *if* $n \in S$, *otherwise* $\psi(n) = 0$. *In other words,* ψ *is the characteristic function for* S. *A set* S *is said to be* computable *if there exists a Turing machine which is an oracle for* S.

Definition (Computability of a bi-infinite sequence). *Suppose that* X *is a subshift with finite alphabet* $\mathcal{A}$ *and let* $x \in X$. *An* oracle approximation of x *is a function* ψ *such that on input* $n \in \mathbb{N}, \psi(n)$ *is the word* $x_{[-n,n]}$, *that is, the central block of length* $2n + 1$ *of* x. *A bi-infinite sequence* $x \in X$ *is said to be* computable *if there exists a Turing machine which is an oracle for* x.

Definition (Computability of a subshift). *Suppose that* X *is a subshift with finite alphabet* $\mathcal{A}$. *An* oracle approximation of X *is a function* ψ *such that on input* $n \in \mathbb{N}$, $\psi(n)$ *is a finite list of all admissible words of length* $2n + 1$.

The definitions of computability share a common theme: approximating a complex and possibly infinite structure by finite objects to arbitrary degree. We aim to extend this notion to develop a natural definition of computability for a code. Since a code C is a set of words and is at most countable, it can be

expressed as a sequence $(c_1, c_2, \ldots)$. By treating the code as a sequence, we can extend Turing's definition of sequence computability to codes [11].

Definition (Computability of a code)**.** *Let C be a code over the alphabet $\mathcal{A}$. An* oracle approximation of *C is a function ψ such that for a fixed enumeration $(c_1, c_2, \ldots)$ and input $n \in \mathbb{N}$, $\psi(n)$ returns c_n. In the case that C is finite and $n > |C|$, then $\psi(n)$ returns the empty word.*

3. MAIN RESULT

3.1. **Theoretical construction.**

We outline the construction of an unambiguous code for a coded shift X, following the method of of Béal–Perrin–Restivo (see Theorems 31 and 34 in [4]). Their proof relies on the fundamental connection between a shift space X and its associated graph $\mathcal{G}$. The goal is to construct a certain labeled graph $\mathcal{G}$ which realizes X, and then leverage the properties of $\mathcal{G}$ to show that $X = X(\mathcal{G})$ is unambiguously coded. The unambiguous code can be directly read from the graph $\mathcal{G}$.

First, fix a strongly connected labeled graph $\mathcal{E}$ which realizes X. If X consists of a single periodic orbit, the result is trivial. Otherwise, there exists a vertex q from which there are at least two outgoing distinct finite labeled paths. Let $ya, yb \in \mathcal{L}(X)$ be the labels of these two paths where $a, b \in \mathcal{A}$ are distinct symbols and y is their maximal common prefix (note that y could be the empty word). Similarly, since the graph is strongly connected, there exist distinct symbols $c, d \in \mathcal{A}$ and admissible paths labeled ct and dt that both terminate at q, with $c \neq d$. Here t is a (possibly empty) word in $\mathcal{L}(X)$. Because the graph is strongly connected, there exist connecting words $u_1, u_2 \in \mathcal{L}(X)$ such that the concatenations

$$yau_1ct \quad \text{and} \quad ybu_2dt$$

label admissible paths in the graph. These words represent two loops based at q that branch (via a versus b) and merge again before returning to q (via c versus d). Figure 2 shows selected vertices and edges of the graph $\mathcal{E}$ with distinguished vertex q and additional vertices arranged so that:

- there are two distinct outgoing paths from q labeled ya and yb,
- there are two distinct incoming path to q labeled ct and dt,
- the paths labeled yau_1ct and ybu_2dt form distinct loops based at q.

This configuration is the fundamental branching structure used in [4].

Now consider all finite closed paths in $\mathcal{E}$ that start and end at q. By a *closed path around* q we mean a finite directed path in $\mathcal{E}$ whose initial and terminal vertices are both q. Let $(w'_i)_{i\geq 1}$ be a fixed enumeration of all such closed paths. In general this collection is infinite, and therefore $\mathcal{E}$ (and the graph constructed below) may be infinite. For each i, let w'_i also denote the

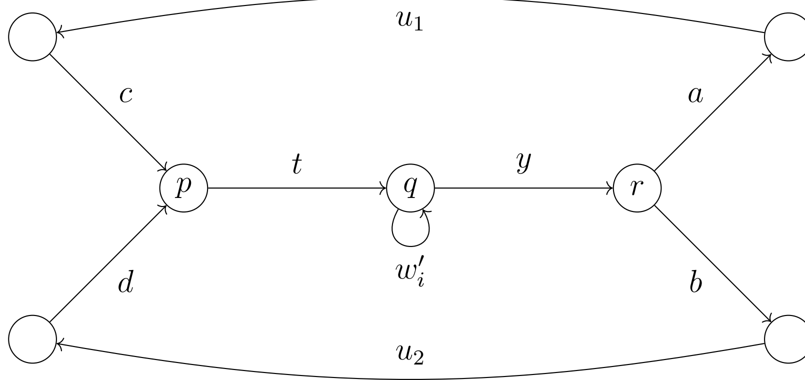

FIGURE 2. Selected vertices and edges of the graph $\mathcal{E}$

label of the corresponding closed path (depicted in Figure 2) and define a padded word

$$w_{-i} = tw'_i y.$$

By construction, each w_{-i} labels a path that starts at p, travels along a loop w'_i, and ends at r. To simplify the notation let $w = ty$, $u = au_1c$, and $v = bu_2d$

Next, choose a strictly increasing sequence of positive integers (m_i) that grows sufficiently rapidly. More precisely, for each i we require

$$|uw|^{m_i-1} \geq 2(i+1)|v| \cdot |w_{-1}w_{-2}...w_{-i}| \cdot |uw|^{m_1+m_2+\cdots+m_{i-1}}.$$

It was shown in [4, Theorem 34] that this growth condition ensures that overlaps between different concatenations are impossible; this is the key combinatorial mechanism guaranteeing strong unambiguity.

With the words defined as above, consider the bi-infinite sequence

$$x = \cdots (uw)^{m_2} vw_{-2}(uw)^{m_1} vw_{-1}u \,.\, (wu)^{\infty},$$

where the blocks are arranged so that each w_{-i} is separated by sufficiently long intervening segments. For reference, x_0 is the first symbol of w, and x_{-1} is the final symbol of u. By construction, the orbit of x is dense in X.

We now define a new graph $\mathcal{G}$ from this sequence. Let $\mathbb{Z}$ be the vertex set, and include an edge from vertex i to vertex $i+1$, labeled x_i, for each $i \in \mathbb{Z}$. This produces a bi-infinite directed graph whose edge labels are given by the symbols of x (see Figure 3).

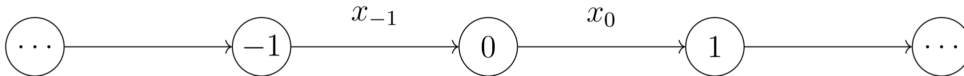

FIGURE 3. Directed graph labeled by the coordinates of x.

Next, we insert additional directed paths that run from right to left, beginning at a vertex labeled by a positive integer and terminating at one labeled by a negative integer. To this end, fix a strictly increasing sequence $(k_i)_{i=1}^{\infty} \subset \mathbb{N}$ with $k_1 = m_1$. Let $N_i < 0$ denote the terminal vertex of the first edge labeled u in the subword $(uw)^{m_i} v w_{-i}$ of x, and let $M_i > 0$ denote the terminal vertex of the path labeled $(wu)^{k_i} w$ that begins at 0. For each $i \in \mathbb{N}$, add a directed path labeled v from the vertex M_i to the vertex N_i (see Figure 4).

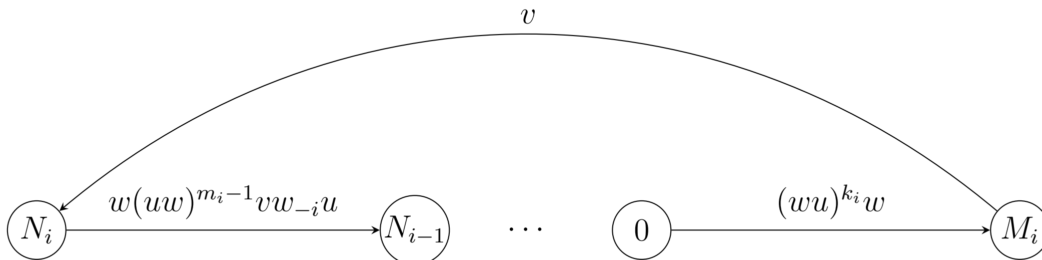

FIGURE 4. Selected vertices and edges of the graph $\mathcal{G}$

The resulting graph $\mathcal{G}$ is realized by X. By examining edge labels, it is straightforward to verify that $\mathcal{G}$ is strongly connected, deterministic, and co-deterministic. This means there is a path from any vertex to any other vertex, and every pair of edges entering or exiting a vertex has distinct labels. It is shown in [4, Theorem 34] that $\mathcal{G}$ is strongly unambiguous, as each bi-infinite path has a unique label. The set of cycles around the vertex 0 gives a prefix code C that unambiguously codes X.

3.2. **Computability of an unambiguous code.** A computer program cannot directly manipulate an infinite code, but it can work with finite lists of arbitrary length. We therefore adopt a standard approach from computability theory and invoke an oracle that supplies finite data in response to user queries. The interaction proceeds as follows:

(1) The user is given a (possibly infinite and possibly ambiguous) code B together with an enumeration (b_i) of B. This role may be fulfilled by a separate algorithm or by an oracle.
(2) The user requests n words of an unambiguous code C such that $X(C) = X(B) = X$.
(3) The program outputs an integer m, indicating how many words from B are required to generate the first n words in an enumeration of C.
(4) The user or an oracle provides the finite list $(b_1, \dots, b_m)$, consisting of the first m words in the enumeration of B.
(5) The program returns the list $(c_1, \dots, c_n)$, consisting of the first n words in the enumeration of C.

This procedure shows that the unambiguous code C is computable. Indeed, given access to an algorithm, oracle, or explicit description of an ambiguous code B generating X, the user can obtain arbitrarily many words of an unambiguous code C with $X = X(C)$.

Formally, if ψ_2 is an oracle approximation generating the sequence (b_i), then we may construct an algorithm ψ_1 which, given oracle access to ψ_2, outputs the first n words of an enumeration of C. While ψ_1 is described here as returning the list $(c_1, \dots, c_n)$, it can be trivially modified to output only c_n. With this modification, ψ_1 is an oracle approximation for the code C in the sense of Section 2.6.

Since ψ_1 queries only finitely many values of ψ_2, it is computable relative to ψ_2. Equivalently, ψ_1 is an oracle Turing machine with oracle ψ_2. This verifies that the unambiguous code C is computable with respect to the code B, and hence computable in the sense of Definition 2.6.

3.3. **Implementation in Python.**

The starting point of the theoretical construction in [4] is to fix a strongly connected labeled graph $\mathcal{F}$ that realizes X. As discussed in Subsection 2.5, such a graph is known to exist. However, in order to implement the program we must construct such a graph using only the data available to us, namely the finitely many words from the code B that the program can query.

We fix a distinguished vertex, denoted by q, and represent each element of B as a labeled cycle that starts and ends at q. This graph plays the role of the graph $\mathcal{F}$. The first two words in the enumeration of B serve as the words yut and yvt in the theoretical construction (Subsection 3.1). The next step in the theory is to enumerate all finite paths that start and end at q. While such an enumeration exists in principle, since there are countably many such paths, the computer has access only to finitely many elements of B. Therefore we must enumerate all concatenations of the available words in a way that remains consistent as additional loops are added. This is implemented in the function `generate_paths` of our program, which is described in detail below.

Next, the elements of the new unambiguous code C are created as concatenations of words, following the procedure outlined in Section 3.1. From the structure of this construction, the program determines how many elements of B are required to produce a prescribed number of elements of the code C. We note that although the code C depends on the chosen enumeration of B, this number does not.

We now provide an overview of the program, following its control flow, which is the order in which its main functions are called (Appendix A).

First, the function `get_user_input` is called (Line 122), prompting the user to input a positive integer n, representing the number of words of an

unambiguous code that the program is requested to produce. The program calculates the number of words m that the user must supply in order for the program to generate n words of an unambiguous code. The user then inputs m words of a known code. We define the array `words = [word_1, word_2, ..., word_m]` of the m words submitted by the user.

Next, the function `build_graph` is called (Line 123), which assigns values to the variables depicted in the graph shown in Figure 2. Only two words from `blocks` are required to define a set of edge labels according to the requirements given.

Next, the function `generate_paths` is called (Line 124) and returns an enumeration of labels for all finite closed paths around the vertex q. Each word $b_i \in B$ is the label of a closed path around q. Consequently, any concatenation of words $w = b_1 b_2 \dots b_k \in B^*$ is also the label a closed path.

To enumerate B^*, we define an order on the set of all finite sequences of non-negative integers $\mathbb{N}^*$, where $*$ is the Kleene star operation. This order on $\mathbb{N}^*$ induces an order on C^* where each word $a_i \in C$ is identified with its index. Defining this order requires careful consideration. A word can be concatenated with itself multiple times before adding a second word. The order is defined as follows:

Let $\mathbb{N}^*$ be the set of finite lists of non-negative integers. For a list $x = a_1 a_2 \dots a_k \in \mathbb{N}^*$, define $h(x) = a_1 + \dots + a_k + k$. For $x, y \in \mathbb{N}^*$, define

$$x < y \quad \text{if} \quad \begin{cases} h(x) < h(y), \text{ or} \\ h(x) = h(y) \text{ and } x \text{ is less than } y \text{ in lexicographic order.} \end{cases}$$

The first 10 elements of $\mathbb{N}^*$ under this ordering are listed below. For readability, lists are joined into strings; for example, $(0, 1, 0)$ is written 010.

$$\mathbb{N}^* = \{\, \emptyset,\ 0,\ 00,\ 1,\ 000,\ 01,\ 10,\ 2,\ 0000,\ 001, \dots\}.$$

This diagonalization over lengths and values ensures that we have a bijection $\mathbb{N} \to \mathbb{N}^*$. Applied to a list which begins `blocks = [cat, dog, pig, ...]`, the ordering gives the first eight elements as follows:

blocks* = `[∅, cat, catcat, dog, catcatcat, catdog, dogcat, pig, ...]`

The choice of enumeration determines how the value of m is calculated from n. Under our enumeration, an input list $(b_1, \dots, b_m)$ of length m can generate an output list $(c_1, \dots, c_n)$ with up to $n = 2^m$ words.

When the user queries more words from the algorithm, it returns a longer list that includes all previously provided words along with the new words. This process can continue indefinitely, with each new output preserving the order

and content of earlier, shorter outputs, even though the complete list may be infinite and never fully generated.

Next, the function `generate_integers` (Line 125) calculates a list of non-negative integers m_i, used to determine $(uw)^{m_i}$ — the repeated concatenations of the block uw in each successive code word. The program calculates only as many increasing integers as is needed to return the list $\{c_i\}_{i=1}^n$.

Finally, the function `generate_code` (Line 126) follows the rules outlined in Section 3.1 and uses the previously calculated outputs to return a list containing the first n words of an unambiguous code for $X = X(B)$. The program then terminates.

3.4. **Example.** While the theoretical construction is fully defined in its mathematical description, implementing and running the algorithm on a computer provides valuable practical insights. To illustrate this, we present an example (see Appendix B) that demonstrates how the algorithm behaves when given real data, giving readers a concrete understanding of its operation. We take for an example a Dyck shift, defined as follows. Let $\mathcal{A} = \{\texttt{(,),[,]}\}$ be an alphabet. Let D be the set of words on $\mathcal{A}$ that reduce to the identity under the relations $\texttt{()} = \texttt{[]} = \text{id}$, and further, which cannot be written as a concatenation of words that reduce to the identity, i.e. '`()[]`' $\notin D$ [9]. This requirement can be understood as describing strings of balanced parentheses, where D is a set of blocks in which all parentheses open and close correctly, and are not themselves concatenations of other words in D.

We define the Dyck shift X as the shift generated by the code D. In fact, D unambiguously codes X. The language $\mathcal{L}(X)$ is not regular, so X is not sofic. However, D is context-free, meaning it can be described by a context-free grammar:

$$S \ \rightarrow \ \epsilon \mid \texttt{(}S\texttt{)} \ \mid \texttt{[}S\texttt{]}$$

This context-free grammar, when applied systematically, produces the canonical generating set for the Dyck shift X, which begins

$$D = \{\texttt{(),[],(()),([]),}\ldots\}.$$

Thus $X = X(D)$ is a non-sofic, transitive coded shift. To demonstrate the program's functionality, we input a modified, intentionally ambiguous code B that also generates X. The program then handles the ambiguity and returns the initial words of a code that unambiguously generates the Dyck shift X. Specifically, if we define $B = D \cup \{\texttt{()[]}\}$ then B is an ambiguous code, yet $X(B) = X(D)$. In the program, we enter the 3 words `(), [], ()[]` after requesting 8 output words to show the program's handling of an ambiguous input set (Appendix B: Lines 1-5).

The program then constructs a representation of the graph $\mathcal{F}$ (see Figure 2) with edge labels based on the user's input words according to the rules detailed

in Section 3.1. The assignment of each edge is shown (Appendix B: Lines 11-21). The enumeration described in Section 3.3 is used to create the labels of the first eight cycles $w_1, \ldots, w_8$ which are then used to form the first eight words of the unambiguous code C (Appendix B: Lines 28 - 37). The algorithm requires an increasing sequence of integers with specific constraints. Minimal values for these integers are calculated and shown (Appendix B: Line 40.)

A notable feature of the program's output is the significant increase in length of each successive returned word. With input words of length 2, 2, and 4, the first produced codeword is 98 characters, while the eighth codeword balloons to a staggering 18,562 characters. This increase is due to the sequence of integers that determine how often the block uw is repeated in each codeword. Although the algorithm determines the minimal feasible integer, the eighth codeword in our example still contains 580 copies of the block `w = ()()` (Appendix B: Line 67).

This lengthening may not be immediately obvious in reading [4]. However, this feature becomes strikingly apparent after running this program, which sheds light on the algorithm's theoretical value over its practical use. Future research might explore algorithms that find more useful unambiguous generating sets with shorter code words, optimizing for practicality and effectiveness.

APPENDIX A: MAIN PROGRAM CODE

```
import math

def print_dict(graph):
    """
    Input:
        graph (dict): A dictionary representing selected
            components of the graph F.
    Output:
        None (prints to stdout).
    Purpose:
        Pretty-print the key-value pairs of the graph
            dictionary with aligned spacing.
    """
    for key, value in graph.items():
        space = ' ' if len(str(key)) == 1 else ''
        print(f'  {key}:{space} {value}')
    print()


def print_enumeration(words, graph):
    """
    Input:
```

```
        words (dict): A dictionary enumerating padded words.
        graph (dict): The graph dictionary containing prefix/
            suffix data.
    Output:
        None (prints to stdout).
    Purpose:
        Display the enumeration of B* after removing the
            padding words t and y.
    """
    print('Enumeration of B*:')
    for i, word in words.items():
        middle_word = word[len(graph['t']):-len(graph['y'])]
        print(f'  Word {i}: {middle_word}')


def print_cycles(graph):
    """
    Input:
        graph (dict): The graph dictionary containing cycles
            and branches.
    Output:
        None (prints to stdout).
    Purpose:
        Display the two fundamental cycles in the graph F,
            aligned for readability.
    """
    u1 = graph['u1']
    u2 = graph['u2']
    diff = len(u1) - len(u2)

    # Pad the shorter word so both cycles align visually
    if diff > 0:
        u2 += ' ' * diff
    elif diff < 0:
        u1 += ' ' * (-diff)

    print('\nCycles in the graph F:')
    print('  y a u1 c t : ' + graph['y'] + ' ' + graph['a'] + '
        ' + str(u1) + ' ' + graph['c'] + ' ' + graph['t'])
    print('  y b u2 d t : ' + graph['y'] + ' ' + graph['b'] + '
        ' + str(u2) + ' ' + graph['d'] + ' ' + graph['t'])
    print()


def get_user_input():
```

```
60      """
        Input:
            None (interactive user input).
        Output:
            words (list of str): A list of user-provided code words
               .
            requested (int): Number of unambiguous words requested.
        Purpose:
            Collect user input and enforce a minimum number of code
                 words required
            for the construction, based on log2 of the requested
               output size.
        """
70      requested = int(input('Enter number of words requested: '))
        m = max(math.floor(math.log2(requested)), 2)

        print(f'\nTo receive the first {requested} words of the
           unambiguous code, please enter {m} words.')

        while True:
            words = input('\nEnter strings separated by commas: ')
            words = [s.strip() for s in words.split(',') if s.strip
               ()]
            if len(words) >= m:
                return words, requested
80          print(f'\nIncorrect length. Please enter at least {m}
               words.')


    def first_difference(string_1, string_2):
        """
        Input:
            string_1, string_2 (str): Two strings to compare.
        Output:
            int or False: Index of the first differing character,
               or False if none exists.
        Purpose:
90          Identify the first position at which two strings differ
               .
        """
        for i, (char_1, char_2) in enumerate(zip(string_1, string_2
           )):
            if char_1 != char_2:
                return i
        return False
```

```
def build_fork(word_a, word_b):
    """
100 Input:
        word_a, word_b (str): Two words from the (possibly
            ambiguous) generating set.
    Output:
        dict or False: A dictionary describing a fork structure
            , or False if none exists.
    Purpose:
        Construct a prefix fork between two words, identifying
            a common trunk
        and divergent branches.
    """
    a, b = word_a, word_b

110 # Extend words until they differ in a non-prefix manner
    while a != b:
        if a.startswith(b):
            b += word_b
        elif b.startswith(a):
            a += word_a
        else:
            i = first_difference(a, b)
            return {
                'trunk': a[:i] if a[:i] else word_a,
120             'branch_1': a[i],
                'branch_2': b[i],
                'u1': a[i + 1:],
                'u2': b[i + 1:]
            }
    return False


def build_graph(a, b):
    """
130 Input:
        a, b (str): Two words used to construct the graph.
    Output:
        dict or False: A dictionary encoding the graph F, or
            False if construction fails.
    Purpose:
        Construct the full graph F by combining forward and
            reversed fork structures.
```

```
    """
    fork_dict = build_fork(a, b)
    main_dict = {key: value[::-1] for key, value in build_fork(
        a[::-1], b[::-1]).items()}

    if not (fork_dict and main_dict):
        return False

    # Construct internal paths
    u_1 = fork_dict['u1'] + main_dict['u1'] or a
    u_2 = fork_dict['u2'] + main_dict['u2'] or a

    return {
        'y': fork_dict['trunk'],
        'a': fork_dict['branch_1'],
        'b': fork_dict['branch_2'],
        't': main_dict['trunk'],
        'c': main_dict['branch_1'],
        'd': main_dict['branch_2'],
        'u1': u_1,
        'u2': u_2,
        'u': fork_dict['branch_1'] + u_1 + main_dict['branch_1
            '],
        'v': fork_dict['branch_2'] + u_2 + main_dict['branch_2
            '],
        'w': main_dict['trunk'] + fork_dict['trunk']
    }


def generate_paths(words, graph, words_requested):
    """
    Input:
        words (list of str): Original code words.
        graph (dict): Graph dictionary containing padding
            symbols.
        words_requested (int): Number of words to generate.
    Output:
        dict: An enumeration of padded words representing paths
            .
    Purpose:
        Enumerate concatenations of code words and pad them
            with t and y.
    """
    number_list = [[0]]
```

```
    # Generate index sequences
    for i in range(words_requested - 1):
        if len(number_list[-1]) == 1:
            number_list += [[0] * (number_list[-1][0] + 2)]
        else:
180         last = number_list[-1]
            next = last[:-2] + [last[-2] + 1] + [0] * last[-1]
            number_list.append(next)

    # Build the words corresponding to index sequences
    word_list = [''.join(words[i] for i in indices) if indices
       else '' for indices in number_list]

    # Pad with trunk and tail
    padded_list = [graph['t'] + graph['y']]
    padded_list += [graph['t'] + s + graph['y'] for s in
       word_list]
190
    return dict(enumerate(padded_list))


def generate_integers(graph, w):
    """
    Input:
        graph (dict): Graph dictionary.
        w (dict): Enumerated padded words.
    Output:
200     list of int: A sequence of integers controlling
           repetition lengths.
    Purpose:
        Compute repetition counts ensuring disjointness of code
            words.
    """
    m, s = [0], [0]
    s.append(len(graph['v'] + w[1] + graph['v']))
    uw = graph['u'] + w[0]

    for i in range(len(w) - 2):
        m += [math.ceil(2 * s[-1] / len(uw)) + 1]
210     s.append(len(graph['v'] + w[i + 2] + uw * m[i + 1]))

    m += [math.ceil(2 * s[-1] / len(uw)) + 1]
    return m
```

```
def generate_code(graph, w, m):
    """
    Input:
        graph (dict): Graph dictionary.
220     w (dict): Enumerated padded words.
        m (list of int): Integer repetition parameters.
    Output:
        list of str: The resulting unambiguous code words.
    Purpose:
        Construct the unambiguous code using controlled
           concatenations
        of graph paths.
    """
    u, v = graph['u'], graph['v']
    s = ' '
230 words, ends = [], []

    for i in range(len(m) - 1):
        block_start = (w[0] + s + u) * (m[1] + i) + s + w[0] +
           s + v
        ends += [s + w[0] + s + (u + s + w[0]) * (m[i + 1] - 1)
            + s + v + s + w[i + 1] + s + u]
        block_end = ''.join(ends[::-1])
        words += [block_start + block_end]

    return words

240
# ---------------- Main execution ----------------

words, number = get_user_input()
graph = build_graph(words[0], words[1])
w = generate_paths(words, graph, number)
m = generate_integers(graph, w)
c = generate_code(graph, w, m)

print(f'\nThe user has provided {len(words)} words of a (
   possibly ambiguous) code.'
250   f'\nThe program will return {number} words of an
         unambiguous code.'
      f'\nThe following data are computed for the calculation.\
         n')

print('Selected edges of the graph F:')
print_dict(graph)
```

```
print_cycles(graph)
print_enumeration(w, graph)

print('\nIncreasing sequence of integers:')
print(f'  {m} \n')

for k in range(len(c)):
    print(f'\nWord {k + 1}:\n\n{c[k]}\n')
```

APPENDIX B: EXAMPLE

```
Enter number of words requested: 8

To receive the first 8 words of the unambiguous code, please
enter 3 words.

Enter strings separated by commas: (),[],()[]

The user has provided 3 words of a (possibly ambiguous) code.
The program will return 8 words of an unambiguous code.
The following data are computed for the calculation.

Selected edges of the graph F:
  y:  ()
  a:  (
  b:  [
  t:  ()
  c:  )
  d:  ]
  u1: )(
  u2: ][
  u:  ()()
  v:  [][]


Cycles in the graph F:
  y a u1 c t : () ( )( ) ()
  y b u2 d t : () [ ][ ] ()

Enumeration of B*:
  Word 0:
  Word 1: ()
  Word 2: ()()
  Word 3: []
```

```
  Word 4: ()()()
  Word 5: ()[]
  Word 6: []()
  Word 7: ()[]
  Word 8: ()()()()

Increasing sequence of integers:
[0, 5, 14, 32, 69, 142, 288, 580, 1165]


Word 1:

  ()() ()()()() ()()()() ()()()() ()()()() ()() ()() [][] ()()
  ()() ()()()() ()()()() ()()()() ()() [][] ()()() ()()


Word 2:

  ()() ()()()() ()()()() ()()()() ()()()() ()()()() ()() ()()
  [][] ()() ()() ()()()() ()()()() ()()()() ()()()() ()()()()
  ()()()() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()
  () ()() [][] ()()()() ()() ()() ()() ()()()() ()()()() ()()()
  () ()() [][] ()()() ()()


Word 3:

  ()() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()() ()
  () ()() [][] ()() ()() ()()()() ()()()() ()()()() ()()()() ()
  ()()() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()()
  ()()()() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()
  () ()()()() ()()()() ()()()() ()()()() ()()()() ()()()() ()()
  ()() ()()()() ()()()() ()()()() ()()()() ()()()() ()() [][]
  ()[]() ()() ()() ()() ()()()() ()()()() ()()()() ()()()() ()
  ()()() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()()
  ()()()() ()() [][] ()()()() ()() ()() ()() ()()()() ()()()()
  ()()()() ()() [][] ()()() ()()


...
```

```
Word 8:

()() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()() ()()
()() ()()()() ()()()() ()()()() ()()()() ()() ()() [][] ()() ()
() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()
() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()
() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()
() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()
() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()
() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()
() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()
() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()
() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()
() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()
() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()
() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()
() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()
() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()
() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()
() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()
() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()
() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()
() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()
() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()
() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()
() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()
() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()
() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()
() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()
() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()
() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()
() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()
() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()
() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()
() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()
() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()() ()()()
() ()()()() ()() ...

                  ###  Total of 18,562 characters  ###

70 ... () ()()()() ()()()() ()()()() ()()()() ()()()() ()()()() ()
()()() ()() [][] ()()()() ()() ()() ()() ()()()() ()()()() ()()
()() ()() [][] ()()() ()()
```

REFERENCES

[1] F. Blanchard and G. Hansel, *Systèmes codés,* Theoretical Computer Science **44** (1986), 17–49.

[2] M. Burr, S. Das, C. Wolf, and Y. Yang, *Computability of topological pressure on compact shift spaces beyond finite type*, Nonlinearity **35** (2022) 4250.

[3] M. Burr and C. Wolf, *Computability in dynamical systems. Recent De- velopments in Fractal Geometry and Dynamical Systems*, Contemporary Mathematics, 797:85 **98**, 2024.

[4] M. Béal, D. Perrin, and A. Restivo, *Unambiguously coded systems*, European Journal of Combinatorics **119** (2021) 103812.

[5] V. Climenhaga, *Specification and Towers in Shift Spaces*, Communications in Mathematical Physics, **364(2)** (2018) 441–504.

[6] D. Fiebig and U. Fiebig, *Covers for coded systems,* In Peter Walters, editor, Symbolic Dynamics and its Applications, volume 136 of Contemporary Mathematics, American Mathematical Society, 1992.

[7] T. Kucherenko, M. Schmoll, and C. Wolf, *Ergodic theory on coded shift spaces*, Advances in Mathematics, 457:109913, 2024.

[8] D. Lind and B. Marcus, *An Introduction to Symbolic Dynamics and Coding*, Cambridge University Press, New York, NY, 1989.

[9] R. Pavlov, *On entropy and intrinsic ergodicity of coded shifts*, Proceedings of the American Mathematical Society, **148** (2022), 4717–4731.

[10] A. Sardinas and G. Patterson, *A necessary and sufficient condition for the unique decomposition of coded messages. In Convention Record of the I.R.E.*, 1953 National Convention, Part 8: Information Theory, pages 104 **108**, 1953.

[11] A. Turing, *On computable numbers, with an application to the entscheidungsproblem*, Proceedings of the London Mathematical Society, s2-42(1):230 **265**, 1936. 18

Department of Mathematics, The City College of New York, New York, NY, 10031, USA

Email address: `tkucherenko@ccny.cuny.edu`

Department of Mathematics, The City College of New York, New York, NY, 10031, USA

Email address: `btupper000@citymail.cuny.edu`