Periods and Binary Partial Words*

F. Blanchet-Sadri1        Brian Shirey2

Abstract

A well known and unexpected result of Guibas and Odlyzko states that the set of all periods of a word is independent of the alphabet size (alphabets with one symbol are excluded here). Recently, Blanchet-Sadri and Chriscoe extended this fundamental result to words with one “do not know” symbol also called partial words with one hole. They showed that for every partial word $u$ with one hole, there exists a partial word $v$ over the alphabet $\{0, 1\}$ sharing the same length and same sets of periods and weak periods as $u$, and satisfying $H(u) \supset H(v)$ where $H(u), H(v)$ denote the sets of holes of $u, v$. In this paper, we provide an algorithm that given a nonspecial partial word $u$ with an arbitrary number of holes, computes a partial word $v$ as described. A World Wide Web server interface at www.uncg.edu/cmp/research/bintwo has been established for automated use of the program.

1 Introduction

Words, sequences or strings of symbols from a finite alphabet, arise naturally in several research areas. Notions and techniques related to periodic structures in words find applications in virtually every area of theoretical and applied computer science, notably in coding [2], computational biology [8], data compression [12], string searching and pattern matching algorithm design [6]. In pattern matching, for instance, several algorithms take advantage of the periods of the pattern to speed up the search of its occurrences in a text. Periods in strings have played over the years a central role in the development of combinatorics on words, and Reference [10] contains a systematic and self-contained exposition of this theory including significant results such as a theorem of Guibas and Odlyzko which shows that the set of all periods of a word is independent of the alphabet size [7].

In many practical applications, repetitions admit a certain variation between copies of the repeated pattern because of errors due to experiments, etc. Approximate repeated patterns, or repetitions where errors are allowed, are playing a central role in different variants of string searching and pattern matching problems [11]. Partial words, or strings that may have a number of “do not know” symbols, also called “holes” or “⋄”s, have acquired great importance in this context. Berstel and Boasson introduced partial words [1], and several of their combinatorial properties have been investigated since then [3]. In particular, Blanchet-Sadri and Chriscoe

---

*This material is based upon work supported by the National Science Foundation under Grants CCF–0207673 and DMS–0452020.

1Department of Computer Science, University of North Carolina, P.O. Box 26170, Greensboro, NC 27402–6170, USA, blanchet@uncg.edu. A research assignment from the University of North Carolina at Greensboro is gratefully acknowledged. Some of this assignment was spent at the LIAFA: Laboratoire d’Informatique Algorithmique: Fondements et Applications of Université Paris 7, Paris, France.

2Department of Computer Science, NC State University, Campus Box 8206, Raleigh, NC 27695–8206, USA. A research assistantship from the University of North Carolina at Greensboro is gratefully acknowledged.
proved a theorem analogous to the one of Guibas and Odlyzko for the one-hole case. After them, Blanchet-Sadri, Gafni and Wilson extended this result to arbitrary partial words showing that every partial word has a “binary equivalent” [5]. Phrased differently, if \( u \) is a partial word over an alphabet \( A \), then there exists a partial word \( v \) over the alphabet \( \{0, 1\} \) such that (1) \(|v| = |u|\), (2) \( \mathcal{P}(v) = \mathcal{P}(u) \), and (3) \( \mathcal{P}'(v) = \mathcal{P}'(u) \) (here \(|u|, |v|\) denote the lengths of \( u \), \( v \), \( \mathcal{P}(u), \mathcal{P}(v) \) the sets of strong periods of \( u, v \), and \( \mathcal{P}'(u), \mathcal{P}'(v) \) the sets of weak periods of \( u, v \)).

In [9], Halava, Harju and Ilie gave a simple constructive proof of the theorem of Guibas and Odlyzko which computes \( v \) in linear time. The one-hole case was proved by extending Halava et al.’s approach [4]. More specifically, given a partial word \( u \) with one hole over an alphabet \( A \), a partial word \( v \) over \( \{0, 1\} \) exists such that Conditions (1)–(3) hold as well as the following condition (4) \( H(u) \supset H(v) \) where \( H(u), H(v) \) denote the sets of holes of \( u, v \). However, Conditions (1)–(4) cannot be satisfied simultaneously in the two-hole case. For the partial word \( \text{abaca}\diamond\text{acaba} \) can be checked by brute force to have no such binary equivalent (although it has the binary equivalent \( 01111\diamond111\diamond1\diamond1\diamond \)). In this paper, we design an algorithm for computing a binary equivalent satisfying Conditions (1)–(4) when such equivalent exists.

## 2 Preliminaries

This section is devoted to fix notations, and to review definitions and elementary properties of words and partial words. Let \( A \) be an alphabet, or a finite set of symbols or characters. A sequence of \( n \) symbols of \( A \) indexed from 0 to \( n - 1 \) is called a string or word of length \( n \) over \( A \). We denote the length of the word \( u = a_0a_1\ldots a_{n-1} \) by \(|u|\). For any \( 0 \leq i \leq j < n \), \( u[i..j] = a_i \ldots a_j \) is called a factor or substring of \( u \) (we also denote \( u[i..j] \) by \( u[i..j+1] \)). Moreover, \( u[0..j] \) is a prefix and \( u[i..n-1] \) is a suffix of \( u \). We denote by \( A^* \) (respectively, \( A^n \)) the set of all finite words (respectively, all words of length \( n \)) over \( A \). The unique sequence of length 0, denoted by \( \varepsilon \), is called the empty word. We denote the concatenation of two words \( u \) and \( v \) by \( uv \), and the \( i \)-power of \( u \) or the \( i \)-fold concatenation of \( u \) with itself by \( u^i \).

A word \( u = a_0a_1\ldots a_{n-1} \) of length \( n \) over \( A \) can be defined by the total function \( u : \{0, \ldots, n - 1\} \rightarrow A \) where \( u(i) = a_i \). A partial word \( u \) of length \( n \) over \( A \) is a partial function \( u : \{0, \ldots, n - 1\} \rightarrow A \). For \( 0 \leq i < n \), if \( u(i) \) is defined, then we say that \( i \) belongs to the domain of \( u \) (denoted by \( i \in D(u) \)), otherwise we say that \( i \) belongs to the set of holes of \( u \) (denoted by \( i \in H(u) \)). A word over \( A \) is a partial word over \( A \) with an empty set of holes (we sometimes refer to words as full words). If \( u \) is a partial word of length \( n \) over \( A \), then the companion of \( u \) (denoted by \( u_* \)) is the total function \( u_* : \{0, \ldots, n - 1\} \rightarrow A \cup \{\diamond\} \) defined by \( u_*(i) = u(i) \) if \( i \in D(u) \), and \( u_*(i) = \diamond \) otherwise. The bijectivity of the map \( u \mapsto u_* \) allows us to define for partial words concepts such as concatenation, factor, prefix, suffix, \( i \)-power, etc. in a trivial way. For convenience in the sequel, we consider a partial word over \( A \) as a word over the augmented alphabet \( A \cup \{\diamond\} \), where the additional symbol \( \diamond \) is viewed as a “do not know” symbol. Thus, we say for instance “the partial word \( \text{oabob} \) instead of “the partial word with companion \( \text{oabob} \)”. The partial word \( u = \text{abbobobcb} \) has length 9, \( D(u) = \{0, 1, 2, 4, 6, 7, 8\} \) and \( H(u) = \{3, 5\} \).

An integer \( p \) satisfying \( 1 \leq p \leq n \) is a (strong) period of a partial word \( u \) of length \( n \) over \( A \) if \( u(i) = u(j) \) whenever \( i, j \in D(u) \) and \( i \equiv j \mod p \). In such a case, we call \( u \) (strongly) \( p \)-periodic. Similarly, an integer \( p \) satisfying \( 1 \leq p \leq n \) is a weak period of \( u \) if \( u(i) = u(i + p) \) whenever \( i, i + p \in D(u) \). In such a case, we call \( u \) weakly \( p \)-periodic. The partial word \( \text{abbobobcb} \) is weakly 3-periodic but is not 3-periodic. The latter shows a difference between partial words and words since every weakly \( p \)-periodic full word is strongly \( p \)-periodic. We denote the set of all periods of \( u \) by \( \mathcal{P}(u) \) and the set of all weak periods of \( u \) by \( \mathcal{P}'(u) \).
If \( u \) and \( v \) are two partial words of equal length, then \( u \) is said to be contained in \( v \), denoted by \( u \subset v \), if all elements in \( D(u) \) are in \( D(v) \) and \( u(i) = v(i) \) for all \( i \in D(u) \). We say that the greatest lower bound of a pair of partial words \( u \) and \( v \) of length \( n \) is the partial word \( u \land v \) with \( D(u \land v) = \{0 \leq i < n \mid i \in D(u) \cap D(v) \} \) and \( u(i) = v(i) \) for all \( i \in D(u \land v) \). Note that \( u \land v \) is constructed so that \( (u \land v) \subset u \) and \( (u \land v) \subset v \). Moreover, it is easily seen that \( u \land v \) is maximal in the sense that for all partial words \( w \) which satisfy \( w \subset u \) and \( w \subset v \) we have that \( w \subset (u \land v) \). One property we notice immediately about the greatest lower bound is the fact that if \( u, v \in A_n^\ast \), then \( \mathcal{P}(u) \cup \mathcal{P}(v) \subset \mathcal{P}(u \land v) \) and \( \mathcal{P}'(u) \cup \mathcal{P}'(v) \subset \mathcal{P}'(u \land v) \). The partial words \( u \) and \( v \) are called compatible, denoted by \( u \uparrow v \), if there exists a partial word \( w \) such that \( u \subset w \) and \( v \subset w \). As an example, \( u = abaa\cdots a \) and \( v = a\cdots b\cdots a \) are compatible.

### 3 Extension of Guibas and Odlyzko’s periodicity result

In this section, we first state Guibas and Odlyzko’s periodicity result on words as well as its extension to partial words.

**Theorem 1** ([7]). For every word \( u \) over an alphabet \( A \), there exists a word \( v \) over the alphabet \( \{0, 1\} \) such that \( |v| = |u| \) and \( \mathcal{P}(v) = \mathcal{P}(u) \).

**Theorem 2** ([4]). For every partial word \( u \) with one hole over an alphabet \( A \), there exists a partial word \( v \) over the alphabet \( \{0, 1\} \) such that (1) \( |v| = |u| \), (2) \( \mathcal{P}(v) = \mathcal{P}(u) \), (3) \( \mathcal{P}'(v) = \mathcal{P}'(u) \), and (4) \( H(v) \subset H(u) \). Moreover, \( v \) can be computed optimally in linear time.

The proof given by Guibas and Odlyzko of Theorem 1 uses properties of correlations and is somewhat complicated. In [9], Halava, Harju and Ilie give a simple constructive proof for this result, and as a consequence, they describe an algorithm which computes \( v \) in linear time. The first step in proving Theorem 2 was to extend to partial words with one hole the properties of words and periods needed in Halava et al.’s proof. Those properties of words relate to primitivity testing and to the well known Fine and Wilf’s periodicity result. In [5], using a completely different approach, Blanchet-Sadri, Gafni and Wilson extend Guibas and Odlyzko’s result further to all partial words showing that every partial word has a “binary equivalent” satisfying Conditions (1)-(3).

**Theorem 3** ([5]). For every partial word \( u \) over an alphabet \( A \), there exists a partial word \( v \) over the alphabet \( \{0, 1\} \) such that Conditions (1)-(3) hold.

The proof, which provides an algorithm to compute a binary equivalent, is based on the following construction: For \( n \geq 3 \) and \( p < n \), let \( n = kp + r \) where \( 0 \leq r < p \). Then define

\[
\omega_p = \begin{cases} 
(01^{p-1})^k & \text{if } r = 0 \\
(01^{p-1})^k01^{r-1} & \text{if } r > 0 
\end{cases}
\]

\[
\psi_p = 01^{p-1} \cdot 1^{n-p-1}
\]

Then, given a partial word \( u \) of length \( n \), the partial word

\[
v = \left( \bigwedge_{p \in \mathcal{P}(u)} \omega_p \right) \land \left( \bigwedge_{p \in \mathcal{P}'(u) \setminus \mathcal{P}(u)} \psi_p \right)
\]

satisfies Conditions (1)-(3). For example, given \( abaca\cdots acaba \) which has strong periods 9, 11 (and 12) and strictly weak period 5, a binary equivalent is computed in the following figure:
\[ \begin{align*}
\omega_9 & = 011111111011 \\
\omega_{11} & = 011111111110 \\
\psi_5 & = 0111111 \diamond 11111111 \\
\end{align*} \]

However, Conditions (1)–(4) cannot be satisfied simultaneously in the two-hole case. For the partial word \textit{abacacabacoacaba} can be checked by brute force to have no such binary equivalent (although it has the binary equivalent 01111011 as computed above). The partial words that do not have an equivalent over the binary alphabet \{0, 1\} satisfying Conditions (1)–(4) will be called \textit{special}.

In this paper, we design an algorithm for computing a binary equivalent satisfying Conditions (1)–(4) when such equivalent exists. Our algorithm also recognizes the special partial words. The basic steps of our algorithm are:

- **Collect period/weak period data** The algorithm starts with the input partial word \( u \) over augmented alphabet \( A \cup \{ \diamond_1, \ldots, \diamond_N \} \) where \( \diamond_i \) is the symbol/character for \( i \)th hole of \( u \). Then, it computes \( \mathcal{P}(u) \) and \( \mathcal{P}'(u) \) and sorts their elements in ascending order, that is, \( \mathcal{P}(u) = \{ p_1, p_2, \ldots, p_n \} \) where \( p_1 < p_2 < \cdots < p_n = |u| \) and \( \mathcal{P}'(u) = \{ p'_1, p'_2, \ldots, p'_m \} \) where \( p'_1 < p'_2 < \cdots < p'_m = |u| \).

- **Normalize input** This step, which is described in Section 4, involves two major steps: one relates to periods and the other to weak periods.

- **Rule-Tree** This step, which is described in Section 5, consists of two major steps: generate rules and construct tree (assembly and insertion of rules).

- **Traverse/Convert** The two steps of tree traversal and binary conversion using the coloring of assignment graphs are described in Section 6.

### 4 Normalization routine

In this section, we describe the normalization routine.

**Routine Normalization**

The routine starts with input \( u \) and initializes \( u' \). For \( 0 \leq i < |u| \), \( u'(i) \) is initialized with a symbol \( a_i \) over an alphabet where it is assumed that \( a_j \neq a_k \) for all \( 0 \leq j < k < |u| \). Holes are exempt from being assigned a symbol (a hole remains a hole: either \( \diamond_1 \) (first hole) or \( \diamond_2 \) (second hole) or \( \ldots \)). The routine then consists of two steps.

**Step 1:** For each \( p_i \in \mathcal{P}(u) \setminus \{p_n\} \), represent \( u' \) as a 2-dimensional structure where symbols are aligned into \( p_i \) columns. All entries in any given column need to be assigned a selected symbol. This is done by using the period alignment guidelines described below.

**Step 2:** For each \( p'_i \in \mathcal{P}'(u) \setminus \mathcal{P}(u) \), represent \( u' \) as a 2-dimensional structure where symbols are aligned into \( p'_i \) columns. All entries in any given column need to be assigned a selected symbol based on the weak period alignment guidelines also described below.

Upon exiting the routine, input \( u \) over augmented alphabet \( A \cup \{ \diamond_1, \ldots, \diamond_N \} \) has been transformed into \( u' \), a string over alphabet \( A' \cup \{ \diamond_1, \ldots, \diamond_N \} \). The string \( u' \) has same length as \( u \) and is such that \( \mathcal{P}(u') = \mathcal{P}(u) \) and \( \mathcal{P}'(u') = \mathcal{P}'(u) \).

We first describe the general alignment guidelines. It is assumed that \( u' \) has been initialized and that the symbols in \( u' \) have been aligned into a certain number of columns as in Step 1 or
Step 2. To illustrate our ideas, if \( u \) is a full word of length 8 where \( \mathcal{P}(u) = \{5, 7, 8\} \), then we start by initializing \( u' \) with \( abcdefgh \) say, and we align the symbols in \( u' \) into five columns:

\[
\begin{array}{c|c|c|c|c}
  C_0 & C_1 & C_2 & C_3 & C_4 \\
  a & b & c & d & e \\
  f & g & h \\
\end{array}
\]

A system for flagging positions is needed to keep track of which ones have been assigned a symbol. For our purposes, we will underline these positions to represent this assignment. We assume that \( u'(0) \) is flagged to begin with. There are two situations which arise: either the position, say \( i \), has been flagged or it has not been flagged (it is assumed that \( i \) is in a column with other positions than \( i \)).

**Not flagged:** Assign the position \( i \), which is not flagged as exemplified by \( u'(i) = a \), the selected symbol say \( b \), then underline the symbol \( b \) to show that the position \( i \) is now flagged (now \( u'(i) = b \)). Referring to \( C_0 \), say the selected symbol is \( a \). Here \( a \leftarrow a \) and \( f \leftarrow a \) (the underlines show that the positions 0 and 5 have now been flagged). We proceed similarly for \( C_1 \) and \( C_2 \) where \( b \) and \( c \) are the selected symbols respectively, and we obtain \( abcedabc \).

**Flagged:** Assign the position \( i \), which is currently flagged as exemplified by say \( u'(i) = a \), the new selected symbol say \( b \). Then search the word \( u' \) for other \( a \)'s, and re-assign the corresponding positions \( b \)'s (now, if \( j \) is any position such that \( u'(j) = a \), then \( u'(j) \) becomes \( b \)). For example, say that the positions of the new \( u' \) are aligned into seven columns:

\[
\begin{array}{c|c|c|c|c|c|c}
  C_0 & C_1 & C_2 & C_3 & C_4 & C_5 & C_6 \\
  a & b & c & d & e & a & b \\
  c \\
\end{array}
\]

Referring to \( C_0 \) of the preceeding figure, say the symbol \( a \) is selected for that column. Since \( u'(7) = c = u'(2) \), both positions 7 and 2 get assigned symbol \( a \). We get \( abadeaba \).

Now, we need to provide more details about the flagging process in order to handle columns with holes.

First, let us describe the period alignment guidelines. Repeat the following for all \( p_i \in \mathcal{P}(u) \setminus \{p_n\} \). Align \( u' \) into \( p_i \) columns. For each column containing at least two non-hole positions, select a symbol for that column. Then iterate through each non-hole position in the column, replacing its symbol with the selected symbol. This is because all non-hole positions in a column function as symbolic equivalents. Remember that when a flagged position is reached, it means that the position has already been assigned a symbol, so a “search and replace” must be performed across the entire word \( u' \) replacing all of these positions’ symbols with the new selected symbol. For example, Routine Normalization starts with input \( u = abca01acca \) and initializes \( u' = abcd01efghi \). Since \( \mathcal{P}(u) = \{9, 10\} \), \( u' \) is first transformed into \( u' = abcd01efgha \) say.

Second, let us describe the strictly weak period alignment guidelines. Repeat the following for all \( p'_i \in \mathcal{P}'(u) \setminus \mathcal{P}(u) \). When aligning \( u' \) into \( p'_i \) columns, the non-hole positions in a column are not necessarily assigned the same symbol. This is an important difference between the handling of a period and the handling of a weak period. Within a column, the positions are grouped into sets of symbolic equivalents as follows: If there are \( N \) holes in the column, then the column is broken up into \( N + 1 \) sets \( E_1, \ldots, E_{N+1} \) of equivalent positions: all positions above the first hole, all positions between the \( i \)th and \((i + 1)\)th hole where \( i \) ranges over \( \{1, \ldots, N - 1\} \), and all positions below the \( N \)th hole. Note that any set of symbolically equivalent positions can be the empty set.

Now, just iterate through the sets of symbolic equivalence. Select a symbol to assign each position of the set. If a position is unflagged, then flag it and assign the symbol. If the position is flagged,
then re-assign its symbol, and replace all such positions throughout \( u' \) with the newly selected symbol. Referring to the example above, since \( \mathcal{P}'(u) = \{3, 9, 10\} \), upon exiting the routine, input \( u \) has been transformed into \( u' = abca\varnothing_1cadea \) say. Note that after Normalization, no position remains flagged.

5 The Rule-Tree algorithm

There are two types of rules: rules of the form \((a \uparrow b)\) and rules of the form \((a \vartriangleright b)\) where \(a, b \in A' \cup \{\varnothing_1, \ldots, \varnothing_N\}\). For instance, the rule \((a \vartriangleright b)\) describes the relationship \(a \vartriangleright b\) where \(a, b\) denote the binary values of symbols \(a, b\) respectively. So, the binary value assigned to the character on the left cannot be computed with the binary value assigned to the character on the right. Note that the rule \((a \vartriangleright a)\) is invalid, and the rule \((a \vartriangleright b)\) is equivalent to \((b \vartriangleright a)\).

Let \(u\) be a partial input word over augmented alphabet \(A \cup \{\varnothing_1, \ldots, \varnothing_N\}\) and let \(u'\) be the normalized input partial word over augmented alphabet \(A' \cup \{\varnothing_1, \ldots, \varnothing_N\}\). Create a table with \(|u| - 1\) rows where \(R_{0}o\) corresponds to an alignment of \(u'\) with its suffix \(u'[|u| - i..|u|]\). For each \(0 < i < |u|\) such that \(|u| - i \notin \mathcal{P}'(u)\), a rule is generated such that \(u'(j) \vartriangleright u'[|u| - i..|u|](j)\), where \(j\) is a position of both \(u'\) and \(u'[|u| - i..|u|]\). For each \(0 < i < |u|\) such that \(|u| - i \in \mathcal{P}'(u)\), a rule is generated such that \(u'(j) \uparrow u'[|u| - i..|u|](j)\), where \(j\) is a position of both \(u'\) and \(u'[|u| - i..|u|]\). Let \(S_{i}\) be the (ordered) set of rules corresponding to \(R_{i}\). If \(|u| - i \notin \mathcal{P}'(u)\), then \(S_{i}\) is called an OR-set, otherwise it is called an AND-set. The rule-tree \(T_{u}\) provides a structured order in which rules are applied to the normalized input partial word \(u'\). It can be built by induction as follows: Let \(0 < i_1 < i_2 < \cdots < i_r < |u|\) be all the \(i\)'s for which \(S_{i}\) is an OR-set. The rules in \(S_{i_1}\) are the children of a dummy root (they are ordered from left to right as they appear in \(S_{i_1}\)) for \(2 \leq j \leq r\), the children of a node at depth \(j - 1\) are the rules in \(S_{i_j}\) ordered from left to right as they appear in \(S_{i_j}\). We can trim \(T_{u}\) in an obvious way.

We first illustrate our ideas of the rule generation process by considering input \(u = abo_1o_2cd\) where \(\mathcal{P}(u) = \{6\}\) and \(\mathcal{P}'(u) = \{2, 6\}\). After exiting the Normalization routine, \(u' = abo_1o_2cd\). Then \(u'[|u| - 1..|u|] = d, u'[|u| - 2..|u|] = cd, u'[|u| - 3..|u|] = o_2cd, \ldots\). This rule generating process involves the alignment of \(u'\) with each of its proper suffixes which in turn leads to the sets of rules as pictured in the table below. Begin by considering \(i = 1\), align \(u'\) with \(d\), and then compare all aligned characters: Here \(a\) and \(d\) align, therefore the first rule encountered is \((a \vartriangleright d)\) and so the set of rules produced is \(S_1 = \{(a \vartriangleright d)\}\). Now, consider \(i = 2\), align \(u'\) with \(cd\), and compare all aligned characters. The first rule generated is \((a \vartriangleright c)\) and the second rule is \((b \vartriangleright d)\). The set produced is therefore \(S_2 = \{(a \vartriangleright c), (b \vartriangleright d)\}\). Repeat this, increasing \(i\) until \(i = |u| - 1\). The sets \(S_1, S_2, S_3 = \{(a \vartriangleright o_2), (b \vartriangleright c), (\varnothing_1 \vartriangleright d)\}\) and \(S_5 = \{(a \vartriangleright b), (b \vartriangleright o_1), (\varnothing_1 \vartriangleright o_2), (o_2 \vartriangleright c), (c \vartriangleright d)\}\) are OR-sets. When \(i = 4\), since \(|u| - i = 2 \in \mathcal{P}'(u)\), \(S_4 = \{(a \uparrow o_1), (b \uparrow o_2), (\varnothing_1 \uparrow c), (o_2 \uparrow d)\}\) is an AND-set.

We now describe the rule-tree construction process (we will discuss assembling the rules and inserting the rules). Referring to our example where \(u' = abo_1o_2cd\), the tree has \((a \vartriangleright d)\) as the child of a dummy root; This child has a child, \((a \vartriangleright c)\), which itself has a sibling \((b \vartriangleright d)\) (which is considered the right child of \((a \vartriangleright d)\)); Each of these two children in turn have three children (the siblings are ordered from left to right): \((a \vartriangleright o_2), (b \vartriangleright c)\) and \((\varnothing_1 \vartriangleright d)\). And finally, each of these six children in turn have five children: \((a \vartriangleright b), (b \vartriangleright o_1), (\varnothing_1 \vartriangleright o_2), (o_2 \vartriangleright c)\) and \((c \vartriangleright d)\).

An obvious constraint to assembling the rule \((a \vartriangleright b)\) is that \(a \neq b\). In other words, the two symbols/characters cannot be the same. If the rule is valid and belongs to an OR-set, then it will be inserted in the tree. For each \(i\), all rules inserted in the tree for this \(i\) are siblings. If only one rule is inserted in the tree for this \(i\), then it is considered to be definitive (it will be
traversed and applied to $u'$ no matter what). Any level with only one rule is a definitive path. All rules on the same level are siblings, and treated as OR. It is important to note that when constructing the tree, a child cannot be equivalent to any of its predecessors. So, going back to the example of $u' = ab \diamond_1 c d$, the possible paths through the tree (and their binary conversions (if any) which will be described in Section 6) include: Path 1: (root), ($a \uparrow d$), ($a \uparrow c$), ($a \uparrow \diamond_2$), ($a \uparrow b$) with binary conversion 01\diamond111; Path 2: (root), ($a \uparrow d$), ($a \uparrow c$), ($a \uparrow \diamond_2$), ($b \uparrow \diamond_1$) with no binary conversion, etc.

We now describe the insertion routine. The tree construction begins by creating a dummy node to be used as the root. The root is assumed to have a depth of 0, and it is assumed that its value is unique and will not match the value of any other rule. The purpose of using a dummy node for the tree is to ensure that there is one and only one entry point into the tree. In the C++ implementation, if $i = 1$ produced an invalid rule, ($a \uparrow a$) for example, then the insertion process might have encountered multiple potential entry points from the rule production steps where $i > 1$.

The initial call to Routine Insert always attempts to insert a rule at the root node. The initial call then recursively traverses the tree and determines a node’s placement in the tree (if such placement exists). The result is a binary tree in which the left traversals represent parent-child relationships and right traversals indicate sibling-sibling relationships.

**Routine Insert**

A tree is created by inserting the rules into a specialized tree structure as follows:

- **Begin by trying to insert the rule at the root node of the tree.**

- **For any node, if the i value that created the rule is greater than the depth of the current node, then the rule is a descendant of the current node, as well as a descendant of all siblings of the current node.**

- **As such try to insert the node at the sibling (if any) and at each of the children.**

- **If no children are available, then the rule becomes the first child of the current node.**

6 Tree traversal and assignment graphs

In this section, we describe the tree traversal and binary conversion of our algorithm. Unless the user’s input $u$ is special, within the set of paths of the rule-tree, at least one path exists that produces a valid output when the rules of that path are encountered and applied to $A' \cup \{\diamond_1, \diamond_2, \ldots, \diamond_N\}$, the alphabet of $u'$, the normalized equivalent of $u$.

We start with an overview: Begin traversing the tree of $u'$ depth first by selecting the leftmost path in the tree. Associate a graph with the path by first considering the rules of the AND-sets and then the rules of the path (which are rules in OR-sets). Then attempt to 3-color the graph generated in order to produce a binary equivalent $v$. If no contradiction occurs, then periods and weak periods of $v$ match those of $u$ and a valid conversion has been obtained (in which case exit
the algorithm). If a contradiction occurs, then exit the path and traverse the next path in the
tree unless there is a queued assignment of binary values that has not been explored (in which
case explore such a queued assignment). If after visiting all the tree no path has been found with
a graph that is 3-colorable, then a special partial word has been recognized.

First, each of the paths in the Rule-Tree is associated with a labelled undirected graph
$G = (V, E)$, called an assignment graph, as follows: $V = A' \cup \{\diamond_1, \diamond_2, \ldots, \diamond_N\}$ and $E = E_+ \cup E_-$ where
$E_+$, the set of positive edges labelled by $+$, consists of the rules in the AND-sets which are of the
form $(a \uparrow b)$, and $E_-$, the set of negative edges labelled by $-$, consists of the rules of the path
which are of the form $(a \not\uparrow b)$. All rules in the AND-sets must be satisfied to reach a solution
regardless of which path through the rule tree we traverse, and so $E_+$ is part of every assignment
graph. Using our example $u' = ab\diamond_1\diamond_2cd$ and the tree we generated in the previous section, we
see that the four rules in the AND-set $S_4$ are $(a \uparrow \diamond_1)$, $(a \uparrow \diamond_2)$, $(\diamond_1 \uparrow c)$, and $(\diamond_2 \uparrow d)$. Now, if
we follow the first path which generates the rules $(a \not\uparrow d)$, $(a \not\uparrow c)$, $(a \not\uparrow \diamond_2)$, $(a \not\uparrow b)$, then we get the
graph shown in Figure 2.

![Figure 2: Uncolored assignment graph of Path 1 in Rule-Tree of $ab\diamond_1\diamond_2cd$.](image)

Now, let us introduce our graph coloring algorithm which divides vertices in $A' \cup \{\diamond_1, \diamond_2, \ldots, \diamond_N\}$ into three sets. Any vertex can be assigned to the 0-set or the 1-set, but of course only elements of
{$\diamond_1, \diamond_2, \ldots, \diamond_N$} can be assigned to the $\diamond$-set (this ensures that $H(v) \subset H(u)$). By default a vertex
is white. White corresponds to 0, black to 1, and grey to $\diamond$. The two vertices of a positive edge
must have compatible colors, and the two vertices of a negative edge cannot have the same color,
and neither can be colored grey. Note that some symbol $\diamond_j$ can appear in an incompatibility rule,
but due to the fact that $\diamond$ is compatible with any other symbol of the alphabet, this $\diamond_j$ does not function as a hole, and therefore its vertex cannot be colored grey. If a vertex has been discovered,
then we flag it. By default, all vertices' discovered flag is set to false.

To begin our coloring of a graph, we must first choose an entry vertex into the graph, which
we color white (it will be white by default). We flag it as discovered, and start visiting incident
edges. We will visit each edge radiating from a vertex in depth first fashion, starting from our
entry vertex. In order to describe how our algorithm visits and colors the vertices of the graph,
we must introduce the idea of an expected color which is calculated as follows:

<table>
<thead>
<tr>
<th>Color</th>
<th>Positive Edge (+)</th>
<th>Negative Edge (−)</th>
</tr>
</thead>
<tbody>
<tr>
<td>White</td>
<td>White/Grey</td>
<td>Black</td>
</tr>
<tr>
<td>Black</td>
<td>Black/Grey</td>
<td>White</td>
</tr>
<tr>
<td>Grey</td>
<td>White/Black</td>
<td>Impossible</td>
</tr>
</tbody>
</table>

So for instance, if we start at a white vertex and cross a positive edge to arrive at a new vertex,
we expect that the color of the new vertex will be white or grey. Otherwise, if we cross a negative
edge, we would expect the new vertex to be black. By default, if we start at a grey vertex, we
choose our expected color to be white. Later, if we discover that white was a bad choice, we can
re-visit this edge and use black as our expected color.
We assume that the edges of the graph are in some order. The method used for ordering is unimportant so long as it is consistent. Our algorithm simply orders edges based on the order which they appear in the path we have chosen from our rule tree. We pick an edge incident on our current vertex \( a \), which is currently the entry vertex. As we cross an edge to visit a neighbor \( b \), we calculate the expected color. One of three situations will occur.

*First*, the vertex \( b \) is undiscovered, in which case flag \( b \) and color it the expected color. If an unvisited edge incident on \( b \) exists, visit it, otherwise return to \( a \), and continue visiting the edges incident on \( a \).

*Second*, the vertex \( b \) is discovered and is colored as expected. Then return to \( a \) (let us note that the edge between these vertices is not the edge which led to the discovery of \( b \)). Then continue visiting edges incident on \( a \) if any remain unvisited.

*Third*, the vertex \( b \) is discovered but is not colored as expected. This case represents a contradiction. If we can return to some vertex \( \diamond_j \) with no negative edges incident on it, then we might be able to color it grey and try a different expected color as we re-traverse the subgraph which led to the contradiction. This would allow us to alleviate the contradictory coloring, and continue processing the graph. Unfortunately, if no such vertex \( \diamond_j \) exists, we know immediately that this assignment graph cannot be 3-colored, so we must abandon it and explore the next path in our rule tree.

After fully visiting this subgraph, we will return to \( \diamond_j \) and explore the other edges incident on it. When exploring these edges, we might again find that our expected color causes a contradiction in the subgraph we are visiting. We only abort the assignment graph if an edge incident on some \( \diamond \) vertex forces us to backtrack to this vertex more than once. Thus, we might find ourselves backtracking to this vertex many times if more than one edge causes a contradiction in its subgraph, but this is not a problem so long as no edge requires us to backtrack more than once.

At some point in our traversal of this assignment graph we will have visited all edges, so long as no contradiction has occurred that we are unable to resolve. Thus, barring these unresolvable graphs, we will have an appropriate 3-colored graph. White vertices correlate to 0 in \( v \), black vertices to 1, and grey vertices to \( \diamond \). The value of a vertex is a symbol of the alphabet of \( u' \), so we look up the color of each position in \( u' \) and replace it with the appropriate encoding over \( \{0, 1, \diamond\} \). After we have finished this substitution process, we have found our solution.

Referring to the example above where \( u' = ab\diamond_1\diamond_2cd \), we have chosen the vertex \( a \) as our entry point into the assignment graph. The vertex coloring process leads to Figures 3. Now we see that

![Figure 3: 3-colored assignment graph of Path 1 in Rule-Tree of ab\diamond_1\diamond_2cd.](image)

\( a \) is white, \( b \) is black, \( c \) is black, \( d \) is black, \( \diamond_1 \) is grey, and \( \diamond_2 \) is black. Translating white, black, and grey to 0, 1 and \( \diamond \), Path 1 leads to the assignment \( a \rightarrow 0, b \rightarrow 1, \diamond_1 \rightarrow \diamond, \diamond_2 \rightarrow 1, c \rightarrow 1, d \rightarrow 1 \), in turn yielding the solution, \( v = 01\diamond111 \).
7 Our main result

In this section, we state our main result.

Lemma 1. Let $x$ be a partial word over augmented alphabet $A \cup \{\diamond_1, \ldots, \diamond_N\}$. Set $u = \diamond_1x$ and let $1 \leq i < |u|$. Then the following hold:

- $|u| - i \in \mathcal{P}(u)$ if and only if $|u| - i \in \mathcal{P}(x)$.
- $|u| - i \in \mathcal{P}'(u)$ if and only if $|u| - i \in \mathcal{P}'(x)$.

Lemma 2. Let $x$ be a non-empty partial word over augmented alphabet $A \cup \{\diamond_1, \ldots, \diamond_N\}$ and let $a \in A$. Set $u = ax$ and let $1 \leq i < |u|$. Also set $|x| = k(|u| - i) + r$ where $0 \leq r < |u| - i$, and set $B = \{x(|u| - i - 1), x(2|u| - 2i - 1), \ldots, x(k|u| - ki - 1)\}$. Then

- If $|u| - i \in \mathcal{P}(u)$, then $|u| - i \in \mathcal{P}(x)$.
- If $|u| - i \in \mathcal{P}'(u)$, then $|u| - i \in \mathcal{P}'(x)$.
- If $x(|u| - i - 1) \in \{\diamond_1, \ldots, \diamond_N\}$, then the following hold:
  - If $|u| - i \in \mathcal{P}(x)$ and $B \subset \{\diamond_1, \ldots, \diamond_N, a\}$, then $|u| - i \in \mathcal{P}(u)$.
  - If $|u| - i \in \mathcal{P}'(x)$, then $|u| - i \in \mathcal{P}'(u)$.
- If $x(|u| - i - 1) \notin \{\diamond_1, \ldots, \diamond_N\}$, then the following hold:
  - If $|u| - i \in \mathcal{P}(x)$ and $x(|u| - i - 1) = a$, then $|u| - i \in \mathcal{P}(u)$.
  - If $|u| - i \in \mathcal{P}'(x)$ and $x(|u| - i - 1) = a$, then $|u| - i \in \mathcal{P}'(u)$.

Now, let $u$ be a partial input word over augmented alphabet $A \cup \{\diamond_1, \ldots, \diamond_N\}$ and let $u'$ be the normalized input partial word over augmented alphabet $A' \cup \{\diamond_1, \ldots, \diamond_N\}$. As in Section 5, create a table with $|u| - 1$ rows where Row$_i$ corresponds to an alignment of $u'$ with its suffix $u'[|u| - i..|u|]$. Let $S_i$ be the (ordered) set of rules corresponding to Row$_i$.

Lemma 3. Let $u$ be a partial input word over augmented alphabet $A \cup \{\diamond_1, \ldots, \diamond_N\}$ and let $u'$ be the normalized input partial word over augmented alphabet $A' \cup \{\diamond_1, \ldots, \diamond_N\}$. If $1 \leq i < |u|$, then the following hold:

1. $|u| - i \in \mathcal{P}(u)$ if and only if no distinct $a, b$ in $A'$ exist such that one of the following holds:
   - $(a \not\equiv b) \in S_i$.
   - There exist $1 \leq i_1 < i_2 < \cdots < i_r \leq N$ such that $(a \not\equiv \diamond_{i_1}, \diamond_{i_1} \not\equiv \diamond_{i_2}, \ldots, \diamond_{i_{r-1}} \not\equiv \diamond_{i_r}, \diamond_{i_r} \not\equiv b) \in S_i$.

2. $|u| - i \in \mathcal{P}'(u) \setminus \mathcal{P}(u)$ if and only if no distinct $a, b$ in $A'$ exist such that $(a \not\equiv b) \in S_i$.

Proof. Upon exiting the Normalization routine, input $u$ over augmented alphabet $A \cup \{\diamond_1, \ldots, \diamond_N\}$ has been transformed into $u' = a_0 \cdots a_{n-1}$ over augmented alphabet $A' \cup \{\diamond_1, \ldots, \diamond_N\}$ where $|u'| = |u|$, $\mathcal{P}(u') = \mathcal{P}(u)$ and $\mathcal{P}'(u') = \mathcal{P}'(u)$. First, let us assume that $1 \leq i \leq \lfloor n/2 \rfloor$. Here $|u| - i \notin \mathcal{P}(u)$ if and only if $a_0 \not\equiv a_{|u|-i}$ and both $a_0, a_{|u|-i} \in A'$, or $a_1 \not\equiv a_{|u|-i+1}$ and both $a_1, a_{|u|-i+1} \in A'$, or ... if and only if at least one of the rules in $S_i$ is satisfied of the form $(a \not\equiv b)$
where $a, b$ are distinct elements of $A'$. And $|u| - i \notin \mathcal{P}'(u) \setminus \mathcal{P}(u)$ if and only if at least one of the rules in $S_i$ is satisfied of the form $(a \not\not\not b)$ where $a, b$ are distinct elements of $A'$.

Second, let us assume that $|n/2| < i < |u|$. Create a multialignment of length $|u| - i$ for segments of $u'$: Row 1 of the alignment corresponds to the prefix of length $|u| - i$ of $u'$, Row 2 to the prefix of length $|u| - i$ of $u'[|u| - i...|u|]$, and so on. In the case of a period, an error is counted for each column of the alignment that contains occurrences of two distinct characters of $A'$. In the case of a weak period (that is not a period), an error is counted for each column of the alignment that contains consecutive occurrences of two distinct characters of $A'$. Here, $|u| - i \notin \mathcal{P}(u)$ if and only if there exists a column number $j, 0 \leq j < |u| - i$, that generates an error. In other words, Column $j$ contains either two consecutive elements, say $a_k$ and $a_{|u| - i + k}$ such that $a_k, a_{|u| - i + k} \in A'$ and $a_k \not\not\not a_{|u| - i + k}$, in which case, $(a_k \not\not\not a_{|u| - i + k}) \in S_i$, or three consecutive elements, say $a_k, a_{|u| - i + k}, a_{2|u| - 2i + k}$, such that $a_k, a_{2|u| - 2i + k} \in A'$, and $a_k \not\not\not a_{2|u| - 2i + k}$, in which case, both $(a_k \not\not\not a_{|u| - i + k})$ and $(a_{|u| - i + k} \not\not\not a_{2|u| - 2i + k})$ are rules in $S_i$, or four consecutive elements ... And $|u| - i \notin \mathcal{P}'(u) \setminus \mathcal{P}(u)$ if and only if there exists a column number $j, 0 \leq j < |u| - i$, that generates an error. In other words, Column $j$ contains two consecutive elements, say $a_k$ and $a_{|u| - i + k}$, such that $a_k, a_{|u| - i + k} \in A'$ and $a_k \not\not\not a_{|u| - i + k}$. In such case, $(a_k \not\not\not a_{|u| - i + k}) \in S_i$.

Theorem 4. Let $u$ be a nonspecial partial word over an alphabet $A$. Then there exists a path in $T_u$ from the root to a leaf that generates a partial word $v$ over the alphabet $\{0, 1\}$ such that $0 \in H(v)$ if and only if $0 \in H(u)$, $H(v) \subset H(u)$, $v$ starts with 0 or $\diamond$, $|v| = |u|$, $\mathcal{P}(v) = \mathcal{P}(u)$ and $\mathcal{P}'(v) = \mathcal{P}'(u)$. 

Proof. The proof is by induction on the length of $u$. The result is trivially true for partial words of length at most 2. If $u$ is a partial word of length $n > 2$, then we first consider the case where $u' = \diamond_1 x$ for some partial word $x$ of length $n - 1$ (here $u'$ denotes the normalized input partial word). Since $u'$ is normalized, $x$ is also normalized. By the inductive hypothesis, there exists a path $q$ in $T_x$ from the root to a leaf that generates a partial word $y$ over the alphabet $\{0, 1\}$ such that $H(y) \subset H(x), 0 \in H(y)$ if and only if $0 \in H(x), y$ starts with 0 or $\diamond, |y| = |x|, \mathcal{P}(y) = \mathcal{P}(x)$ and $\mathcal{P}'(y) = \mathcal{P}'(x)$. Here $v = \diamond y$ satisfies the desired requirements. First, $H(v) \subset H(u)$ since $0 \in H(u) \cap H(v)$ and $H(y) \subset H(x)$, and $|v| = |u|$ since $|y| = |x|$. There remains to show that $\mathcal{P}(v) = \mathcal{P}(u)$ and $\mathcal{P}'(v) = \mathcal{P}'(u)$. Note that both $|u|$ and $|u| - 1$ belong to $\mathcal{P}(v), \mathcal{P}(u), \mathcal{P}'(v)$ and $\mathcal{P}'(u)$. So let $1 \leq i < |u| - 1$. By Lemma 1, if $|u| - i - 1 \in \mathcal{P}(u)$, then $|u| - i - 1 \in \mathcal{P}(x) = \mathcal{P}(y)$ and so $|u| - i - 1 \in \mathcal{P}(v)$ (the converse is similar). The equality $\mathcal{P}'(v) = \mathcal{P}'(u)$ follows similarly.

We now consider the case where $u' = ax$ for some $a \in A'$ and some partial word $x$ of length $n - 1$. By the inductive hypothesis, there exists a path $q$ in $T_x$ from the root to a leaf that generates a partial word $y$ over the alphabet $\{0, 1\}$ such that $H(y) \subset H(x), 0 \in H(y)$ if and only if $0 \in H(x), y$ starts with 0 or $\diamond, |y| = |x|, \mathcal{P}(y) = \mathcal{P}(x)$ and $\mathcal{P}'(y) = \mathcal{P}'(x)$. Let $S'_i, \ldots, S'_{|y|+1}$ be the sets of rules corresponding to the table of $x$. We illustrate ideas of the proof by considering one of several cases.

Let us consider the case where $a$ is a letter in $x$ (by assumption, $a = 0$ and so all $a$'s in $x$ have been assigned 0 to get $y$) and where $x$ ends with $a$ (and $y$ ends with 0). Here $v = 0y$ satisfies the desired requirements. We need to show that $\mathcal{P}(v) = \mathcal{P}(u)$ and $\mathcal{P}'(v) = \mathcal{P}'(u)$. Note that both $|u|$ and $|u| - 1$ belong to $\mathcal{P}(v), \mathcal{P}(u), \mathcal{P}'(v)$ and $\mathcal{P}'(u)$. So let $1 \leq i < |u| - 1$. We prove the inclusion $\mathcal{P}(u) \subset \mathcal{P}(v)$ (the inclusion $\mathcal{P}(v) \subset \mathcal{P}(u)$ is similar). If $|u| - i - 1 \in \mathcal{P}(u)$, then by Lemma 2, $|u| - i - 1 = |x| - i \in \mathcal{P}(x)$ (and $|u| - i - 1 \in \mathcal{P}(y)$ since $\mathcal{P}(y) = \mathcal{P}(x)$).

By Lemma 3, no rule in $S'_i$ exists of the form $(b \not\not\not c)$ with distinct $b, c \in A'$, and no subset exists of $S'_i$ of the form $\{\overline{(b \not\not\not c_1), (c_1 \not\not\not c_2), \ldots, (c_{i-1} \not\not\not c_i), (c_i \not\not\not c)}\}$ with distinct $b, c \in A'$.
A' and 1 ≤ i_1 < i_2 < ⋯ < i_r ≤ N. Again using Lemma 3, no rule in S_{i+1} exists of the form (b ' c) with distinct b, c ∈ A', and no subset exists of S_{i+1} of the form \{(b ' c_{i_1}), (c_{i_1} ' c_{i_2}), \ldots, (c_{i_{r-1}} ' c_{i_r}), (c_{i_r} ' c)\} with distinct b, c ∈ A' and 1 ≤ i_1 < i_2 < ⋯ < i_r ≤ N. So B_x = \{x(|u| - (i + 1) - 1), x(2|u| - 2(i + 1) - 1), \ldots\} ⊂ \{c_1, \ldots, c_N, a\}. There are two cases to consider (in either case the path is q since ∅ = ∅): If x(|u| - i - 2) = a, then since a = 0, we have that y(|u| - i - 2) = 0. By Lemma 2, |u| - i - 1 ∈ P(v). If x(|u| - i - 2) ∈ \{c_1, \ldots, c_N\}, then since H(y) ⊂ H(x), we have three possibilities: (1) y(|u| - i - 2) = 0; (2) y(|u| - i - 2) = 1; and (3) y(|u| - i - 2) = c_j for some j. Possibility (1) follows as before. For (2), since |u| - i - 1 ∈ P(y), we have B_y = \{y(|u| - (i + 1) - 1), y(2|u| - 2(i + 1) - 1), \ldots\} ⊂ \{c_1, \ldots, c_N, 1\} and B_x ⊂ \{c_1, \ldots, c_N\}. In this case, some of the c’s have a binary equivalent of 1. For (3), if B_y ⊂ \{c_1, \ldots, c_N, 0\}, then |u| - i - 1 ∈ P(v). Otherwise, B_y ⊂ \{c_1, \ldots, c_N, 1\} and B_x ⊂ \{c_1, \ldots, c_N\}, and the result follows as in (2).

To conclude, we did some analysis of how rare is “speciality.” We ran tests on partial words of lengths up to 13 over a ternary alphabet, and found that less than .001% of words are “special.”

References