SHIREY, BRIAN, M.S. Periods, Partial Words, and an Extension of a Result of Guibas and Odlyzko. (2007)
Directed by Dr. Francine Blanchet-Sadri. 79pp.

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). More specifically, for every word u, there exists a word v over the alphabet $\{0, 1\}$ such that u and v have the same length and the same set of periods. 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 with at most one hole over the alphabet $\{0, 1\}$ such that u and v have the same length, the same set of periods, the same set of weak periods, and $H(v) \subset H(u)$, where $H(u)$ (respectively, $H(v)$) denotes the set of holes of u (respectively, v). In this paper, we extend this result further to a large class of partial words. Given a partial word u belonging to that class, our proof provides an algorithm to compute a partial word v over $\{0, 1\}$ sharing the same length and same sets of periods and weak periods as u, and satisfying $H(v) \subset H(u)$.

PERIODS, PARTIAL WORDS, AND AN EXTENSION OF A RESULT OF GUIBAS AND ODLYZKO

by

Brian Shirey

A Thesis Submitted to
the Faculty of The Graduate School at
The University of North Carolina at Greensboro
in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Greensboro
2007

Approved by

_____
Committee Chair

Merely a speck of dust, incapable of life without the guidance of a breeze onto the perfect spot, perhaps a mound of fallen leaves, perhaps an emerald bed of moss, an orchid seed, most minuscule of all, cannot survive without the nurture of a symbiote.

So to my mother Alice, my father Jim, and my dearest friend Craig, without whom I surely would have fallen to the ground and never grown into the person I am today, I dedicate this thesis.

APPROVAL PAGE

This thesis has been approved by the following committee of the Faculty of The Graduate School at The University of North Carolina at Greensboro.

Committee Chair_____

Committee Members_____

_____

_____
Date of Acceptance by Committee

_____
Date of Final Oral Examination

ACKNOWLEDGMENTS

On days when I felt most discouraged, facing unforeseen complications and hurdles, I could always count on Dr. Francine Blanchet-Sadri for guidance and her innate ability to kindle the imagination. For this, I wish to thank her.

I also wish to extend my utmost appreciation to this university and its faculty for nurturing my academic growth and for providing this most essential stepping stone along my path to personal success.

Finally, I would like to acknowledge the consideration, time, and effort given to my thesis by the thesis committee members.

This material is based upon work supported by the National Science Foundation under Grant Numbers CCF-0207673 and DMS-0452020.

TABLE OF CONTENTS

Page

CHAPTER

I. INTRODUCTION 1

II. PRELIMINARIES 4

2.1. Words 4
2.2. Partial Words 8
2.3. Graphs and Trees 13

III. NORMALIZATION ROUTINE 16

3.1. Strong Periods of $\boldsymbol{u}$ 19
3.2. Strictly Weak Periods of $\boldsymbol{u}$ 21

IV. THE RULE-TREE ALGORITHM 24

4.1. Rules 24
4.2. Candidate Weak Period Enumeration 25
4.3. AND-sets And OR-sets 32
4.4. Binary Rule-Tree 35

V. TREE TRAVERSAL AND ASSIGNMENT GRAPHS 44

5.1. Vertex Structure 45
5.2. Graph Construction 46
5.3. Graph Coloring 49

VI. THE SPECIAL CLASS OF PARTIAL WORDS 63

BIBLIOGRAPHY 70

CHAPTER I

INTRODUCTION

Periodicity of words and partial words can have many practical applications in ordinary and quite familiar fields such as text searching and pattern matching. These sorts of common place algorithms can in turn be applied to more sophisticated and technologically advanced topics such as computational biology and theoretical mathematics [13, 19, 20, 21, 27, 29, 30, 32, 45]. For instance, DNA is a biological analog of partial words, and therefore many concepts introduced in this thesis are relevant to computations on genetic code [15, 30, 41, 44, 48]. Research on DNA and gene sequences is advancing at a rapid pace due to the potential benefits of genetically modified crops, gene therapy for incurable diseases, and a myriad other genetic technologies. Therefore, the importance of research on partial words cannot be understated. Intrinsically, gene sequencing techniques are often prone to error, be it human error or otherwise. Mutations themselves may be viewed as errors. Mutations can be as minuscule as a single base substitution or the deletion of a position, and they can be as complex as an insertion of many nucleotides [52]. Mathematically we can acknowledge mutations and other practical errors by representing them with holes in our sequence. However, we must then alter our mathematical models to accommodate these holes.

We cannot begin a proper discussion on periodicity without first introducing the theorem of Fine and Wilf. Given a word u with at least two periods, p, q, if $|u| \geq p + q - \gcd(p, q)$, then $\gcd(p, q)$ is a period of u [26]. The theorem of Fine and Wilf allows Guibas and Odlyzko's theorem to guarantee that for every word u

over an alphabet A, there exists a word v over $\{0, 1\}$ such that the set periods of u and the set of periods of v are equal [28]. This tells us that the periodicity of a word is independent of its alphabet. The next step in the discipline of periodicity is to extend these concepts to partial words. Blanchet-Sadri and Chriscoe were able to prove that the periodicity of a partial word with at most one hole is also independent of its alphabet [6]. A novel and efficient algorithm was devised to aid in the proof of concept. The algorithm of Blanchet-Sadri and Chriscoe, which finds some solution v for partial word u with at most one hole, works by classifying u into a series of cases and sub cases. Each classification dictates how to construct a primitive factor which can be used to compute a final solution. Attempts to amend this approach have thus far proven to be unsuccessful due to the relative complexity of the algorithm itself. Indeed, we could potentially use this idea to design an algorithm which works for partial words with at most two holes, but the possibility of generalizing the approach to an arbitrary number of holes seems unlikely.

Fundamentally the problem of finding a binary conversion for a partial word containing an arbitrary number of holes is significantly more complex than it is for the cases of one hole or no holes at all. Ideally we would hope to extend the efficient, linear time algorithm first developed by Blanchet-Sadri and Chriscoe. However, due to the relatively unscalable nature of the former algorithm, we have devised a completely new approach to deal with this complex problem on partial words. Initially the goal of this new algorithm was to determine some v for any u with at most two holes, but as the algorithm solidified, it became apparent that the new approach would likely work for partial words with any number of holes. The algorithm described in this thesis has been implemented using C++ and the World Wide Web server interface is available at `http://www.uncg.edu/mat/bintwo/`.

Now let us briefly summarize the various stages of this algorithm. We first explore a partial word to determine key pieces of information about it such as its periodicity and the location of its holes. Using this information, we will be able to construct a tree whose nodes characterize the relationship between sets of positions of our word. Each path of our tree represents a potential solution, but not every path is acceptable. So, we must scrutinize each path until we find a valid solution, and we do this using graphs. By 3-coloring a graph, constructed from data contained in a particular path of our tree, we can tell if that path provides us a valid output or not. This relatively complex algorithm is not computationally efficient as the algorithm of Blanchet-Sadri and Chriscoe, but it does provide an output v over $\{0, 1\}$ for any u, so long as such a conversion exists. Otherwise, this algorithm can be treated as a recognizer of the special class of partial words for which no v over $\{0, 1\}$ exists with $H(v) \subset H(u)$, $P(v) \subset P(u)$, and $P'(v) \subset P'(u)$. We use $H(w)$ to denote the set of holes of partial word w, $P(w)$ to denote the set of periods of w, and $P'(w)$ to denote the weak periods of w.

CHAPTER II

PRELIMINARIES

This section is devoted to fix notations, and to review definitions and elementary properties of words and partial words.

2.1 Words

An alphabet is a concept with which we can all identify, however we need to examine a precise mathematical definition for this term. In mathematics, an alphabet is a finite set of symbols. For instance, we are all familiar with alphabets like the English alphabet, $\{a, b, c, \ldots, z\}$. Of particular relevance to this research and computer science in general is the binary alphabet, $\{0, 1\}$. As we have explained, an alphabet is a set, therefore the usual set operations are applicable. These operations include union($\cup$), intersection($\cap$), and star(*). The star operation is one with which you might not be familiar, so we will start with a demonstration of it.

Example 1 *We represent our alphabet, whatever it may be, with A. For this example, let us assume $A = \{0, 1\}$. Here the elements of A^* are written in lexicographic order yielding the set $A^* = \{\epsilon, 0, 1, 00, 01, 10, 11, \ldots\}$.*

We use A^* to denote the infinite set of words over alphabet, A, where each element, $s \in A^*$, is called a word. Be aware that there is a special word called the empty word, denoted by ϵ, which has a length of 0. The empty word is the result of a 0-concatenation and is the first element of A^* when this set is ordered

lexicographically. You will find that the mathematical definition of a word is similar to our inherent understanding of words in general. Words are analogous to character arrays or strings in computer science. This observation allows us to borrow the concept of a position index to identify specific positions within a word. We have chosen to denote the symbol of a position by enclosing its index within a set of parentheses. Using this notation for $u = 011$, we would write $u(1) = 1$ to indicate that position one of u is the symbol 1. Do not let the index confuse you, we have simply chosen to start our indices at 0 and end them with $|u| - 1$, where $|u|$ denotes the length of our word u, since this adheres more closely to the C++ implementation of our algorithm. Obviously this is a matter of convention, and in certain programming environments, it may be more sensible to denote positions with indices spanning from 1 to $|u|$.

A factor or substring of a word is a discrete stretch of positions within the word. We use basic boundary notation (*i.e.* "$[,], (,)$") when describing substrings. The symbols "[" and "]" are inclusive bounds, whereas "(" and ")" are exclusive. So, $u[0..4]$ represents the first five positions of u but $u[0..4)$ represents the first four positions. A factor of u which begins at position 0, such as $u[0..4]$, is called a prefix of u. Likewise, a factor of u which ends at position $|u| - 1$, such as $u[2..5]$ where $|u| = 6$, is called a suffix of u. Let us look at an example of indexing and substrings.

Example 2 *Let us assume our alphabet,* $A = \{a, b\}$ *and we pick* $u = ababb$ *from this alphabet. Notice* $u(1) = u(3) = u(4) = b$*, which we would state by saying that the symbols in positions* 1, 3, *and* 4 *are all b's. In contrast,* $u(0) = u(2) = a$*, and we would say that the symbols in positions* 0 *and* 2 *are both a's. Here,* $u[0..1]$ *represents ab, as does the equivalent notation* $u[0..2)$*.*

A word $u = a_0 a_1 \dots a_{n-1}$ of length n over A can be defined by the total function $u : \{0, \dots, n-1\} \to A$ where $u(i) = a_i$. Words, as well as alphabets,

are subject to concatenation. If we have the words $u_1 = 001$ and $u_2 = 111$, we can concatenate them as $u_1u_2 = 001111$. We can designate that we want to concatenate a word with itself a specific number of times using the notation u^i, where $i \geq 0$. Thus we get ...

$$u_i = \underbrace{uu\ldots u}_{i \text{ copies of } u}$$

Most often we refer to this kind of concatenation as the i-power of u. The 0-power of any word u generates the empty word, ϵ. Let us look at an example of i-power.

Example 3 *We select $u = aba$ from the alphabet $\{a, b\}$. We should mention that this particular alphabet is a binary alphabet, though we have not explicitly chosen to represent the elements with 0 and 1. If we pick $i = 3$ the result is $u^3 = abaabaaba$.*

Notice that our word, u^3, contains a repeated substring, *aba*. We call such repetitions *periods*. Before we formally explain what a period is, we can use tables to visually demonstrate the concept.

Let us make a table with three columns. Initially our table has only one row, and in each cell, we place one position of u^3. Once we reach the end of a row, if there are any positions left in our word, then we start a new row, and continue filling out each entry with the next position of u^3. Placing $u^3 = abaabaaba$ into a table with three columns means we will have three rows for a total of 9 cells to accommodate the length of this word. This is what the table would look like.

Row	1^{st} column	2^{nd} column	3^{rd} column
1	a	b	a
2	a	b	a
3	a	b	a

Notice when we align u^3 into three columns like this, each column is uniformly composed of the same symbol. This is an especially interesting characteristic of certain words, and a characteristic which is a focus of this research. Eventually we will be able to demonstrate that these patterns, which we will call periods, are independent of the alphabet of a word. Now let us give a formal definition for this feature of words. We use $\mathcal{P}(u)$ to denote the periods of partial word u. A *period* is an integer p satisfying $1 \leq p \leq n$ where $a_i = a_{i+p}$ for $0 \leq i < n - p$. Here, $n = |u|$, the length of our word, u. In other words, p is a period of u if the prefix and suffix of u of length $n-p$ are equal. Arbitrarily, $|u|$ is always a period of u. We call the smallest period of a word, u, the *minimal period*, and it is denoted by $p(u)$. Some properties of periods in words include: If p is a period of a word u, then any multiple of p smaller or equal to n is also a period of u. For more details, we refer the reader to [17, 28, 39].

Example 4 *Let us build the tables corresponding to each period of* $u = abaabaaba$. *The set of periods of* u, $\mathcal{P}(u)$, *is* $\{3, 6, 8, 9\}$.

$3 \in \mathcal{P}$

a	b	a
a	b	a
a	b	a

$6 \in \mathcal{P}$

a	b	a	a	b	a
a	b	a			

$8 \in \mathcal{P}$

a	b	a	a	b	a	a	b
a							

If we align the positions of u into p columns and verify that for each column, all positions in that column are compatible, we can conclude that $p \in \mathcal{P}(u)$. In

order to define the concept of *compatible*, let us first denote our alphabet as the set of distinct symbols $\{a_1, a_2, \ldots, a_n\}$ such that $a_k \in A$ for all $1 \leq k \leq n$. Using this notation, we say that two symbols are compatible, $a_i \uparrow a_j$, if and only if $i = j$. Conversely, two symbols are considered incompatible, $a_i \not\uparrow a_j$, if $i \neq j$.

Example 5 *As a counterexample to our alignment of u into columns corresponding to its periods, let us align u into four columns. Note that $4 \notin \mathcal{P}(abaabaaba)$.*

a	b	a	a
b	a	a	b
a			

Figure 2.1: Misalignment of u into four columns

In Figure 2.1 you can see that there are several columns which are not uniform. In the first column alone, we observe $u(0) \not\uparrow u(4)$ and $u(4) \not\uparrow u(8)$. Therefore, you should conclude that 4 is not in the set of periods of u.

2.2 Partial Words

Scenario 1 *Think about a scenario such as this...*

You have received an important letter through the mail. Moisture has leaked onto the contents, and as a result, the ink has been water marked and is illegible in certain spots. You can clearly read some pieces of the letter, but some areas are, in essence, gone. You might read a partial word, like "per..ds", and try to decipher it by context. However, in the absence of such context, you would have to treat such gaps in information much more carefully. This is because you know nothing about the gap, or hole. Indeed, you do not even know exactly how many characters are missing, let alone the identity of such characters.

In theoretical and real world applications, we do not always have the luxury of dealing with full words as they were described in the previous section. Often times, we do not have a complete set of data. In these situations, there are holes in our words, which we must somehow account for when we examine the information we have. Unfortunately, we know absolutely nothing about this missing information; it could span hundreds, thousands, or even more positions. Conversely, we might be missing no information at all, or perhaps, we might only be missing a single position of data. Somehow, we must cope with this lack of information if we intend to extend our concepts on words to the larger class, partial words. A *partial word* is a word of length n over the alphabet $A \cup \{\diamond\}$, where the $\diamond$ symbol is used to represent a hole. Notice that $\diamond$ is not a part of the alphabet. Similar to a full word, a partial word can be represented by a partial function, $u : \{0, \ldots, n-1\} \to A$. 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 $u_\diamond$) is the total function $u_\diamond : \{0, \ldots, n-1\} \to A \cup \{\diamond\}$ defined by

$$u_\diamond(i) = \begin{cases} u(i) & \text{if } i \in D(u), \\ \diamond & \text{otherwise.} \end{cases}$$

For the sake of convenience, we will contract the statement "the partial word with companion *abb*$\diamond$*bbcbb*" to a simpler form "the partial word *abb*$\diamond$*bbcbb*". As you can see, the placeholder symbol, $\diamond$, provides us a convenient notation to designate positions in which we are missing information. To simplify our representation of partial words, let us consider the bijectivity of the map $u \mapsto u_\diamond$, which allows us

to define for partial words concepts such as concatenation, factor, prefix, suffix, i-power, etc. in a trivial way. For instance, the word $u_\diamond = abb\diamond b\diamond cbb$ is the companion of the partial word u of length 9 where $D(u) = \{0, 1, 2, 4, 6, 7, 8\}$ and $H(u) = \{3, 5\}$.

When dealing with partial words, we have introduced a new symbol $\diamond$. A surprising property of this symbol is that it is compatible with every other symbol of A. So, for any symbol $a \in A$ we can say $a \uparrow \diamond$ (the symbol a is compatible with $\diamond$). This is an expansion of our earlier definition of compatibility because now we define it as $a_i \uparrow a_j$ if $i = j$, or if one of the following is true ... $a_i = \diamond$ or $a_j = \diamond$. We will further refine the definition of incompatibility as $a_i \not\uparrow a_j$ if $i \neq j$ and neither $a_i \neq \diamond$ nor $a_j \neq \diamond$. Let us look at how this property affects the periodicity of a partial word. Let us say we align $u = abb\diamond b\diamond cbb$ into three columns just as we have done in previous examples. If we examine each column of Figure 2.2, we see that

a	b	b
$\diamond$	b	$\diamond$
c	b	b

Figure 2.2: Alignment of $abb\diamond b\diamond cbb$ into three columns.

only the second column is uniform. The first and third columns contain different symbols on each row. An interesting observation to make is that the first column is $a\diamond c$, in addition you see that $a \uparrow \diamond$ and $\diamond \uparrow c$. Also, the third column is $b\diamond b$ which can be summarized as $b \uparrow \diamond$ and $\diamond \uparrow b$. In essence, the $\diamond$ has divided the columns into discrete sets, such that incompatible symbols are isolated from each other, and therefore we get another kind of pattern that is similar to, but not as strict as, a period. We call this kind of alignment a weak period.

An integer p satisfying $1 \leq p \leq n$ is a *period* of a partial word u of length n over A if $u(i) = u(j)$ whenever $i, j \in D(u)$ and $i \equiv j \bmod p$. In such a case, we call u *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*. Let us compare the partial word $abb\diamond bbabb$ with the partial word $abb\diamond bbcbb$, since these partial words provide us a good example to demonstrate the difference between a period and a weak period.

a	b	b
$\diamond$	b	b
a	b	b

a	b	b
$\diamond$	b	b
c	b	b

Figure 2.3: Comparison of a period and weak period.

Although the first column of both partial words contains a $\diamond$, the partial word $abb\diamond bbcbb$ is weakly 3-periodic but not 3-periodic, whereas the partial word $abb\diamond bbabb$ is weakly 3-periodic and 3-periodic. Yes, there is a $\diamond$ in the first column of our table for $abb\diamond bbabb$, but notice that the column begins with a, then is bisected with a $\diamond$, and the position after the $\diamond$ is also a. Since the symbol of the positions in this column does not change, this column does not contradict the definition of a period. Based on Figure 2.3, you may notice that weakly p-periodic full words are p-periodic, but we cannot say the same about partial words. Another difference worth noting is the fact that even if the length of a partial word u is a multiple of a weak period of u, then u is not necessarily a power of a shorter partial word. The smallest period of u is called its *minimal period* and is denoted by $p(u)$, and the smallest weak period of u is called its *minimal weak period* and is denoted by $p'(u)$. We denote the *set of all periods* of u by $\mathcal{P}(u)$ and the *set of all weak periods* of u by $\mathcal{P}'(u)$. We have that

$\{|u|\} \subset \mathcal{P}(u) \subset \mathcal{P}'(u) \subset \{1, \ldots, |u|\}$. In this context we say $X \subset Y$ if all elements in X are in Y.

We mentioned compatibility between symbols of an alphabet and this concept can be extended to partial words. Before we can determine compatibility, denoted $u \uparrow v$ where u and v are both partial words of the same length, we need to examine a couple of properties of those words. Foremost, we must ensure that u and v have the same length. If u and v are of equal length, then u is said to be contained in v, denoted by $u \subset v$, if $D(u) \subset D(v)$ or all elements in $D(u)$ are in $D(v)$ and $u(i) = v(i)$ for all $i \in D(u)$. Let us say we have two words $u_1 = a\diamond\diamond ab$ and $u_2 = ab\diamond ab$. Here, $u_1 \subset u_2$, but $u_2 \not\subset u_1$. Figure 2.4 demonstrates the concept of containment.

$$\begin{array}{cccccccc} u_1 & = & a & \diamond & \diamond & a & b \\ & & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ u_2 & = & a & b & \diamond & a & b \end{array} \qquad \begin{array}{cccccccc} u_2 & = & a & b & \diamond & a & b \\ & & \downarrow & \not\downarrow & \downarrow & \downarrow & \downarrow \\ u_1 & = & a & \diamond & \diamond & a & b \end{array}$$

Figure 2.4: Containment

The important thing to notice is that $u_2 \not\subset u_1$, since $D(u_2) \not\subset D(u_1)$. Contrast this with $u_1 \subset u_2$, where $D(u_1) \subset D(u_2)$ and $u_1(i) = u_2(i)$ for all $i \in D(u_1)$. Formally, we say that the partial words u and v are *compatible*, denoted by $u \uparrow v$, if there exists a partial word w such that $u \subset w$ and $v \subset w$. We call such a minimal w a least upper bound of u and v, denoted $u \vee v$. *The least upper bound* is such that $u \subset u \vee v$ and $v \subset u \vee v$ and $D(u \vee v) = D(u) \cup D(v)$.

Example 6 *We have two partial words, $u = ab\diamond bab$ and $v = a\diamond\diamond ba\diamond$. Is $u \uparrow v$? We see that $|u| = |v| = 6$. If the least upper bound of u and v, denoted $w = (u \vee v)$, exists, then let us find it.*

$$\begin{array}{cccccccc} u & = & a & b & \diamond & b & a & b \\ v & = & a & \diamond & \diamond & b & a & \diamond \\ \hline w & = & a & b & \diamond & b & a & b \end{array}$$

Notice that all three elements in each column are compatible with one another, thus we have found a w such that $w = (u \vee v)$. To check compatibility, we do not need to check for containment, but if we have found a w such that $u \subset w$ and $v \subset w$, then we can conclude that $u \uparrow v$.

2.3 Graphs and Trees

Graphs are computationally useful constructs which are particularly suitable in decision making processes. Let us look at a graph, G, which is defined by the set of its vertices, V, and the set of its edges, E, which connect one vertex to another. We define G as $G = (V, E)$. The edges of a graph can be directed, such that movement between nodes is restricted to one direction. If an edge between two nodes, r_0 and r_1 is undirected, then the possibility of two transitions exists, that is, the transition from $r_0 \rightarrow r_1$ or $r_0 \leftarrow r_1$. A sequence of transitions from node to node over some set of edges is called a *path*. A cycle is a path which ends where it begins. Not all graphs contain cycles, these graphs which contain no cycles are called *acyclic graphs*. Directed acyclic graphs are called *trees*. In the context of this research, a node is a vertex of a tree. Trees begin with a root node, which can contain any number of children. Each child, too, may have any number of children. Eventually, in a finite tree, we will encounter leaf nodes. *Leaf nodes* are terminal nodes which have no children. The graph in Figure 2.5 is a simple example of a tree.

Binary trees are trees in which each node has at most two children. Binary trees are especially useful in computer science because of the simplicity they offer in regards to traversal algorithms, and their suitability for recursion. In this binary configuration, left traversals represent a move to Child1, and right traversals represent a move to Child2. In succeeding chapters, left traversals will represent a different data relationship from a right traversal, but we choose not to go into those

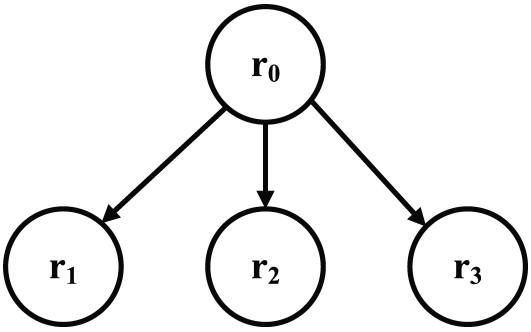

Figure 2.5: A Non-Binary Tree

specific details until later. Figure 2.6 is an example of a basic node structure within a binary tree. Superficially, the data we desire to represent in this thesis does not

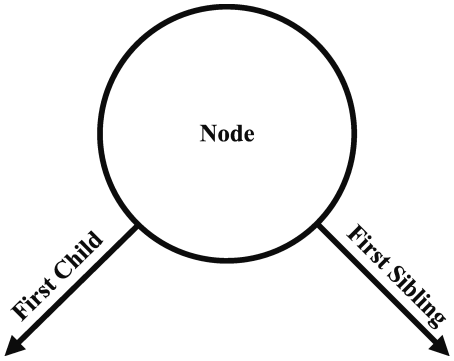

Figure 2.6: Basic Node Structure

adhere to a binary relationship, but a simple shift in the way we conceptualize a tree can lead us from the non-binary graph in our first figure to the binary graph in the second figure. In Figure 2.5 we have node r_0, which has three children, r_1, r_2, and r_3. In the binary graph such as the one shown in Figure 2.7, we would reserve the left pointer such that it points to one of the three children of our non-binary representation As you can see in Figure 2.7 we have chosen to set our pointer to r_1. The right pointer of r_0 remains null. Node r_1 has two siblings in Figure 2.5, and no

children. Thus, we set the left pointer of r_1 to null, and the right pointer now leads us to r_2. Similarly, the left pointer of r_2 is null, and the right pointer points to r_3. Node r_3 is the final child of r_0, and having no children of its own, both the left and right pointer are null in Figure 2.7 Using the binary representation in Figure 2.7,

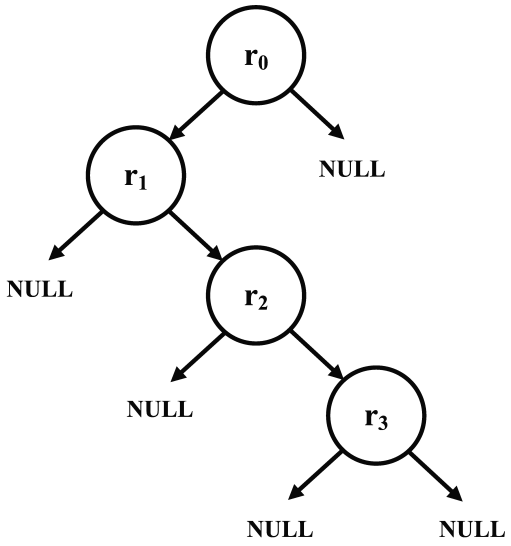

Figure 2.7: Binary Representation of a Tree

if we want to find all the direct descendants of r_0, we must access the left child of r_0, which is node r_1. Then we will discover the nodes which were siblings of r_1 in our non-binary tree (Figure 2.5) by following the right pointer of each node until we reach the null pointer of node r_3. This gives us $r_0 \rightarrow r_1 \rightarrow r_2 \rightarrow r_3 \rightarrow NULL$. We stop at NULL, and thus we have determined that the children of r_0 are r_1, r_2, and r_3.

CHAPTER III
NORMALIZATION ROUTINE

We should not make any assumptions about our partial word, u, nor should we make any assumptions about its alphabet, A. We must treat u as though each position was arbitrarily assigned a symbol from its alphabet. Normalization will distill u into its basic properties, at which point, a new non-minimal alphabet will be assigned. Here, we define the non-trivial alphabet $\{0, 1\}$ as minimal for any partial word. The resulting partial word u' will share the same properties as u, except that each $a \in A'$, where A' is the new alphabet, will no longer be arbitrarily assigned to positions in u'. Although the necessity of this step may not be immediately apparent, as we progress through the algorithm, you will see that assigning a meaning to the symbols of the alphabet is vital to discovering our final result.

Normalization is the process by which we convert partial words to a standardized form. For the purposes of normalization, we can describe any partial word, u, with the 4-tuple $(|u|, D(u), \boldsymbol{\mathcal{P}}(u), \boldsymbol{\mathcal{P}}'(u))$. This 4-tuple represents an $|u|$-length partial word independent of any specific alphabet. The original alphabet is discarded since it becomes irrelevant. Recall that $|u|$ is the length of partial word u and $D(u)$ is its domain. This leaves us to describe $\boldsymbol{\mathcal{P}}(u)$ and $\boldsymbol{\mathcal{P}}'(u)$, terms we have not previously encountered. The periods, $\mathcal{P}(u)$, and weak periods, $\mathcal{P}'(u)$, are essential to describe the behavior of a partial word u. However, notice that the list of all periods and weak periods contains redundant information, since $\{|u|\} \subset \mathcal{P}(u) \subset \mathcal{P}'(u)$. We use this information to trim the sets $\mathcal{P}(u)$ and $\mathcal{P}'(u)$ to their respective minimal

forms, $\boldsymbol{\mathcal{P}}(u)$ and $\boldsymbol{\mathcal{P}}'(u)$. Let us look at the first half of our relationship.

Example 7 *We can minimize the set $\mathcal{P}(u)$ using the relationship $\{|u|\} \subset \mathcal{P}(u)$. Notice that $|u|$ is a member of the 4-tuple which describes u. The maximum period of u is p_n, and we know $p_n = |u|$. Because $|u|$ is a period of every partial word u, we consider $|u|$ a trivial period. If we trim p_n from $\mathcal{P}(u)$, we get a new set, $\boldsymbol{\mathcal{P}}(u) = \mathcal{P}(u) \setminus \{p_n\}$. We can use $\boldsymbol{\mathcal{P}}(u)$ in our 4-tuple instead of $\mathcal{P}(u)$ without losing any information about u, since p_n is preserved by the $|u|$ member.*

Looking at this example, you see that a similar technique can be applied to $\mathcal{P}'(u)$ to minimize the number of relevant periods. Doing so will lead to a more concise description of u. Let us examine the second half of the relationship between periods and weak periods.

Example 8 *We can minimize the set $\mathcal{P}'(u)$ using the relationship $\mathcal{P}(u) \subset \mathcal{P}'(u)$. If we trim $\mathcal{P}(u)$ from $\mathcal{P}'(u)$, we get a new set, $\boldsymbol{\mathcal{P}}'(u) = \mathcal{P}'(u) \setminus \mathcal{P}(u)$. We can use $\boldsymbol{\mathcal{P}}'(u)$ to describe u instead of $\mathcal{P}'(u)$ without losing any information about u since the elements of $\mathcal{P}(u)$ which we removed from $\mathcal{P}'(u)$ to make $\boldsymbol{\mathcal{P}}'(u)$ are preserved by $|u|$ and $\boldsymbol{\mathcal{P}}(u)$ in our 4-tuple.*

The set $\boldsymbol{\mathcal{P}}(u)$ is composed of all non-trivial periods of u. We call this set the *strong periods* of u. We call $\boldsymbol{\mathcal{P}}'(u)$ the *strictly weak periods* of u since this set is composed entirely of weak periods and nothing else. Notice that $\boldsymbol{\mathcal{P}}(u) \cap \boldsymbol{\mathcal{P}}'(u) = \emptyset$, because we have guaranteed that these sets share no common members.

Our algorithm prepares for the normalization process by calculating ($|u|$, $D(u)$, $\boldsymbol{\mathcal{P}}(u)$, $\boldsymbol{\mathcal{P}}'(u)$) for the input, u. Our goal is to discover a partial word, u', over the non-minimal, augmented alphabet A', where u and u' are both represented by the same 4-tuple. We begin with a generic $|u|$-length partial word which is composed as follows ...

1. By default, each position of u' is assigned a unique symbol over A'.
   - $u' = a_0 a_1 \dots a_{n-1}$ where $a_j, a_k \ \in A'$ and $a_j \neq a_k$ for all $0 \leq j < k < |u|$.
2. The positions which are holes in u remain holes in u'.
   - We will enumerate the holes of u' such that ...
     - $\diamond_1$ is the first hole,
     - $\diamond_2$ is the second hole,
     - $\diamond_i$ is the ith hole, and so on.
     - The alphabet of u' becomes $A' \cup \{\diamond_1, \diamond_2, \dots, \diamond_N\}$ where N is the number of holes of u.

Example 9 *If we have $u = abca\diamond cacc$ with $A = \{a, b, c\}$, then the generic 9-length partial word we initially generate is $u' = abcd\diamond_1 efgh$, with $A' = \{a, b, c, d, e, f, g, h\}$.*

Normalization will re-assign symbols to the positions of this generic u', and in doing so, we will remove unused symbols from our new alphabet A'. This step by step process causes u' to evolve into its final form. You must be mindful that we intend to revise both u' and A' at each step based on each additional piece of information we process. Overall, normalization can be broken up into two distinct stages. The first stage enforces the strong periods of u upon u'. We re-assign symbols of u' so that for each $p \in \boldsymbol{\mathcal{P}}(u)$ we have $p \in \boldsymbol{\mathcal{P}}(u')$. In the second stage, we enforce the strictly weak periods of u upon u', such that for each $p \in \boldsymbol{\mathcal{P}}'(u)$ we have $p \in \boldsymbol{\mathcal{P}}'(u')$. Let us explain the details of how these two stages operate. For reasons which will be evident later, we need a flagging system to track which positions of u' have been assigned a symbol. We will underline these positions to flag them. By convention, we flag the first non-hole position in u'. Holes are never flagged because they remain $\diamond$'s throughout the entire normalization process. If we attempt to assign a symbol to a non-hole position, $u'(i)$, one of two situations will occur.

- If $u'(i)$ **is not** flagged, we assign position i with our selected symbol, b, and we underline the position to indicate that it is flagged. Here, $u'(i) = a$ becomes $u'(i) = \underline{b}$.

- If $u'(i)$ **is** flagged, we assign position i with our selected symbol, b. Let us assume that previously $u'(i) = \underline{a}$. The position $u'(i)$ remains flagged, but now instead of just changing $u'(i)$ to $\underline{b}$, we search the entire partial word u' and re-assign all positions having the value $\underline{a}$ with the new value $\underline{b}$.

Both stages of normalization adhere to this flagging system. However, the mechanism for assigning a value to a position varies in the two stages. Let us look at the first stage now.

3.1 Strong Periods of u

For each $p_i \in \boldsymbol{\mathcal{P}}(u)$ we place our u' into a 2-dimensional table such that the positions are aligned into p_i columns as shown in Figure 3.1.

$\boldsymbol{C_0}$	$\boldsymbol{C_1}$	...	$\boldsymbol{C_{p_i-1}}$
$u'(0p_i+0)$	$u'(0p_i+1)$	...	$u'(0p_i+p_i-1)$
$u'(1p_i+0)$	$u'(1p_i+1)$	...	$u'(1p_i+p_i-1)$
$u'(2p_i+0)$	$u'(2p_i+1)$	...	$u'(2p_i+p_i-1)$
$\vdots$	$\vdots$		$\vdots$

Figure 3.1: Our Table Structure

The positions in each column function as though they are *symbolically equivalent*. To ensure this characteristic is preserved in our final u' we must assign a symbol for each column which contains at least two non-hole positions. Our algorithm typically chooses the symbol of the first non-hole position in a given column as the representative symbol for that column. Then it assigns this representative symbol

to the rest of the positions in the column. Each position in the column is assigned a value according to the guidelines we previously outlined. If the algorithm encounters a position which is already flagged, then we know it has already been assigned a symbol, so now we must not only re-assign the representative symbol to this position, but we must perform a search and replace routine through the entire u' to ensure all such positions assigned with the old value are re-assigned with this new symbol. To demonstrate this idea, let us look at an example using $u = abaaba$. We transform this u into an initial $u' = abcdef$ noting that $\boldsymbol{\mathcal{P}}(u) = \{3, 5\}$.

- **Step 1**: Enforce $p_2 = 5$

C_0	C_1	C_2	C_3	C_4
$\underline{a}$	b	c	d	e
f				

C_0	C_1	C_2	C_3	C_4
$\underline{a}$	b	c	d	e
$\underline{a}$				

In this step, we have chosen symbol a for the first column, constructed from positions $u'(0)$ and $u'(5)$, because this column begins with a. Both positions are flagged when we assign this value to them. No other columns contain two or more symbols, so we do not consider them.

- **Step 2**: Enforce $p_1 = 3$

C_0	C_1	C_2
$\underline{a}$	b	c
d	e	$\underline{a}$

C_0	C_1	C_2
$\underline{c}$	$\underline{b}$	$\underline{c}$
$\underline{c}$	$\underline{b}$	$\underline{c}$

This step considers three columns. In the first column, we first encounter $u'(0) = \underline{a}$, thus $u'(0)$ and $u'(3)$ become $\underline{a}$. Next we consider the second column,

which begins with $u'(1) = b$. We choose to represent this column with b such that we assign $u'(1)$ and $u'(4)$ with $\underline{b}$. The third column begins with $u'(2) = c$, and we choose c to represent this column. However, when we encounter $u'(5) = \underline{a}$, we are forced to perform a search and replace operation whereby all positions $u'(i) = \underline{a}$, where $0 \leq i < |u'|$, are assigned $\underline{c}$.

- **Result:** Strictly for aesthetic purposes, we opt to replace the symbol $\underline{c}$ with the symbol $\underline{a}$ such that our alphabet $A' = \{a, b\}$. Therefore, we get $u' = \underline{abaaba}$.

3.2 Strictly Weak Periods of u

Just as we did with the strong periods, we now represent u' with a 2-dimensional table where the positions of u' are aligned into p'_i columns for each $p'_i \in \boldsymbol{\mathcal{P}}'(u)$. When dealing with strictly weak periods, the columns of our table will fall into one of two cases ...

Case 1 *There are no holes in the column.*

Case 2 *There are N holes in the column.*

In the first case, we can treat the column exactly as we did when dealing with strong periods. That is, the column is treated as one distinct set of symbolically equivalent symbols. A symbol is chosen to represent the column, then the process of assignment or re-assignment takes place on each of the positions. Fundamentally, the columns which contain at least one hole behave differently than columns with no holes in them when we are dealing with strictly weak periods. At least one column of our table will be broken up into $N+1$ sets, where N is the number of holes in the column. Instead of treating each column as one distinct symbolically equivalent set,

we must now treat the positions which are isolated by the $\diamond$'s as discrete symbolically equivalent sets. The first of these discrete sets is composed of all symbols above the first hole, next come all of the inner sets which are the positions between the i^{th} and $i^{th}+1$ hole, and the final set is composed of all positions below the N^{th} hole. Let us mention that any set of symbolically equivalent positions can be the empty set whenever two holes appear in a column without any positions in between them. Here is a simple demonstration of a column with three holes.

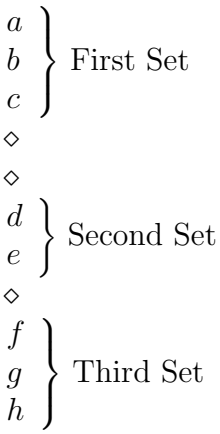

Figure 3.2: A Column With Three Symbolically Equivalent Sets

As you can see in Figure 3.2, the general idea of strictly weak period alignments is similar to that of strong periods, but we have to modify our symbol assignment technique to accommodate the discrete sets bounded by the holes which occur in at least one of the columns of our table. Now, instead of assigning a symbol to a column, we assign a symbol for each symbolically equivalent set (which may or may not span an entire column). The flagging and assignment rules remain intact. In other words, if we assign a symbol to an unflagged position, we must remember to flag it. If we attempt to assign a symbol to a flagged position, we re-assign its value, and we must also perform a search and replace to re-assign all flagged positions

which shared the original symbol. Let us look at an example using $u = abca\diamond cacc$. Initially $u' = abcd\diamond_1 efgh$. Observe that $\boldsymbol{\mathcal{P}}(u) = \emptyset$, therefore, u' remains $abcd\diamond efgh$ as we enter stage 2 of normalization. Here, $\boldsymbol{\mathcal{P}}'(u) = \{3\}$.

- **Step 1**: Enforce $p'_1 = 3$

$\boldsymbol{C_0}$	$\boldsymbol{C_1}$	$\boldsymbol{C_2}$
$\underline{a}$	b	c
d	$\diamond_1$	e
f	g	h

$\boldsymbol{C_0}$	$\boldsymbol{C_1}$	$\boldsymbol{C_2}$
$\underline{a}$	$\underline{b}$	$\underline{c}$
$\underline{a}$	$\diamond_1$	$\underline{c}$
$\underline{a}$	$\underline{d}$	$\underline{c}$

- **Result:** We have chosen to assign $u'(7) = \underline{d}$ instead of $\underline{g}$ as a matter of aesthetics. Thus, we get $u' = \underline{abca}\diamond_1\underline{cadc}$.

Once stage 2 of normalization is complete, the process as a whole is complete. Our goal was to discover a partial word u' over some augmented alphabet A' which shares the same 4-tuple description as our original input, u. Now that we have accomplished our goal, the flags used in this process can be discarded, as they were temporary flags which served only to track whether a position had previously been assigned a symbol. From this point forward, our algorithm will use u' for all of its processes. Thus, we may also discard the input u.

CHAPTER IV

THE RULE-TREE ALGORITHM

Our normalization process has left us with an augmented partial word, u', which is now in a standardized form. Normalization is a necessary preprocessing step because the symbols in the alphabet are assigned based on their behavior in relation to the periodicity of u'. In the Rule-Tree phase of our algorithm, we generate rules regarding the behavior of each symbol $a \in A'$.

4.1 Rules

The rules we generate fall into one of two categories. The first category of rules indicates that the binary values of two symbols are compatible, and this is written $(a \uparrow b)$ where $a, b \in A' \cup \{\diamond_1, \diamond_2, \ldots, \diamond_N\}$. On the other hand, we may wish to indicate that two symbols cannot have the same binary values, that is, they are incompatible, denoted as $(a \not\uparrow b)$ where $a, b \in A' \cup \{\diamond_1, \diamond_2, .., \diamond_N\}$. Recall that our ultimate goal is to discover some v over the alphabet $\{0, 1\} \cup \{\diamond\}$ such that every symbol in A' will eventually be assigned a binary value of 0 or a binary value of 1. We may also choose to assign any element in $\{\diamond_1, \diamond_2, \ldots, \diamond_N\}$ a binary value of 0 or 1, though a hole may remain a $\diamond$ in the final v if neither assignment, 0 nor 1, leads to a solution. We use a boldface font to denote the binary value of a symbol in A'. One of the properties of $\diamond$ is that it is compatible with every other symbol in the alphabet. Be aware that rules of the form $(\diamond_k \not\uparrow a)$ where $a \in A'$ and $0 < k \leq N$ are possible, but in such a case we know that $\diamond_k$ cannot remain a $\diamond$ in the final solution.

In binary relationships, $\diamond$ is compatible with both 0, 1, and $\diamond$, and thus we cannot say $\diamond \not\gamma 0$, $\diamond \not\gamma 1$, or $\diamond \not\gamma \diamond$. Let us look at the compatibility matrix for the binary alphabet $\{0, 1\} \cup \{\diamond\}$, shown here in Figure 4.1.

$\uparrow$	0	1	$\diamond$
0	True	False	True
1	False	True	True
$\diamond$	True	True	True

Figure 4.1: Compatibility Matrix For $\{0, 1\} \cup \{\diamond\}$

4.2 Candidate Weak Period Enumeration

Recall this figure, which we covered in our chapter on Normalization ...

$\boldsymbol{C_0}$	$\boldsymbol{C_1}$	...	$\boldsymbol{C_{p_i-1}}$
$u'(0p_i + 0)$	$u'(0p_i + 1)$	...	$u'(0p_i + p_i - 1)$
$u'(1p_i + 0)$	$u'(1p_i + 1)$	...	$u'(1p_i + p_i - 1)$
$u'(2p_i + 0)$	$u'(2p_i + 1)$	...	$u'(2p_i + p_i - 1)$
$\vdots$	$\vdots$		$\vdots$

Figure 4.2: p_i-Column Alignment Of u'

If we want to test whether an integer, p_i, is a weak period of partial word u, we align u into p_i columns and inspect each column to ensure it is composed of discrete, uniform sets divided by $\diamond$'s. We call this table a *p_i-column table*. Now we want to introduce a methodology to enumerate the plausible values of p_i, also called the *candidate weak periods* of partial word u. Let us first generate the 3-column table of an anonymous partial word, u, of length 8. Here we are testing if $p_i = 3 \in \mathcal{P}'(u)$, as shown in Figure 4.3.

C_0	C_1	C_2
$u'(0)$	$u'(1)$	$u'(2)$
$u'(3)$	$u'(4)$	$u'(5)$
$u'(6)$	$u'(7)$	

Figure 4.3: 3-Column Alignment Of An Anonymous u

To ensure that 3 is a weak period of u, we would test $u'(0) \uparrow u'(3)$, $u'(3) \uparrow u'(6)$, $u'(1) \uparrow u'(4)$, $u'(4) \uparrow u'(7)$, and $u'(2) \uparrow u'(5)$. Contrast this with a candidate weak period enumeration table, described next, which as the name implies, is a method to enumerate all possible values of p_i, where $0 < p_i \leq |u|$. Each row of this table should check the same compatibilities that our p_i-column method would test. Figure 4.4 is a generalized reference showing how we build such a table.

Row	p_i	$u'(0)$	$u'(1)$	$u'(2)$	...	$u'(n-2)$	$u'(n-1)$
1	$n-1$	$u'(n-1)$					
2	$n-2$	$u'(n-2)$	$u'(n-1)$				
3	$n-3$	$u'(n-3)$	$u'(n-2)$	$u'(n-1)$			
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$			
$n-1$	1	$u'(1)$	$u'(2)$	$u'(3)$	...	$u'(n-1)$	
n	0	$u'(0)$	$u'(1)$	$u'(2)$	...	$u'(n-2)$	$u'(n-1)$

Figure 4.4: p_i-Enumeration Table

Now if we specifically look at the p_i-enumeration table for our anonymous partial word u of length 8, shown in Figure 4.5. The topmost row of the table in Figure 4.5 is the column header row. The first two columns, labeled Row and p_i, provide auxiliary information identifying the row number and corresponding candidate weak period (p_i) of each row. Each subsequent column in this header, labeled $u'(0), u'(1), ..., u'(7)$ in this example, indicates a position in u. Below the column header are rows

containing the proper suffixes of u, starting with the proper suffix of length 1 and ending with the proper suffix of length $|u| - 1$.

Row	p_i	$u'(0)$	$u'(1)$	$u'(2)$	$u'(3)$	$u'(4)$	$u'(5)$	$u'(6)$	$u'(7)$
1	7	$u'(7)$							
2	6	$u'(6)$	$u'(7)$						
3	5	$u'(5)$	$u'(6)$	$u'(7)$					
4	4	$u'(4)$	$u'(5)$	$u'(6)$	$u'(7)$				
5	3	$u'(3)$	$u'(4)$	$u'(5)$	$u'(6)$	$u'(7)$			
⋮	⋮	⋮							

Figure 4.5: p_i-Enumeration Table For u

Example 10 *If $u = abcd$, the set of proper suffixes of length 1 to $|u| - 1$, ordered from shortest length to longest, would be $\{d, cd, bcd\}$.*

We are dealing with weak periods, and the information we yield from this p_i-enumeration table will be compatibility rules. For the moment, we are concerned only with rows that correspond to weak periods of u.

Let us assume that 7 and 3 are weak periods of this anonymous 8-length u. Row 1 of the table in Figure 4.5 corresponds to the weak period 7. This row contains $u'(7)$, the proper suffix of u having a length of 1. This suffix has only one single position, $u'(7)$, and if you look at the column head under which position $u'(7)$ is aligned, you will see that this column is labeled with $u'(0)$. Because $u'(7)$ aligns with $u'(0)$, we construct the rule $(u'(0) \uparrow u'(7))$ to represent this relationship.

We have completed our inspection of this row, so now we move to Row 5 since it corresponds to the weak period 3 of our anonymous u. The proper suffix placed in this row is $u'(3)u'(4)u'(5)u'(6)u'(7)$. Here, $u'(3)$ falls in the column labeled $u'(0)$, $u'(4)$ falls in the column labeled $u'(1)$, $u'(5)$ falls in the column labeled $u'(3)$, $u'(6)$ falls in the column labeled $u'(4)$, and $u'(7)$ falls in the column labeled $u'(4)$.

Thus, we generate the set of rules $\{(u'(3) \uparrow u'(0)), (u'(4) \uparrow u'(1)), (u'(5) \uparrow u'(2)), (u'(6) \uparrow u'(3)), (u'(7) \uparrow u'(4))\}$.

If we compare this result to the results we got from the p_i-column method, we still get the same set of rules, albeit the rules are generated in differing orders, but this is a trivial difference. Having demonstrated the equivalence between the weak period enumeration table and the p_i-column alignments, let us note that from this point on, we choose to use the weak period enumeration table for partial words.

Now that you understand what information the p_i-enumeration tables generate, you may be asking some questions. Where do rules come from? How are they generated? How are rules and p_i-enumeration tables related? Rules act as guidelines, telling us how to assign 0's and 1's to symbols in u' such that we end up with a v with the same periods, weak periods, and a subset of the holes of u. So, if we know that $(a \not\gamma b)$ in u', then we know that $\mathbf{a} \not\gamma \mathbf{b}$ in v, meaning the binary value of a is not compatible with the binary value of b. Let us visualize this concept with an example.

Example 11 *Let us look at the partial word* $u' = ab\diamond_1 abc$. *If* $5 \in \mathcal{P}(u')$, *then we should be able to align* u' *into a table with 5 columns as such ...*

0	1	2	3	4
a	b	$\diamond_1$	a	b
c				

However, in the first column, you see that $u'(0) \not\gamma u'(5)$, *since* $a \not\gamma c$. *If we allowed* $\boldsymbol{u'(0) \uparrow u'(5)}$, *then* $v(0) \uparrow v(5)$ *in our final solution* v *and* 5 *would be a period of* v. *This must be avoided. To exclude* 5 *as a weak period of* v, *we might select a* $v = 00\diamond 001$ *such that we generate the following alignment ...*

0	1	2	3	4
0	0	$\diamond$	0	0
1				

Since $v(0) = 0$ *and* $v(5) = 1$, $v(0) \not\uparrow v(5)$. *We have carefully selected binary values for these two positions which prohibit* v *from being aligned into* 5 *columns.*

In a normalized partial word, you must consider all positions represented with the same symbol as an atomic set. Ultimately, we are trying to find equivalences amongst these sets of positions, separating or combining them until we have reached a state in which only two sets are left, corresponding to 0 and 1. When we encounter a compatibility rule, $(a \uparrow b)$, this implies that all positions corresponding to symbol a are compatible with all positions corresponding to symbol b. Incompatibility implies that all positions corresponding to symbol a are incompatible with all positions corresponding to symbol b. We begin to look past individual rules now, and start looking at the rules as sets. We will build a p_i-enumeration table with $|u| - 1$ rows, from which we can infer information about v, our output over $\{0, 1\} \cup \{\diamond\}$. For each i, Row_i relates to an alignment of u' as if $|u| - i$ were a weak period of u'. We compare each row to some weak period of u, and if indeed $|u| - i$ is an element of $\mathcal{P}'(u')$, then the rules generated on that row are all compatibility rules of the form $(a \uparrow b)$ where $a, b \in A' \cup \{\diamond_1, \diamond_2, \ldots, \diamond_N\}$. Otherwise, the rules generated will all be incompatibility rules of the form $(a \not\uparrow b)$ where $a, b \in A' \cup \{\diamond_1, \diamond_2, \ldots, \diamond_N\}$. Our table is built such that the first row, Row_0 is empty. The second row, Row_1 will be composed of one position, the suffix of u' of length 1 which is the last position in u'. The third row, Row_2 is composed of the suffix of u' of length 2 such that it is the last two positions in u'. This continues until the final row which is composed of the substring corresponding to the last $|u| - 1$ positions of u'.

Example 12 *This is the p_i-enumeration table for $u' = ab\diamond_1 abc$, the partial word we introduced in Example 11 ...*

i	a	b	$\diamond_1$	a	b	c
0						
1	c					
2	b	c				
3	a	b	c			
4	$\diamond_1$	a	b	c		
5	b	$\diamond_1$	a	b	c	

Each row of the table is examined independently of the other rows. First we check to see if $(|u| - i) \in \mathcal{P}'(u)$. If $(|u| - i) \notin \mathcal{P}'(u)$, then we want to build incompatibility rules. Otherwise, we want to build compatibility rules. The first row, Row_0 is trivial, and may be ignored. In Row_1, only one column is populated, the $u'(0)$ column. The element which populates it, $u'(|u| - 1)$ or c, implies that $(a \not\gamma c)$. Thus, Row_1 generated only one rule. In fact, this row will always generate at most one rule. It is plausible however that no rule is generated. This situation would occur if $|u| - i$, 5 for this example, is a weak period of u. In the second row two rules are generated, $(a \not\gamma b)$ and $(b \not\gamma c)$. In order to prevent a specific alignment from corresponding to a weak period, we only need to consider one of these possibilities.

Row_2, in the top table, corresponds to a weak period of 4. Since $4 \notin \mathcal{P}'(u)$, we must ensure that $4 \notin \mathcal{P}'(v)$. Notice in the lower two tables of Figure 4.6 corresponding to v over $\{0, 1\}$, it is sufficient to consider only one possibility, either $v(0) \not\gamma v(4)$ or $v(1) \not\gamma v(5)$. It is too early to determine which of these possibilities to choose. Therefore, at this time, we will not discard either.

a	b	$\diamond_1$	a
b	c		

0	0	$\diamond_1$	0
1	0		

0	**0**	$\diamond_1$	0
0	**1**		

Figure 4.6: Two Options To Prevent A Period Of 4

We use $(|u| - i)$ to determine which row corresponds to the two periods of u', where $\mathcal{P}'(u') = \{3, 6\}$. For $u = ab\diamond_1 abc$, $|u| = 6$ and we have a weak period 3. Thus we use $Row_{(|u|-i)}$ to calculate that Row_3 corresponds to the weak period 3. In order to preserve 3 as a weak period in our solution v, we have to make sure that each column generates a compatibility rule. Notice that $(a \uparrow a)$ and $(b \uparrow b)$ are trivial rules which are always true. They do not give us any useful information, thus we drop them. However, the final rule generated on this row, $(\diamond_1 \uparrow c)$ is preserved. If it were not for the fact that the two former rules are trivial, this row could have potentially generated 3 rules; we would have no choice except to observe all three rules.

a	b	$\diamond_1$
a	b	c

0	1	$\diamond$
0	1	1

Figure 4.7: Enforce Weak Period 3

Row_3 corresponds to a weak period of 3. Since $3 \in \mathcal{P}'(u)$, we must ensure that $3 \in \mathcal{P}'(v)$. Here we see that $u'(0) \uparrow u'(3)$ and $u'(1) \uparrow u'(4)$ and $u'(2) \uparrow u'(5)$

indicating that $v(0) \uparrow v(3)$ and $v(1) \uparrow v(4)$ and $v(2) \uparrow v(5)$ in our final solution. Here, we must observe all of these rules, which is a striking difference compared to the way we handled rules generated in Row_2.

4.3 AND-sets And OR-sets

Example 12 illustrates two metrics for weak period enforcement. The first metric occurs when a row does not correspond to a weak period. We know that at least one column of the row must be incompatible with the corresponding position in u'. We call these rows OR-sets. The second metric covers all other rows, the ones which do correspond to a weak period. This situation requires that all positions in the row be compatible with the positions of u' which fall in the same column. We call these rows AND-sets. If we can ensure that v, our result, observes all rules in our AND-set, then we guarantee that $\mathcal{P}(u) \subset \mathcal{P}(v)$ and $\mathcal{P}'(u) \subset \mathcal{P}'(v)$. In addition, if we can discover a v which observes at least one rule from each of the OR-sets, then we can ensure that $\mathcal{P}(v) \subset \mathcal{P}(u)$ and $\mathcal{P}'(v) \subset \mathcal{P}'(u)$. Combining these concepts, we form a tight bound on the periodicity of v, thus ensuring that $\mathcal{P}(v) = \mathcal{P}(u)$ and $\mathcal{P}'(v) = \mathcal{P}'(u)$. Later, we will discuss how to satisfy our final criterion that $H(v) \subset H(u)$, but for now, we need to elaborate more on how we intend to use the information contained in our p_i-enumeration table.

We intend to convert our p_i-enumeration table to a binary tree. This is because algorithms which operate on binary trees are inherently simple. Our binary tree will represent all potentially feasible solutions, so eventually we must search the various paths until we discover one which produces the expected result. Let us assume we have already created our enumeration table. Instead of having a column labeled with Row, we will now label that column with i. Additionally, we have two more columns to consider for each table. The first new column, we label "**AND** or

OR", and we use this to denote whether a row corresponds to an AND-set or to an OR-set. The second new column is labeled $\boldsymbol{S_i}$, and in this column, we list the set of all rules generated for Row_i. Each Row_i 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 $u'(j) \not\gamma u'[|u|-i..|u|)(j)$, 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 Row_i. If $|u| - i \notin \mathcal{P}'(u)$, then S_i is called an OR-set, otherwise it is called an AND-set.

Example 13 *Let us look at the partial word $u = ab\diamond b\diamond bcb$. For this partial word, $\mathcal{P}(u) = \{4, 8\}$ and $\mathcal{P}'(u) = \{2, 4, 8\}$. Normalization results in $u' = ab\diamond_1 b\diamond_2 bcb$.*

Alignment								$\boldsymbol{i}$	**AND** *or* **OR**	$\boldsymbol{S_i}$
a	b	$\diamond_1$	b	$\diamond_2$	b	c	b	*0*		
b								*1*	*OR*	$\{(a \not\gamma b)\}$
c	b							*2*	*OR*	$\{(a \not\gamma c)\}$
b	c	b						*3*	*OR*	$\{(a \not\gamma b), (b \not\gamma c), (\diamond_1 \not\gamma b)\}$
$\diamond_2$	b	c	b					*4*	*AND*	$\{(a \uparrow \diamond_2), (\diamond_1 \uparrow c)\}$
b	$\diamond_2$	b	c	b				*5*	*OR*	$\{(a \not\gamma b), (b \not\gamma \diamond_2), (\diamond_1 \not\gamma b), (b \not\gamma c)\}$
$\diamond_1$	b	$\diamond_2$	b	c	b			*6*	*AND*	$\{(a \uparrow \diamond_1), (\diamond_1 \uparrow \diamond_2), (\diamond_2 \uparrow c)\}$
b	$\diamond_1$	b	$\diamond_2$	b	c	b		*7*	*OR*	$\{(a \not\gamma b), (b \not\gamma \diamond_1), (b \not\gamma \diamond_2), (b \not\gamma c)\}$

In order to build a tree efficiently, and to minimize the size of our tree which can become quite extensive in size, it is beneficial for us to perform some reductions on the OR-Set rules. For instance, if a row contains only one valid rule, then that rule will occur in every path of the tree. We call these types of rules *definitive rules*. We can safely eliminate other rows of our table which contain these definitive rules, because in essence the definitive rule guarantees that we have selected a rule

from these other rows. This is also beneficial to us because it may also eliminate contradictions which we would otherwise be helpless to resolve.

Example 14 *Consider Example 13 above in which we have two definitive rules* $(a \not\gamma b)$ *and* $(a \not\gamma c)$*, generated for* $i = 1$ *and* $i = 2$*, respectively. If the rule* $(b \not\gamma c)$ *were to occur in some other row, it would lead to a contradiction. These three rules,* $\{(a \not\gamma b),(b \not\gamma c),(a \not\gamma c)\}$*, form a negative cycle.*

Let us assume we assign a the binary value 0*, then b must be* 1 *in order to satisfy the rule* $(a \not\gamma b)$*. Since b is assigned the value* 1*, then c would have to be assigned* 0 *to satisfy the second rule,* $(b \not\gamma c)$*. At this point, we reach a contradiction if we try to observe the third rule,* $(a \not\gamma c)$*, since* $a = 0$ *and* $c = 0$*, and thus we cannot accommodate* $(a \not\gamma c)$*.*

Alignment								i	**AND** or **OR**	S_i
a	b	$\diamond_1$	b	$\diamond_2$	b	c	b	0		
b								1	OR	$\{(a \not\gamma b)\}$
c	b							2	OR	$\{(a \not\gamma c)\}$
b	c	b						3	OR	
$\diamond_2$	b	c	b					4	AND	$\{(a \uparrow \diamond_2), (\diamond_1 \uparrow c)\}$
b	$\diamond_2$	b	c	b				5	OR	
$\diamond_1$	b	$\diamond_2$	b	c	b			6	AND	$\{(a \uparrow \diamond_1), (\diamond_1 \uparrow \diamond_2), (\diamond_2 \uparrow c)\}$
b	$\diamond_1$	b	$\diamond_2$	b	c	b		7	OR	

Figure 4.8: Reduced p_i-Enumeration Table

Visually, these types of contradictions can be easy to spot, but algorithmically, the time complexity required to discover negative cycles can be prohibitive. Therefore, this kind of contradiction often goes undiscovered until after our tree is generated. Notice however, that the rows corresponding to the i values 3, 5, and 7 can be ignored because they each duplicate the definitive rule $(a \not\gamma b)$, and in doing

so, we avoid dealing with the rule ($b \not\gamma c$). So, often times, using definitive rules to eliminate redundant information saves us from costly computations related to detecting negative cycles. The reduced table, with rows containing definitive rules removed, is shown here in Figure 4.8.

4.4 Binary Rule-Tree

Within a p_i-enumeration table, our rules are generated in a hierarchical structure, ordered from the lowest i-value to the highest, $0 \leq i < |u|$. Later in our algorithm, we want to choose one rule from each row correlating to an OR-set. This approach does not guarantee that every combination of rules will lead to a valid output whenever such output exists. At best we can say that if a valid output exists, then at least one combination of rules is acceptable. If we choose an unacceptable collection of rules, we will need some structured technique to guide us in making our next choice in such a way that we avoid repeatedly selecting the same collection of rules. The structure we use in our algorithm is a binary tree, which is both computationally efficient and algorithmically elegant.

Member	**Data type**	**Accessor Function**
value	Rule	ValueOf($Node_j$)
i	Integer	$i(Node_j)$
Child1	Node-pointer	Child1($Node_j$)
Child2	Node-pointer	Child2($Node_j$)

Figure 4.9: Node Data Members

The binary tree is a structure composed of nodes, each of which possesses at most two edges leading to its children. The nodes of our adaptation of the tree contain four important pieces of information, *value*, i, Child1, and Child2, which

we intend to describe now. Figure 4.9 summarizes these data members. The value of a node is a rule, and with the exception of our root node which we will discuss later, the value of a node will be a rule from some row of our p_i-enumeration table. The i-value of a node is equivalent to the i-value of the row from our table in which its value, a rule, resides. The i-value of a node guides us in its placement within the tree. The final two data members of our node structure are pointers ... think of these as the edges of our tree. Each node has two pointers, Child1 and Child2, which are null by default. A null pointer is a non-existent edge. In other words, in a binary tree, a node can have at most two children, but if a node has no children or only one child, then it will have no outbound edges or one outbound edge respectively. Similarly, our node can have at most two pointers, but if there are no children, then both child pointers are null, and if there is only one child, then one of our two pointers will be null. The Child1 and Child2 pointers are not equivalent, and the subtrees to which they lead us indicate two different relationships.

The Child1 pointer directs us to the node which is the root of a subtree for which the i-value of all its members is strictly greater than the value of our node. In the context of our p_i-enumeration table, the Child1 pointer leads us to the next row of our table correlating to an OR-set.

The Child2 pointer leads us to a node which is analogous to a sibling. That is, it points to a node which has an equivalent i-value because it is part of the same OR-set and thus it was generated on the same row of our p_i-enumeration table. The subgraph rooted at the node to which our Child2 pointer directs us is parallel to, though not necessarily equivalent to, the subgraph to which Child1 points.

Let us also note that each data member of our node has an accessor function. The accessor function is used to access the value of a data member. If we want to know the value of a node, $Node_2$, whose value is $(a \parr b)$, we would denote this using

ValueOf($Node_2$) = ($a \not\gamma b$). Similarly we have a function to access the i-value of a node, it is $i(Node_j)$. The child pointers of a node are accessed with the functions Child1($Node_j$) and Child2($Node_j$).

Our tree begins its life as a single node, the root node. For the purposes of our C++ implementation, we use a dummy node as our root. This node, labeled $Node_{root}$, is initialized with an i-value of 0, to denote its depth as 0, and its pointers are initialized to null. It is assumed that the root node is assigned a unique value, and this value will not match the value of any other rule in the tree. We have chosen to use a dummy root node because we are not guaranteed that any Row_i will result in a valid rule. This is because any row, including Row_1, could correspond to a weak period which would produce an AND-set, thus it would not appear in our tree. So, for instance if $|u|-1$ is a weak period of u, Row_1 would not contribute a node to the tree. We would move to Row_2, a row which could potentially generate two rules, and if that is the case, there would be two entry points into our tree unless we have a dummy root. We want to avoid multiple entry points, since this complicates tree assembly and traversal algorithms.

We start at Row_1 of our enumeration table, which will produce at most one rule. Let us assume that this row produces an OR-set. Thus our rule will be inserted into the tree. We try to insert the new node, $Node_0$ at our root node, which has an i-value of 0, and two null pointers, Child1 = $null$ and Child2 = $null$. We compare the i-values of the root node with the i-value of our new node and discover that the new node has a higher value, $i(Node_{root}) < i(Node_0)$, thus we look at the Child1 pointer of the root node, which is null. The null pointer indicates that we have found the proper home for $Node_0$, so we set the Child1 pointer of our root node such that it points to $Node_0$. This is denoted by Child1($Node_{root}$) $\leftarrow Node_0$.

Looking at our pseudo-code in Figure 4.4, you see that there are two main

cases to consider. In the first case, NewNode and InsertAt both have the same i-value. This means that NewNode belongs in the parallel subtree of InsertAt. If Child2 of InsertAt is null, then NewNode is the first node of its subtree. Otherwise, we repeat the insertion process, following the Child2 links of nodes until we discover a node whose Child2 pointer is null, and that is where we insert NewNode. Of course, any time we encounter a situation where the value and i-value of NewNode and the value of InsertAt are equivalent, we abort the attempt to insert the node NewNode along this path of our tree.

The second case is a bit more complicated. First of all, this case occurs when the i-value of NewNode is greater than the i-value of InsertAt. In general, this means that NewNode belongs in both subtrees of InsertAt. If InsertAt has no parallel subtree, Child2 is null, then we only need to consider placing NewNode in the direct subtree, which is pointed to by Child1. Notice that we make one additional check before we allow NewNode to be inserted in the direct subtree. In order to prevent redundancy, we check to make sure that the value of InsertAt is different from the value of NewNode. In this way, we ensure that no rule occurs more than once in any given path of our tree.

```
Insert(Node NewNode,Node InsertAt)
    If NewNode And InsertAt Have The Same Value Then
        // Do Not Insert NewNode Into The Tree
        Exit Function
    End If

    // Case One:  The Two Nodes Have The Same i Value
    // Thus NewNode Is A Sibling Of InsertAt
    If InsertAt And NewNode Have The Same i Value Then
        If Child2 Of InsertAt Is Null Then
            // NewNode Is The First Sibling Of InsertAt
            Child2(InsertAt) ← NewNode
        Else
            // Make NewNode As A Sibling Of InsertAt
            Insert(NewNode, Child2(InsertAt))
        End If
    Else
        // Case Two:  NewNode Has A Higher i Value Than InsertAt
        If Child1 Of InsertAt Is Null Then
            // NewNode Is The First Child Of InsertAt
            Child1(InsertAt) ← NewNode
        Else
            // NewNode Belongs In The Child1 Subtree Of InsertAt
            Insert(NewNode, Child1(InsertAt))
        End If
        // NewNode May Be A Child Of The Siblings Of InsertAt
        If InsertAt Has Siblings Then
            Insert(NewNode, Child2(InsertAt))
        End If
    End If
End Function
```

Figure 4.10: Node Insertion Routine

In order to demonstrate this algorithm, let us look at the example of $u = ab\diamond_1\diamond_2cd$ where $\mathcal{P}(u) = \{6\}$ and$\mathcal{P}'(u) = \{2, 6\}$. After exiting the Normalization routine, $u' = ab\diamond_1\diamond_2cd$. The table we generate would look like Figure 4.11.

Alignment	i	**AND** or **OR**	S_i
$a \quad b \quad \diamond_1 \quad \diamond_2 \quad c \quad d$	0		
d	1	OR	$\{(a \not\gamma d)\}$
$c \quad d$	2	OR	$\{(a \not\gamma c), (b \not\gamma d)\}$
$\diamond_2 \quad c \quad d$	3	OR	$\{(a \not\gamma \diamond_2), (b \not\gamma c), (\diamond_1 \not\gamma d)\}$
$\diamond_1 \quad \diamond_2 \quad c \quad d$	4	AND	$\{(a \uparrow \diamond_1), (b \uparrow \diamond_2), (\diamond_1 \uparrow c), (\diamond_2 \uparrow d)\}$
$b \quad \diamond_1 \quad \diamond_2 \quad c \quad d$	5	OR	$\{(a \not\gamma b), (b \not\gamma \diamond_1), (\diamond_1 \not\gamma \diamond_2), (\diamond_2 \not\gamma c), (c \not\gamma d)\}$

Figure 4.11: Weak Period Enumeration Table For ab$\diamond\diamond$cd

The possible paths through the tree are shown here in Figure 4.12.

Path 1	(root)	$(a \not\gamma d)$	$(a \not\gamma c)$	$(a \not\gamma \diamond_2)$	$(a \not\gamma b)$	01$\diamond$111
Path 2	(root)	$(a \not\gamma d)$	$(a \not\gamma c)$	$(a \not\gamma \diamond_2)$	$(b \not\gamma \diamond_1)$	Impossible
Path 3	(root)	$(a \not\gamma d)$	$(a \not\gamma c)$	$(a \not\gamma \diamond_2)$	$(\diamond_1 \not\gamma \diamond_2)$	Impossible
Path 4	(root)	$(a \not\gamma d)$	$(a \not\gamma c)$	$(a \not\gamma \diamond_2)$	$(\diamond_2 \not\gamma c)$	Impossible
Path 5	(root)	$(a \not\gamma d)$	$(a \not\gamma c)$	$(a \not\gamma \diamond_2)$	$(c \not\gamma d)$	Impossible
Path 6	(root)	$(a \not\gamma d)$	$(a \not\gamma c)$	$(b \not\gamma c)$	$(a \not\gamma b)$	Impossible
$\vdots$						$\vdots$

Figure 4.12: Paths Through Rule Tree Of ab$\diamond\diamond$cd

Let us build a tree to represent this same information. You will see that the tree is much easier to read. Figure 4.13 gives you an idea of the structure and size of the tree we have built for the partial word, ab$\diamond\diamond$cd.

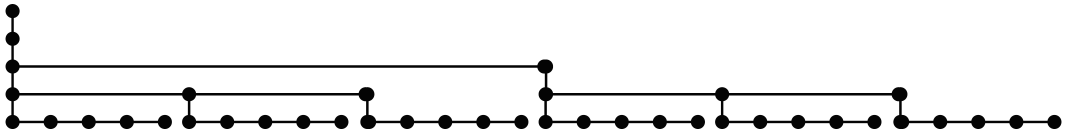

Figure 4.13: Tree Overview

In the succeeding figures, we will follow the most basic traversal of our tree stopping at each stage for a close up of the structure.

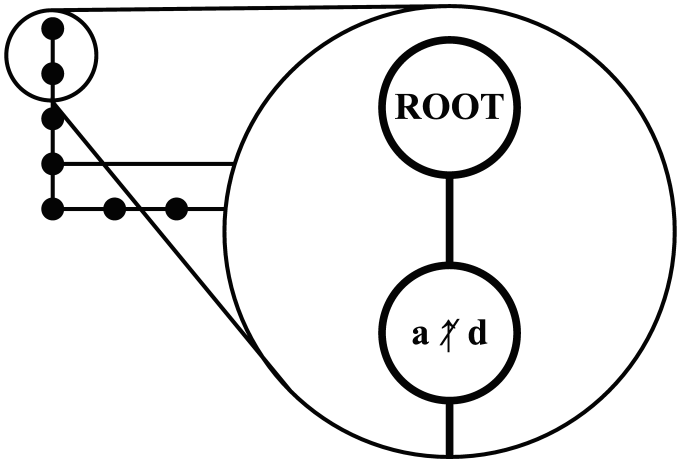

Figure 4.14: Rule Tree For $ab\diamond_1\diamond_2cd$, Close-Up Of Step One

Figure 4.14 shows us the root of our tree. The root is a dummy node, which exists solely for the purpose of providing a single entry point into our tree. The next node we encounter is $(a \parr d)$.

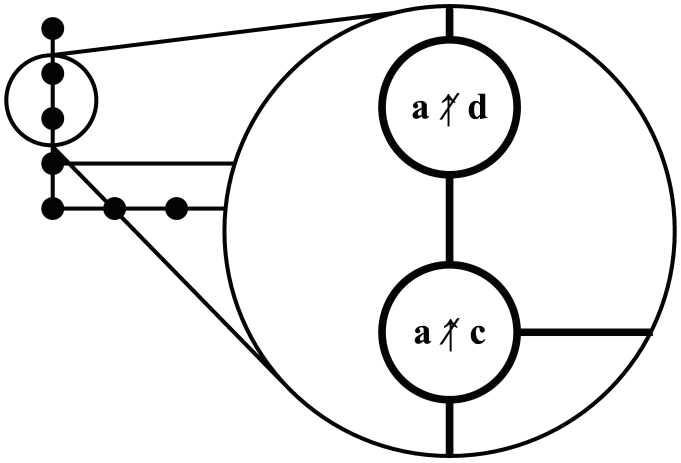

Figure 4.15: Rule Tree For $ab\diamond_1\diamond_2cd$, Close-Up Of Step Two

Figure 4.15 shows us that a move down the this path leads us to our next rule $(a \not\curlyvee c)$.

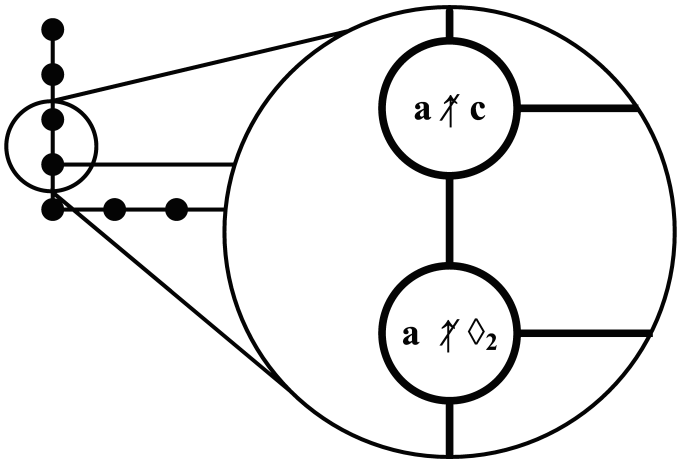

Figure 4.16: Rule Tree For $ab\diamond_1\diamond_2cd$, Close-Up Of Step Three

In Figure 4.16 we encounter the rule $(a \not\curlyvee \diamond_2)$.

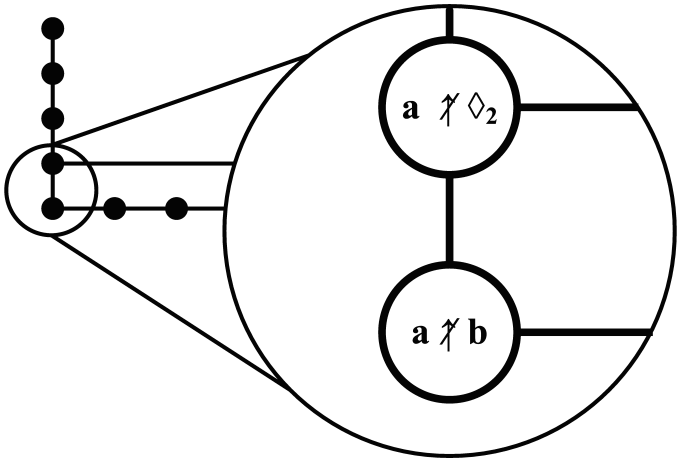

Figure 4.17: Rule Tree For $ab\diamond_1\diamond_2cd$, Close-Up Of Step Four

Finally in Figure 4.17 we have completed our path, which ends at $(a \not\curlyvee b)$. If we observe the rules in our AND-set, $(a \uparrow \diamond_1)$, $(b \uparrow \diamond_2)$, $(c \uparrow \diamond_1)$, $(d \uparrow \diamond_2)$, as well as the rules we encountered along our path, we would conclude that $01\diamond111$ is a solution for $ab\diamond\diamond cd$.

CHAPTER V

TREE TRAVERSAL AND ASSIGNMENT GRAPHS

If a solution, v, exists which meets our expectations, then that solution lies in one of the branches of our rule tree. Therefore, we intend to extract information about v from the rule tree. In order to do this, we must traverse at least one of the various paths of the tree. In the best case scenario, the first path of the tree we choose will lead us to an answer. However, in the worst case, we might have to traverse all paths of the tree before we can determine an acceptable solution. Each path is a set of OR rules, which are rules specifying incompatible symbols, and this set will potentially lead us to a feasible solution. First we apply AND rules, and then we apply the rules along a path to the alphabet of u'. If no problems arise, then we have found a solution. Otherwise, we choose a different path, and make another attempt to discover a solution.

When we talk about a path in the rule tree, we are talking about a sequence of nodes starting from the root of our tree and ending at a terminal node. Here, we define a terminal node as a node in which the Child1 pointer is null. This modified definition is necessary because our binary tree is representing information which is not binary in nature. In this case, a dummy node serves as the root. The various paths in our rule tree do not all have the same number of nodes, the paths are not uniform because we choose to exclude duplicate values along a path. As we built our tree, we did not insert a rule if it was equivalent to one of its predecessors, however the rule might have been inserted in some other paths, and therefore those paths would have more nodes. Because of this, the size of our tree is quite unpredictable.

In general, longer words with larger alphabets and fewer weak periods produce larger trees. However, it is plausible that fairly short words with concise alphabets might still generate trees with many millions of paths. The primary reason for this is that the paths of our tree are not guaranteed to be unique, so multiple paths might actually be equivalent, that is, they may contain the exact same set of rules. Other implementations of the tree structure could avoid the problem of multiple equivalent paths, but at this time we have chosen to forgo such an implementation.

In this algorithm, rules are applied to the alphabet of u', which is the normalized word we produced in the preprocessing phase of our algorithm. The alphabet of u' is A', and we intend to group the symbols of this alphabet into two sets correlating to 0 and 1. In addition, we have the set of holes, $\{\diamond_1, \diamond_2, \ldots, \diamond_N\}$, each of which we will assign to 0, 1, or $\diamond$. By default, we will assume that each $\diamond$ functions as either a 0 or a 1. We reserve $\diamond$ in v for situations where neither 0 nor 1 are possible.

5.1 Vertex Structure

Our algorithm uses graph coloring to divide symbols from A' into three sets. Any symbol in A' 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 added to the $\diamond$-set. The graph we build for each path will contain a vertex for each symbol of the alphabet, $a'_i \in A' \cup \{\diamond_1, \diamond_2, \ldots, \diamond_N\}$. The vertex structure we use for the assignment graphs is inherently different from the node structure we use in the rule tree. Let us look at the basic vertex structure.

- Value
  - The value of a vertex is a symbol, $a'_i \in A' \cup \{\diamond_1, \diamond_2, \ldots, \diamond_N\}$.

  - Each vertex has a unique value.

- Color
  - This corresponds to 0, 1, $\diamond$.
  - By default a vertex is colored white.
  - White corresponds to 0.
  - Black corresponds to 1.
  - Grey corresponds to $\diamond$.
- Discovered Flag
  - This is a flag used to identify whether a vertex has been discovered.
  - By default, all vertices' discovered flag is set to false.

5.2 Graph Construction

We convert rules into edges between the vertices of this assignment graph. Recall that there are two classes of rules, $(a \uparrow b)$ and $(a \not\uparrow b)$. Rules of the first class result in a *positive edge*, marked with the value $+$, between the vertex with value a and the vertex with value b. The two vertices will either be the same color or at least one of them will be gray. Rules of the form $(a \not\uparrow b)$ result in a *negative edge*, marked with the value $-$, between the vertex with value a and the vertex with value b. These two vertices cannot be the same color under any circumstance. Further, we conclude that neither vertex can be colored gray. A $\diamond$ can appear in an incompatibility rule, but due to the fact that $\diamond$ must be compatible with any other symbol of the alphabet, then this symbol does not function as a hole, and therefore the vertex cannot be colored gray. Since the algorithm involves traversing our assignment graph, let us mention that the edges of this graph are undirected

such that an edge between two vertices A and B allows us to move from A to B or from B to A as we see fit.

All valid rules in our AND-set must be satisfied to reach a solution regardless of which path through the rule tree we traverse. Therefore, these rules will be applied to every possible assignment graph we generate. For this reason, we begin by populating our graph with the positive edges that correspond to the AND-set. Each rule is applied, and for each rule, we will have a new positive edge between two vertices. The graph we build up to this point is composed entirely of positive edges, and we call this the base graph because these edges appear in all graphs we generate. If we apply some incompatibility rules that produce a contradiction, for instance, we will abandon that particular graph and begin anew with this same base graph.

Each path in our rule tree dictates a particular configuration of negative edges of a specific graph. When we encounter a rule such as $(a \not\curlyvee b)$ then we add a negative edge to connect the vertex with value a and the vertex with value b. Upon reaching a terminal node, we know that we have a fully constructed graph, and thus we can begin coloring the vertices.

Example 15 *Using our example* $u' = ab\diamond_1\diamond_2 cd$ *and the tree we generated in the previous chapter, we see that* $A' = \{a, b, c, d\} \cup \{\diamond_1, \diamond_2\}$. *Furthermore, the AND-set,* $\{(a \uparrow \diamond_1), (b \uparrow \diamond_2), (\diamond_1 \uparrow c), (\diamond_2 \uparrow d)\}$. *Using this information we can build the base graph shown in Figure 5.1.*

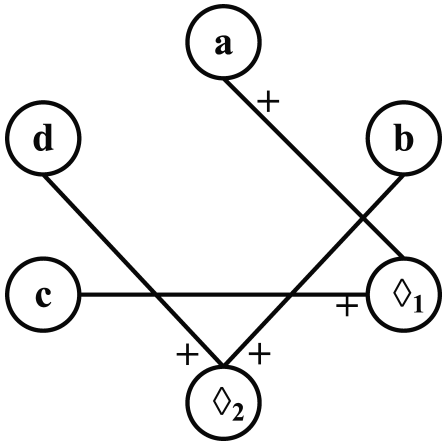

Figure 5.1: Base Graph of $ab\diamond_1\diamond_2cd$

Now, if we follow the path in our tree which generates the rules $(a \not\gamma d), (a \not\gamma c), (a \not\gamma \diamond_2), (a \not\gamma b)$*, then we get the graph shown in Figure 5.2 which we may now attempt to* 3*-color.*

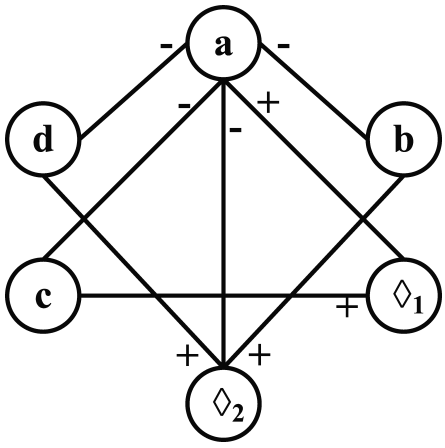

Figure 5.2: A Specific Uncolored Assignment Graph of $ab\diamond_1\diamond_2cd$

5.3 Graph Coloring

Let us begin by introducing our vertex coloring algorithm. If some 3-coloring exists, then our graph represents a solution. Any vertex of our graph can be black or white, just as any position in our solution, v can be encoded as a 0 or 1. We reserve the color gray for specific vertices whose value falls in the set $\{\diamond_1, \diamond_2, \ldots, \diamond_N\}$, the set of holes of u', but we also place an additional constraint that a gray vertex can only have positive edges incident to it. This constraint on coloring a vertex gray allows us to guarantee that $H(v) \subset H(u)$, if possible. No individual graph can promise a solution which fits our criteria, and in fact, it may be possible that no solution exists for the input. Some partial words fit into a class which we will describe later for which there is no solution v over the alphabet $\{0, 1\}$ where $\mathcal{P}(v) = \mathcal{P}(u)$, $\mathcal{P}'(v) = \mathcal{P}'(u)$, and $H(v) \subset H(u)$.

To begin our coloring of the graph, we must first choose an entry vertex into the graph. We want to avoid choosing a vertex with a $\diamond$ value. We also have a design criterion that our output should start with the symbol 0 whenever possible. Therefore, when choosing the entry vertex, we determine the symbol of the first element in $D(u')$, and choose the vertex with that value as our entry point. When we start coloring, this initial vertex is colored white by default, and due to the restriction on vertex coloring, it will remain white throughout the entire coloring and traversal process. However, if we are dealing with a word for which $D(u') = \emptyset$, then we typically just choose to enter the graph at the vertex with a value of $\diamond_1$, which is the first position in u'. An additional complexity to mention is that despite our best attempts, we may still end up with a final result that does not begin with 0, but this situation can easily be remedied simply by replacing all 0's with 1 and all 1's with 0.

Example 16 *If* $u' = \diamond_1\diamond_2 a \diamond_3 babab$*, then we choose the vertex with value* a *as our entry point into the graph because the third position is the first element in* $D(u')$*. We do this as a matter of convention because it is our goal to have the first non-hole position in* v *to be a* 0*. Thus, we color our start vertex white, and we flag it to denote that it has been discovered, and its color cannot be changed since it was not a* $\diamond$ *in* u'*. However, if the vertex with value* $\diamond_1$ *is colored black after our graph is* 3*-colored, then obviously our* v *would not begin with* 0*. Since we prefer our* v *to begin with* 0*, we would simply change all* 0*'s to* 1 *and all* 1*'s to* 0 *in our solution.*

If our graph contains only one vertex, we are finished. However, we describe how our algorithm explores and colors the the vertices of a non-trivial graph, containing more than one vertex. We explore each edge radiating from a vertex in depth first fashion, starting from our entry vertex. We must introduce the idea of an expected color. The expected color is calculated as shown in the table of Figure 5.3. 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 gray. Otherwise, if we cross a negative edge, we would expect the new vertex to be black. By default, if we start at a gray vertex, we choose our expected color to be white. Later, if we discover that white was a bad choice, we can re-explore this edge and use black as our expected color.

Color	Positive Edge (+)	Negative Edge (−)
White	White/Gray	Black
Black	Black/Gray	White
Gray	White/Black	Impossible

Figure 5.3: Expected Color Matrix

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 with our current vertex, which is currently the entry vertex. As we cross the edge to visit the neighbor, we calculate the expected color. One of three situations will occur.

1. The edge leads to an undiscovered vertex.
2. The edge leads to a previously discovered vertex and ...
    - the vertex is colored as expected.
    - the vertex is incorrectly colored.

Case 1 (Undiscovered Vertex) *Flag the vertex as discovered. Color the vertex the expected color. If an unexplored edge exists, explore it, otherwise return to the previous vertex which led to this vertex's discovery, and continue exploring the edges of that vertex.*

Case 2 (Discovered Vertex, Proper Color) *Everything is going as planned since the coloring has not yielded a contradiction. Return to the vertex which led to this vertex. Let us note that the edge between these vertices is not the edge which led to the discovery of this vertex. Then continue explore the remaining edges of the previous vertex if any remain unexplored.*

Case 3 (Discovered Vertex, Color Conflict) *This case is very tricky because it represents a contradiction. Once a vertex has been visited, its color is fixed and should not be changed. So clearly if we arrive at a previously discovered vertex that is not colored as expected, we have a problem. If we can return to a $\diamond$ vertex with*

no negative edges incident to it, then we might be able to color it gray and try a different expected color as we retraverse 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 $\diamond$ *vertex exists, we know immediately that this assignment graph can not be* 3*-colored, so we must abandon it and explore the next path in our rule tree.*

How did we arrive at this contradictory coloring? Assuming there is some $\diamond$ *vertex with no negative edges incident to it, we explored an edge, calculated some expected color, and began exploring and coloring some subgraph of our assignment graph until we reached a contradiction. Therefore, to cope with the contradiction, we must unexplore this subgraph and undo all the changes we have made after crossing the edge incident to the* $\diamond$ *vertex. We undiscover the vertices and color them white, retracing our steps until we once again arrive at the* $\diamond$ *vertex. Now we color the vertex gray, keep in mind that this vertex might already be gray, but this is not a problem. Then we re-explore the problematic edge, but this time we choose a different expected color. So, if we had first tried white, we now try black. If we first tried black, then now we will try white. This time we traverse the subgraph exactly as we did before. However, if another contradiction occurs along the subgraph beginning with this problematic edge which requires us to backtrack to this* $\diamond$ *vertex, then we have exhausted our coloring options, and again, we abort this assignment graph because it can not be* 3*-colored.*

At some point after fully exploring this subgraph, we will return to this $\diamond$ *vertex and explore the other edges incident to it. When exploring these edges, we might again find that our expected color causes a contradiction in the subgraph we are exploring. We only abort the assignment graph if an edge incident to a* $\diamond$ *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 explored 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 correlate to 1 in v, and gray vertices correlate to $\diamond$ in v. 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, if such a solution exists. We may still need to invert our 0's and 1's to ensure that the first non-hole position in v is a 0, but this is an elementary step we take for aesthetic purposes only.

The graph coloring algorithm we use is shown next. The traverse function is the main entry point in our code.

```
Traverse(Vertex Root)
    If Visit(Root) Is Successful And
       The Traversal Was Not Aborted Then
          SOLUTION HAS BEEN DISCOVERED
    End If
End Function
```

```
Explore(Vertex Current, Vertex Neighbor, Color ExpectedColor)
   Returns Boolean
   // First Attempt To Visit The Neighbor
   If Visit(Neighbor, ExpectedColor) Is Unsuccessful Then
      If Current ∈ H(u) And All Incident Edges Are Positive Then
         Color Current Vertex Gray
         // We Tried One Color, Let Us Try The Other
         ExpectedColor ← Not(ExpectedColor)

         textsl// Final Attempt To Visit The Neighbor
         If Visit(Neighbor, ExpectedColor) Is Unsuccessful Then
            // We Have Exhausted Our Coloring Options
            // This Graph Cannot Be 3-Colored
            ABORT
         End If
      Else
         // Current Is Not A Hole So We Must Undo Our Changes
         // And Return To A Vertex Corresponding To A Hole
         Rollback All Changes Which Were Made To The Graph
            After Visiting The Neighbors of Current
         Return False
      End If
   End If
End Function
```

```
Not(Color VertexColor) Returns Color
   If VertexColor Is White Then
      Return Black
   Else
      Return White
   End If
End Function
```

```
Visit(Vertex Current, Color ExpectedColor) Returns Boolean
   // CASE ONE: Current Was Already Visited
   If Current Was Already Visited Then
      If ColorOf(Current) Is ExpectedColor Then
         Return True
      Else
         ExpectedColor ← ColorOf(Current)
         Return False
      End If
   End If

   // CASE TWO: Current Has Not Yet Been Visited
   Mark Current As Visited
   Color Current The Expected Color

   For Each Edge Incident With Current

      If Incident Edge Is Positive Then
         ExpectedColor ← ColorOf(Current)
      Else
         ExpectedColor ← Not(ColorOf(Current))
      End If

      If Explore(Current, Neighbor, ExpectedColor) Is
         Unsuccessful Then
            Return False
      End If

   Next Edge

   // All Neighbors Have Been Visited Without A Conflict
   Return True
End Function
```

Now, let us look at some examples of an assignment graph and its coloring. For these examples, our normalized word is $u' = ab\diamond_1\diamond_2 cd$. Let us assume we have generated the rule tree, and we are traversing a particular path. Here, we have chosen the vertex with value a as our entry point into the assignment graph. Let us look at the vertex coloring process ...

Step 1 *We begin by discovering vertex a. By default, we color it white.*

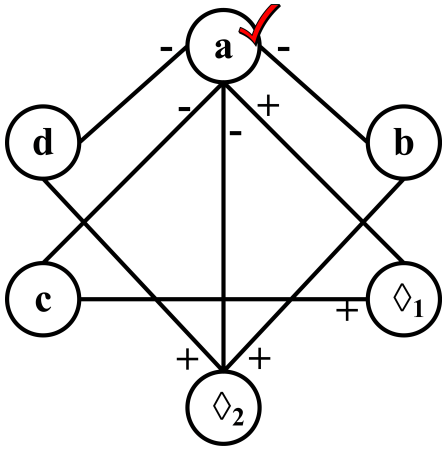

Figure 5.4: Step One

Step 2 *We choose to follow the positive edge from vertex a to vertex* $\diamond_1$*. Since vertex a is white and we are following a positive edge, we determine that our expected color for vertex* $\diamond_1$ *is white. Thus, we discover vertex* $\diamond_1$ *and color it white.*

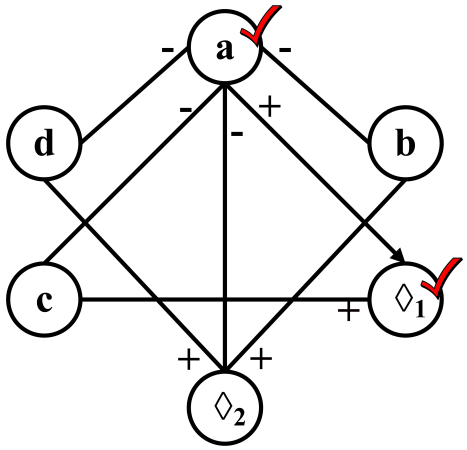

Figure 5.5: Step Two

Step 3 *Here, vertex* $\diamond_1$ *is white, and we follow the positive edge to vertex c. Our expected color will be white, so we discover vertex c and color it white. The only remaining edge from vertex c is negative and it leads us to vertex a, which has already been discovered. We have a problem, though, since vertex c is white and we are crossing a negative edge, we would expect vertex a to be black. Since it is not black, we must undiscover vertex c and return to vertex* $\diamond_1$*. In addition, we recalculate expected value such that we expect vertex a to be white. Crossing the negative edge from vertex a to vertex c leaves us with an expected color of black. Then we cross the positive edge from vertex c to vertex* $\diamond_1$*, leaving us with an expected color of black.*

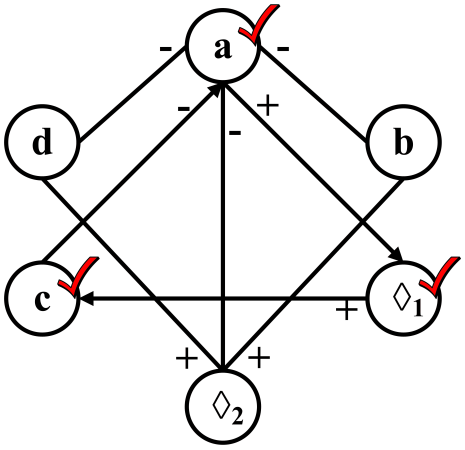

Figure 5.6: Step Three

Step 4 *We have returned to vertex* $\diamond_1$. *Since this vertex corresponds to a hole, we color it gray, then we try to retraverse the graph. This time, our expected value is black.*

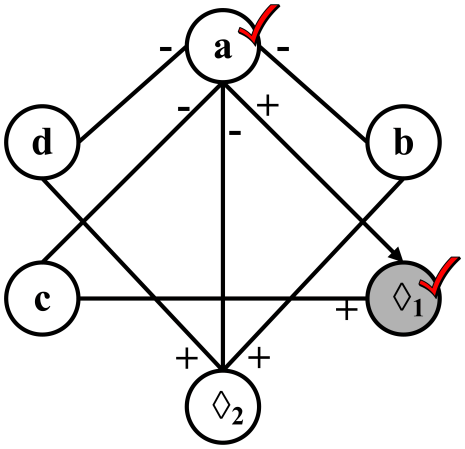

Figure 5.7: Step Four

Step 5 *We are back to vertex c, this time with an expected color of black. We rediscover vertex c, and color it black. From here, we return to vertex a by traversing the negative edge between vertex c and vertex a. We calculate the expected value to be white, and since vertex a was already discovered and already colored white, we have finished traversing the edges of vertex c. We return to vertex* $\diamond_1$*, and again, there are no more edges to traverse, so we return to vertex a.*

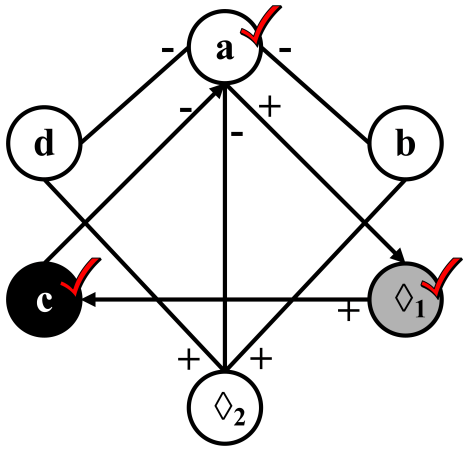

Figure 5.8: Step Five

Step 6 *The next edge we follow from vertex a is a negative edge which leads us to vertex d. Our expected color is black because vertex a is white, and we are traversing a negative edge. We discover vertex d and color it black.*

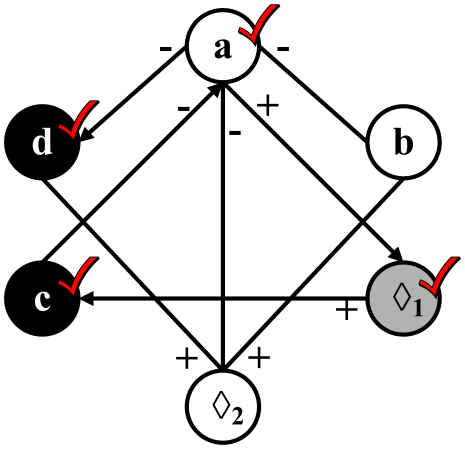

Figure 5.9: Step Six

Step 7 *The only remaining edge of vertex d is a positive edge leading us to vertex* $\diamond_2$. *We calculate the expected value to be black, thus we discover vertex* $\diamond_2$ *and color it black.*

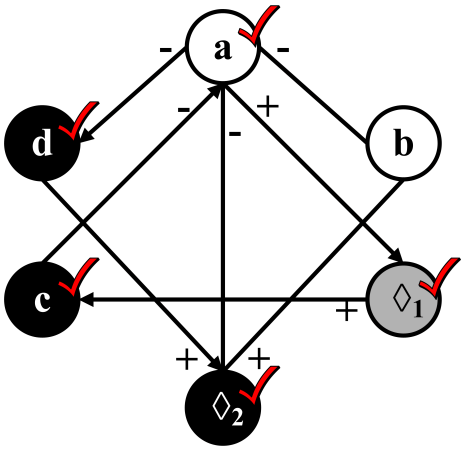

Figure 5.10: Step Seven

Step 8 *There is one positive edge leading from vertex* $\diamond_2$ *to vertex b. We discover vertex b and color it black.*

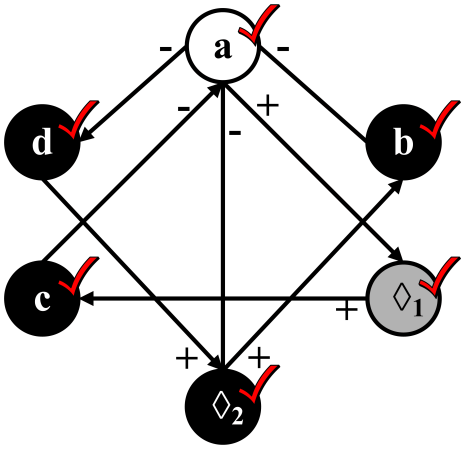

Figure 5.11: Step Eight

Step 9 *There is one negative edge from vertex b to vertex a. Since vertex b is black, we calculate the expected color to be white. Notice that vertex a is already discovered, so all that we need to do is check that vertex a matches the expected color.*

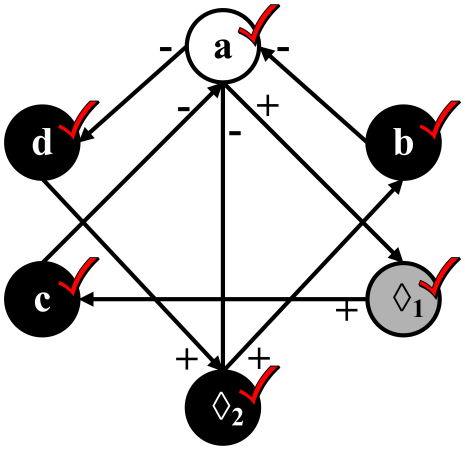

Figure 5.12: Step Nine

Step 10 *Now we return to vertex $\diamond_2$ to explore the final edge in the graph. This edge is a negative edge leading to vertex a. The expected color is white since vertex $\diamond_2$ is black and we are crossing a negative edge. Since vertex a is the proper color and we have traversed all edges in the graph, we have found a valid* 3*-coloring for this graph and our traversal was successful.*

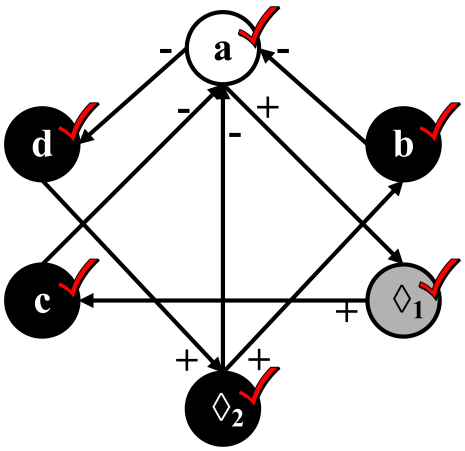

Figure 5.13: Step Ten

Now we clearly see from Figure 5.13 that a is white, b is black, c is black, d is black, $\diamond_2$ is black, and $\diamond_1$ is gray. Instead of colors, we translate white, black, and gray to $0, 1$, and $\diamond$, yielding a solution, $v = 01\diamond111$.

CHAPTER VI
THE SPECIAL CLASS OF PARTIAL WORDS

Originally the focus of this research was on partial words with at most two holes. However, the algorithm described in this thesis proved to be adaptable to partial words with any number of holes, and rather than enforce an artificial limit of two holes, we chose to explore the possibility that it would work for partial words with any number of holes. However, it eventually became apparent that there exists a class of partial words for which no solution exists satisfying our conditions.

Try not to assume that this means the algorithm we have described is fundamentally flawed, to do so would be a mistake. Indeed, if such conversion v over $\{0, 1\}$ exists, our algorithm will find it. However, the class of special partial words presents a challenge because such a v which meets our requirements does not exist. In this class, we can satisfy the constraints that the set of periods of v is equal to the set of periods of u and the set of weak periods of v is equal to the set of weak periods of u, $\mathcal{P}(u) = \mathcal{P}(v)$ and $\mathcal{P}'(u) = \mathcal{P}'(v)$. A problem arises when we attempt to satisfy the constraint that $H(v) \subset H(u)$. It is surprisingly easy to understand this concept.

Example 17 *If we start with a word $u = abcdab$, we would expect our algorithm to output $v = 010001$. Here, $\mathcal{P}(u) = \mathcal{P}(v) = \{4, 6\}$ and $\mathcal{P}'(u) = \mathcal{P}'(v) = \{4, 6\}$. A binary partial word such as $v' = 010\diamond01$ possesses the same periods and weak periods of u, but notice that the set of holes of v' is non-empty. The periods and weak periods of our v' are $\mathcal{P}(v') = \{4, 6\}$ and $\mathcal{P}'(v') = \{4, 6\}$, so we have satisfied*

the requirement that $\mathcal{P}(u) = \mathcal{P}(v')$ *and* $\mathcal{P}'(u) = \mathcal{P}'(v')$. *However* $H(u) = \emptyset$, *yet* $H(v') = \{3\}$, *so clearly* $H(v') \not\subset H(u)$. *Thus, we should conclude that* v' *is not an acceptable result because we already know that some* v, *namely* 010001, *satisfies all of our constraints.*

In the special class of partial words, we must consider the possibility that one or more positions in the domain of u will have to become a hole in our final solution, a situation which we have carefully avoided up to this point. The algorithm as we have described it takes several precautions to ensure that $H(v) \subset H(u)$, such as assigning an element of $\{0, 1\}$ to the each symbol of the alphabet A'. So for instance, there is no possibility of assigning one position with 1 and another position with 0 if both positions are a's in u'. In the contrasting approach, we could have assigned binary values on a position-by-position basis, such that the assignments are made independently of the alphabet, A'. Another precaution we take occurs during the graph coloring phase of the algorithm. During this phase, we do not allow non-hole symbols to be colored gray and thus these symbols cannot become a $\diamond$ in any solution we discover.

Again, we will continue to maintain that the set of periods and weak periods of all partial words is independent of the alphabet size, but in this special class, we are unable to maintain that $H(v) \subset H(u)$. Let us look more closely at this concept.

Example 18 *First, look at the non-special partial word* $u = ab\diamond\diamond cd$ *for which we can find a suitable* v. *This* u *can be normalized such that* $u' = ab\diamond_1\diamond_2 cd$. *For this* u, *we have previously shown that there is a* $v = 01\diamond 111$. *This is a great example because you will notice that* $\diamond_1$ *remains a* $\diamond$ *in* v, *however,* $\diamond_2$ *becomes a* 1, *such that* $H(v) \subset H(u)$. *This is perfectly acceptable because the set of holes in* v *is still a subset of the holes in* u.

Now let us examine the special partial word $u = a\diamond\diamond\diamond\diamond\diamond d\diamond bad$. *We normalize this partial word as* $u' = a\diamond_1\diamond_2\diamond_3\diamond_4\diamond_5 b\diamond_6 cab$. *The alphabet of this* u' *is* $A' = \{a, b, c\} \cup \{\diamond_1, \diamond_2, \diamond_3, \diamond_4, \diamond_5, \diamond_6\}$. *However, after exhausting all branches of the rule tree that this special partial word generates, we would not discover a suitable solution* v. *Notice that some binary partial word* v' *exists that will satisfy the periodicity of this* u, *consider* $v = 0111\diamond\diamond 1\diamond 1\diamond 1$.

A linear time technique has been created by F. Blanchet-Sadri, J. Gafni, and K. Wilson [9] to calculate a v for such special partial words. Let us briefly describe how this works. In our earlier chapters we introduced two derivative periodic sets, $\boldsymbol{\mathcal{P}}(u)$ and $\boldsymbol{\mathcal{P}}'(u)$. The set of strong periods of u, $\boldsymbol{\mathcal{P}}(u)$, is described as $\mathcal{P}(u) - |u|$. The set of strictly weak periods of u, $\boldsymbol{\mathcal{P}}'(u)$, is described as $\mathcal{P}'(u) - \mathcal{P}(u)$.

Example 19 *Recall from our previous example, Example 18, the special partial word* $u = a\diamond\diamond\diamond\diamond\diamond d\diamond bad$. *This periodicity of this word is,* $\mathcal{P}(u) = \{7, 9, 11\}$ *and* $\mathcal{P}'(u) = \{4, 5, 7, 9, 11\}$. *Thus we get* $\boldsymbol{\mathcal{P}}(u) = \{7, 9\}$ *and* $\boldsymbol{\mathcal{P}}'(u) = \{4, 5\}$.

Now we will build two sets of partial words, one for the strong periods of u, $\boldsymbol{\mathcal{P}}(u)$, and one for the strictly weak periods of u, $\boldsymbol{\mathcal{P}}'(u)$. The formula we use to generate these partial words is different for the two sets. Let us begin by describing how to build the partial words for $\boldsymbol{\mathcal{P}}(u)$. In this case, we want to generate a set of $|u|$-length full words for each period in $\boldsymbol{\mathcal{P}}(u)$.

$$\omega_p = \begin{cases} (01^{p-1})^k 01^{r-1} & if\ p > 0\ and\ r > 0 \\ (01^{p-1})^k & otherwise \end{cases}$$

The first condition applies when the length of our word is not modulo divisible by our period, such that there is a remainder. Here, we try to induce periodicity

by repeating a p-length word, with the singleton period p, k times after which we append 01^{r-1}, where r represents the remainder.

The second condition applies whenever our the length of our word is divisible by our period. Here we generate a $\boldsymbol{\mathcal{P}}_i$-length word, 01^{p-1}, and perform an i-concatenation such that the result is a u-length word.

Next we deal with the strictly weak periods of u. For each of these strictly weak periods, we generate a partial word as follows ...

$$\boldsymbol{\mathcal{P}}'_q = 01^{q-1}\diamond 1^{n-q-1}, \text{ where } n = kp + r \text{ and } 0 \leq r < p.$$

The resulting partial word has a weak period of $\boldsymbol{\mathcal{P}}'_i$ and exactly one $\diamond$.

Now we align each of the words we calculated for $\boldsymbol{\mathcal{P}}(u)$ and each of the partial words we calculated for $\boldsymbol{\mathcal{P}}'(u)$ and align them into $|u|$ columns. We perform the AND operation, $\wedge$, on each row. The $\wedge$ operation is a binary operation as described in the following table ...

$\wedge$	0	1	$\diamond$
0	0	$\diamond$	$\diamond$
1	$\diamond$	1	$\diamond$
$\diamond$	$\diamond$	$\diamond$	$\diamond$

Figure 6.1: AND Operation

Visually, it is easier to inspect the columns and perform a short hand summary of the $\wedge$ operation, such that ...

Case 1 *If all the positions in a column are* 0*'s, then this position in our answer will also be an* 0*. Conversely, if all positions in this column are* 1*'s, then this position in our answer will also be* 1*.*

Case 2 *If any position in a column is a* $\diamond$*, or if the column is not uniformly* 0 *or uniformly* 1*, then this position in our answer will be a* $\diamond$*.*

The resulting answer will have the same periods and weak periods as our u, but the set of holes in the answer will not necessarily be a subset of the holes in u.

Example 20 *Let us look at* $u = a\diamond\diamond\diamond\diamond\diamond d\diamond bad$*. Here* $\boldsymbol{\mathcal{P}}(u) = \{7, 9\}$ *and* $\boldsymbol{\mathcal{P}}'(u) = \{4, 5\}$*. Using the formulas described above, we get* $\boldsymbol{\mathcal{P}}_7$ *generates* 01111110111*,* $\boldsymbol{\mathcal{P}}_9$ *generates* 011111111011*,* $\boldsymbol{\mathcal{P}}'_4$ *generates* 0111$\diamond$111111*, and* $\boldsymbol{\mathcal{P}}'_5$ *generates* 01111$\diamond$11111*. We align our computed partial words and perform the column by column calculation.*

	0	1	1	1	1	1	1	0	1	1	1
	0	1	1	1	1	1	1	1	1	0	1
	0	1	1	1	$\diamond$	1	1	1	1	1	1
	0	1	1	1	1	$\diamond$	1	1	1	1	1
Result:	0	1	1	1	$\diamond$	$\diamond$	1	$\diamond$	1	$\diamond$	1

Superficially, you might conclude that we have found a quick way to calculate v for all partial words, but this is not the case. This technique produces a result in linear time, however it makes no attempt to preserve $D(u)$, meaning it can and does assign $\diamond$'s to positions in v which were not $\diamond$'s in u. Unfortunately, we must accept this result for the special class of partial words, but otherwise, the result is only acceptable when no solution exists such that $H(v) \subset H(u)$.

Our algorithm is designed such that it only assigns binary values to symbols in $A' \cup \{\diamond_1, \diamond_2, .., \diamond_N\}$, our normalized alphabet, instead of assigning values to specific positions in u. Ultimately this ensures that we satisfy $H(v) \subset H(u)$, and if we can not satisfy that requirement, no result is given. If we were to modify our algorithm

slightly such that the rules we generate are based on positions in u, we are suddenly able to find a result even for this special class of partial words at the expense of losing our ability to guarantee $H(v) \subset H(u)$. Obviously due to the computational costs of our algorithm, this is not an ideal situation. Our current approach works as such ...

1. Normalize the input u, such that we have a resulting u' over the alphabet A'.
2. Generate our rule tree.
3. Explore paths of the rule tree.
   (a) If a graph is generated which can be 3-colored, reserving one color exclusively for the set $\{\diamond_1, \diamond_2, .., \diamond_N\}$, then use the coloring to encode u' into a v over $\{0, 1\} \cup \{\diamond\}$.
4. Output our v.

Notice that this algorithm acts as a recognizer for partial words u which contain an encoding v over $\{0, 1\} \cup \{\diamond\}$ such that it preserves the periods and weak periods of u and $H(v) \subset H(u)$. Therefore, we can conclude that this algorithm is also a recognizer for the special class of partial words. If we have exhausted all paths in our rule tree, and none of those paths has lead us to a solution, then we can assume that our partial word is special. Furthermore, if we have recognized such special u, we might simply add an additional subroutine to our algorithm that incorporates the techniques described in this chapter to output a solution.

Theoretically our algorithm as described in previous chapters is a recognizer for the class of partial words with a proper output v, but it is impractical to use this algorithm as a recognizer for this class of partial words. Unfortunately, a rule tree

can become quite extensive and it is possible that a tree might have many millions of branches to explore. Computationally, it is very expensive to explore a path, generate a graph, and then attempt to color the graph. If the graph cannot be colored appropriately, then we move on to the next path in the tree, and continue the process. The amount of time involved in seeking a solution dictates that we have some limits on the number of paths we can explore and the amount of time we have to perform computations on a partial word. When these limits are reached, we must stop the process, and thus it can be difficult to determine if we are dealing with a special partial word or whether we have simply discovered a partial word with a particularly low density of valid branches, that is branches of the rule tree which yield a good 3-colored graph and thus a valid result v. Therefore, until a faster linear time recognizer for the special partial words is developed, our algorithm makes a best faith attempt to find a solution. If we are able to fully explore a tree with no result without exceeding our time limit, then we can positively say we have discovered an special partial word. However, if we are unable to fully explore a tree for whatever reason, we do not attempt to display any output, because we do not know if our input is special or simply a computationally expensive partial word.

BIBLIOGRAPHY

[1] J. Berstel and L. Boasson, Partial words and a theorem of Fine and Wilf, *Theoretical Computer Science* **218** (1999) 135–141.

[2] J. Berstel and D. Perrin, *Theory of Codes* (Academic Press, Orlando, FL, 1985).

[3] F. Blanchet-Sadri, Periodicity on partial words, *Computers and Mathematics with Applications* **47** (2004) 71–82.

[4] F. Blanchet-Sadri, Codes, orderings, and partial words, *Theoretical Computer Science* **329** (2004) 177–202.

[5] F. Blanchet-Sadri, Primitive partial words, *Discrete Applied Mathematics* **48** (2005) 195–213.

[6] F. Blanchet-Sadri and Ajay Chriscoe, Local periods and binary partial words: an algorithm, *Theoretical Computer Science* **314** (2004) 189–216 (`http://www.uncg.edu/mat/AlgBin/`).

[7] F. Blanchet-Sadri, Kevin Corcoran and Jenell Nyberg, Fine and Wilf's periodicity result on partial words and consequences, preprint 2005 (`http://www.uncg.edu/mat/research/finewilf/`).

[8] F. Blanchet-Sadri and S. Duncan, Partial words and the critical factorization theorem, *Journal of Combinatorial Theory, Series A* **109** (2005) 221–245 (`http://www.uncg.edu/mat/cft/`).

[9] F. Blanchet-Sadri, J. Gafni, and K. Wilson, Correlations of partial words, *W. Thomas and P. Weil (Eds.): STACS 2007, LNCS 4393* (2007) 97–108.

[10] F. Blanchet-Sadri and Robert A. Hegstrom, Partial words and a theorem of Fine and Wilf revisited, *Theoretical Computer Science* **270** (2002) 401–419.

[11] F. Blanchet-Sadri and D.K. Luhmann, Conjugacy on partial words, *Theoretical Computer Science* **289** (2002) 297–312.

[12] F. Blanchet-Sadri and Nathan D. Wetzler, Partial words and the critical factorization theorem revisited, preprint 2005 (`http://www.uncg.edu/mat/research/cft2/`).

[13] R.S. Boyer and J.S. Moore, A fast string searching algorithm, *Communications of the ACM* **20** (1977) 762–772.

[14] G. Brassard and P. Bratley, *Algorithmics Theory and Practice* (Prentice Hall. Englewood Cliffs, New Jersey, 1988).

[15] D. Breslauer, T. Jiang and Z. Jiang, Rotations of periodic strings and short superstrings, *Journal of Algorithms* **24** (1997) 340–353.

[16] Y. Césari and M. Vincent, Une caractérisation des mots périodiques, *C.R. Acad. Sci. Paris* **268** (1978) 1175–1177.

[17] C. Choffrut and J. Karhumäki, Combinatorics of Words, in G. Rozenberg and A. Salomaa (Eds.), *Handbook of Formal Languages, Vol. 1, Ch. 6* (Springer-Verlag, Berlin, 1997) 329–438.

[18] M. Crochemore, F. Mignosi, A. Restivo and S. Salemi, Text compression using antidictionaries, *Lecture Notes in Computer Science* **1644** (Springer-Verlag, Berlin, 1999) 261–270.

[19] M. Crochemore and D. Perrin, Two-way string matching, *Journal of the ACM* **38** (1991) 651–675.

[20] M. Crochemore and W. Rytter, *Text Algorithms* (Oxford University Press, New York, NY, 1994).

[21] M. Crochemore and W. Rytter, Squares, cubes, and time-space efficient string searching, *Algorithmica* **13** (1995) 405–425.

[22] M. Crochemore and W. Rytter, *Jewels of Stringology* (World Scientific, NJ, 2003).

[23] J.P. Duval, Périodes et répétitions des mots du monoïde libre, *Theoretical Computer Science* **9** (1979) 17–26.

[24] J.P. Duval, Relationship between the period of a finite word and the length of its unbordered segments, *Discrete Mathematics* **40** (1982) 31–44.

[25] J.P. Duval, Périodes locales et propagation de périodes dans un mot, *Theoretical Computer Science* **204** (1998) 87–98.

[26] N.J. Fine and H.S. Wilf, Uniqueness theorems for periodic functions, *Proceedings of the American Mathematical Society* **16** (1965) 109–114.

[27] Z. Galil and J. Seiferas, Time-space optimal string matching, *Journal of Computer and System Sciences* **26** (1983) 280–294.

[28] L.J. Guibas and A.M. Odlyzko, Periods in strings, *Journal of Combinatorial Theory, Series A* **30** (1981) 19–42.

[29] L.J. Guibas and A. Odlyzko, String overlaps, pattern matching, and nontransitive games, *Journal of Combinatorial Theory, Series A* **30** (1981) 183–208.

[30] D. Gusfield, *Algorithms on Strings, Trees, and Sequences* (Cambridge University Press, Cambridge, 1997).

[31] V. Halava, T. Harju and L. Ilie, Periods and binary words, *Journal of Combinatorial Theory, Series A* **89** (2000) 298–303.

[32] D.E. Knuth, J.H. Morris and V.R. Pratt, Fast pattern matching in strings, *SIAM Journal of Computing* **6** (1977) 323–350.

[33] R. Kolpakov and G. Kucherov, Finding approximate repetitions under Hamming distance, Lecture Notes in Computer Science, Vol. 2161 (Springer-Verlag, Berlin, 2001) 170–181.

[34] R. Kolpakov and G. Kucherov, Finding approximate repetitions under Hamming distance, *Theoretical Computer Science* **33** (2003) 135–156.

[35] G. Landau and J. Schmidt, An algorithm for approximate tandem repeats, Lecture Notes in Computer Science, Vol. 684 (Springer-Verlag, Berlin, 1993) 120–133.

[36] G.M. Landau, J.P. Schmidt and D. Sokol, An algorithm for approximate tandem repeats, *Journal of Computational Biology* **8** (2001) 1–18.

[37] A. Lempel and J. Ziv, Compression of individual sequences via variable-rate coding, *IEEE Transactions on Information Theory* **24** (1978) 530–536.

[38] M. Lothaire, *Combinatorics on Words* (Addison-Wesley, Reading, MA, 1983; Cambridge University Press, Cambridge, 1997).

[39] M. Lothaire, *Algebraic Combinatorics on Words* (Cambridge University Press, Cambridge, 2002).

[40] M. Lothaire, *Applied Combinatorics on Words* (Cambridge University Press, Cambridge, 2005).

[41] D. Margaritis and S. Skiena, Reconstructing strings from substrings in rounds, in *Proceedings of the 36th Annual Symposium on Foundations of Computer Science* (1995) 613–620.

[42] S. Maurer and A. Ralston, *Discrete Algorithmic Mathematics, Third Edition* (AK Peters, Ltd. Wellesley, Massachussets, 2004).

[43] F. Mignosi, A. Restivo and S. Salemi, A periodicity theorem on words and applications, *Lectures Notes in Computer Science* **969** (Springer-Verlag, Berlin, 1995) 337–348.

[44] P.A. Pevzner, *Computational Molecular Biology An Algorithmic Approach* (MIT Press, Cambridge, MA, 2000).

[45] M. Régnier and W. Szpankowski, On the approximate pattern occurrences in a text, in *Compression and Complexity of Sequences* (1998) 253–264.

[46] E. Rivals and S. Rahmann, Combinatorics of periods in strings, *Journal of Combinatorial Theory, Series A* **104** (2003) 95–113.

[47] J.P. Schmidt, All highest scoring paths in weighted grid graphs and their application to finding all approximate repeats in strings, *SIAM Journal of Computing* **27** (1998) 972–992.

[48] J. Setubal and J. Meidanis, *Introduction to Computational Molecular Biology* (PWS Publishing Company, Boston, MA, 1997).

[49] A.M. Shur and Y.V. Gamzova, Partial words and the periods' interaction property, *Izvestiya RAN* **68** (2004) 199–222.

[50] A.M. Shur and Y.V. Konovalova, On the periods of partial words, *Lecture Notes in Computer Science* **2136** (2001) 657–665.

[51] H.J. Shyr, *Free Monoids and Languages* (Hon Min Book Company, Taichung, Taiwan, 1991).

[52] M. Sipser, *Introduction to the Theory of Computation* (PWG Publishing Company, Boston, Massachussets, 2005).

[53] J.A. Storer, *Data Compression: Methods and Theory* (Computer Science Press, Rockville, MD, 1988).

[54] H. Wilf, *Algorithmics and Complexity, Second Edition* (AK Peters, Ltd. Natick, Massachussets, 2002).

[55] E. Yang and J. Kieffer, On the performance of data compression algorithms based upon string matching, *IEEE Transactions on Information Theory* **44** (1998) 47–65.

[56] J. Ziv and A. Lempel, A universal algorithm for sequential data compression, *IEEE Transactions on Information Theory* **23** (1977) 337–343.