Encoding methods for DNA languages defined via the subword closure operation

By

Bo Cui

A thesis submitted to

Saint Mary's University, Halifax, Nova Scotia

in Partial Fulfillment of the Requirements for

the Degree of Master of Applied Science (in Computer Science)

May 2007, Halifax, Nova Scotia

Your file Votre référence
ISBN: 978-0-494-35761-3
Our file Notre référence
ISBN: 978-0-494-35761-3

Certification

Encoding methods for DNA languages defined via the subword closure operation

by

Bo Cui

A thesis submitted to Saint Mary's University, Halifax, Nova Scotia,
in partial fulfillment of the requirements for the
degree of Master of Science in Applied Science

August 24, 2007

Examining Committee:

Approved: Dr. Patricia Evans, External Examiner
Department of Computer Science, University of New Brunswick

Approved: Dr. Stavros Konstantinidis, Senior Supervisor
Department of Mathematics and Computing Science

Approved: Dr. Norma Linney, Supervisory Committee
Department of Mathematics and Computing Science

Approved: Dr. Genlou Sun, Supervisory Committee
Department of Biology

Approved: Dr. Sageev Oore, Program Co-ordinator Representative

Approved: Dr. Kevin Vessey, Dean of Graduate Studies

Abstract

In DNA computing, information is encoded onto DNA sequences. The DNA codes in the form of single-stranded DNA sequences are not stable. This is because when two single-stranded DNA sequences, used to carry data, have complement parts on them, they naturally tend to stick to each other. This is due to the Watson-Crick complementarity property and causes the problem of undesirable bonds. Some properties and constraints have been proposed to prevent the problem, but most of them are local constraints which concentrate on a segment of a DNA word of a certain length. Therefore, if we concatenate some DNA words satisfying some local constraints, the resulting words might violate the same constraints. This makes encoding methods for DNA languages difficult to design. To solve this problem, we investigate some properties of the subword closure operation that is used for constructing DNA languages and propose practical encoding methods for such languages. We also implement our methods using advanced C++ tools for finite automata as well as design a web interface that allows users to obtain a DNA language in response to given values for certain parameters.

ACKNOWLEDGMENT

Sincere gratitude is given to all the people who made this thesis possible!

It is my pleasure to work with Dr. Stavros Konstantinidis. He is a knowledgeable and patient supervisor. He helped me to develop my understanding on DNA computing and taught me how to do research.

Special thank is given to Dr. Patricia Evans (University of New Brunswick) for her valuable comments. Also, I would like to thank the members on my thesis committee, Dr. Stravros Konstantinidis, Dr. Norma Linney, and Dr. Genlou Sun for valuable feed-backs on revising my thesis.

I would like to thank the Department of Mathematics and Computing Science, the Faculty of Graduate Studies and Research, and the Faculty of Science for providing me the opportunity to study at Saint Mary's University and the financial support.

I appreciate the help from people in the Department of Mathematics and Computing Science. The courses offered by Dr. Pawan Lingras, Dr. Paul Muir, and Dr. Sageev Oore enriched my understanding in more aspects of computer science. Rose Daurie and Owen Smith helped me a lot on dealing with daily matters and technical problems.

Last but not least, I would like to thank my parents, to whom I dedicate this work, for their support and encouragement.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The structure of DNA

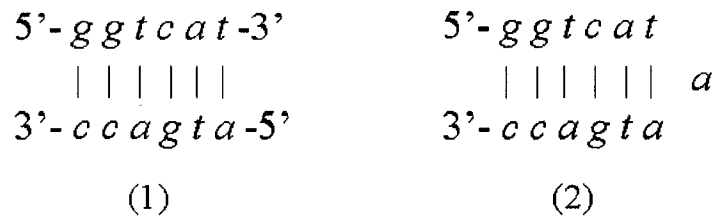

Figure 1.1: The structure of DNA

DNA (*deoxyribonucleic acid*) is naturally a double helix structure. It consists of four distinct nucleotides a [*adenine*], c [*cytosine*], g [*guanine*], and t [*thymine*]. In this thesis, we will simply use the letters a, c, g, and t to represent them. Many nucleotides line up to form a single-stranded *DNA sequence*. Due to the Watson-Crick complementarity property – a is the complement of t and c is the complement of g

– two single-stranded DNA sequences form a double helix by the hydrogen bonding (the vertical bars in Figure 1.1) between complementary nucleotides on each of the two strands.

Each single-stranded DNA sequence is oriented. Figure 1.1 depicts that each single-stranded DNA sequence starts at a *5'* end and ends at a *3'* end. So, when reading the single-stranded DNA sequence that is underneath, we need to read it from right to left, for example, the one in Figure 1.1(1) is read as *5' - atgacc - 3'*. If one single-stranded DNA sequence is of a sufficient length and contains two parts that are complementary, it can bend over to bind to itself. Also, in practice, it is not necessary for all the pairs on the two single-stranded DNA sequences to be complements. The following structure is possible:

```
5'- a g c t g c -3'
    | | | |   |
3'- t c g a t g -5'
```

Figure 1.2: A structure of DNA with a mismatch

1.2 The objective of this research

In DNA computing, we apply operations on physical DNA sequences to perform computations. The power of this kind of computation is that all the DNA operations work in parallel. However, there might be situations where some DNA operations will happen unexpectedly, for example, to form undesirable bonds between DNA

sequences. To avoid the undesirable bonds and improve the accuracy of the computational results, we have to follow some constraints. More details will be presented in the literature review chapter. Therefore, in order to apply DNA operations to solve computational problems, first of all, we have to design DNA sequences that satisfy important constraints and can be applied to encode data. In this thesis, we investigate methods for designing DNA sequences at the theoretical level. In other words, we investigate DNA languages that satisfy desirable constraints and can be used for encoding arbitrary data.

In this research, we investigate some general properties of the subword closure operation and cycles in automata. Therefore, even though this research is motivated from the DNA computing point of view, some of the methods are general for encoding data into languages that satisfy arbitrary local constraints and are defined via the subword closure operation.

An implementation using advanced C++ tools has been developed for producing desired DNA languages defined by the methods in this thesis.

1.3 The structure of the thesis

Explicit definitions and notations are given in Chapter 2. Some background information about theory of computation is also given in Chapter 2.

In Chapter 3, we briefly review the literature of DNA computing. We present some global constraints and local constraints that are usually considered in DNA language design. Also, we present some construction methods for DNA languages satisfying

different constraints.

In Chapter 4, we first present the general problem we want to solve, and then, propose a method, the B-block method, for solving the problem by using the subword closure operation. Moreover, we propose a method to calculate the density of $S^{\otimes}$, which is the language defined via the subword closure operation.

Because the method proposed in Chapter 4 is non-deterministic, in Chapter 5, we investigate the properties of $T^{\otimes}$, the automata that are used for producing languages satisfying desired local constraints, and propose a method for checking whether a $T^{\otimes}$ is able to produce a arbitrarily large set of code words. At last, an algorithm for the method is introduced.

In Chapter 6, we first propose some DNA language construction methods for some constraints that are related to DNA computing. And then, we focus on applying the methods in Chapter 4 to produce DNA words satisfying these constraints.

In Chapter 7, we describe the implementation of a web system that applies the methods in Chapter 4, 5, and 6. Users of the system are able to encode arbitrary data into DNA languages satisfying either some general constraints or the constraints related to DNA computing. In addition, a theorem in [28] is included in the web system to encode arbitrary data into DNA languages satisfying the DNA computing related constraints. Some tables are obtained in response to certain values of the parameters involved.

In Chapter 8, we summarize the important contents of this thesis and discuss directions for future research.

Chapter 2

Definitions, notations, and background information

2.1 Basic definitions and notations

An *alphabet* is a finite non-empty set of symbols. For instance, $A = \{a, b\}$ is an alphabet with two symbols, a and b, and $N = \{0, 1, 2\}$ is another alphabet with three symbols, 0, 1, and 2. A *word* over an alphabet is a finite sequence of symbols such that each symbol in the sequence is in the alphabet. For example, *abaab* is a word over A, but *abac* is not. The *empty word* is the word without any symbols. We denote the empty word by e. The length of a word w is denoted by $|w|$, i.e., if $w = a_1 a_2 \cdots a_n$ with each a_i being in the alphabet, then $|w| = n$. For example, $|aba| = 3$ and $|e| = 0$. A *language* over an alphabet is a set of words over the alphabet. The set of all the words over A is denoted by A^*, and the set of all the words over A of length k is

denoted by A^k.

The reversal of a word can be obtained by reading the word from right to left. We denote the reversal of a word w by w^R. For example, the reversal of *english* is *hsilgne*.

Hamming distance is a measure of similarity between two words of equal length. The Hamming distance between two words, $x = a_1a_2a_3\cdots a_l$ and $y = b_1b_2b_3\cdots b_l$, denoted by $H(x, y)$, is the number of symbols where $a_i \neq b_i$, for $1 \leq i \leq l$. For example, the Hamming distance between *bread* and *brood* is $H(bread, brood) = 2$, because they are different at the third and fourth positions.

The *Hamming ball of a word w with a Hamming distance d* is denoted by $H_d(w)$. Given an alphabet, the Hamming ball of w over the alphabet contains all the words such that the Hamming distance between any word in the Hamming ball and w is less than or equal to d. For example, over alphabet $A = \{a, b\}$, $H_1(abab) = \{abab, bbab, aaab, abbb, abaa\}$, because the Hamming distance between any word in this set and *abab* is less than or equal to 1. Similarly, the Hamming ball of a set A with a Hamming distance d contains all the elements in each of the Hamming balls of the elements in A with Hamming distance d, in symbols, $H_d(A) = \bigcup_{a\in A} H_d(a)$.

In mathematical terminology, a set is a collection of elements. The number of the elements can be finite or infinite. Therefore, a language is a set and a word is an element in a language. The cardinality of a set A, denoted by $|A|$, is the number of elements in A. if a word w is in a language L, this is denoted by $w \in L$. Operator $\cap$ takes the common elements of two sets. The result set is called the *intersection* of

two sets. For example, the intersection of set $A = \{a, b, c, d\}$ and set $B = \{c, d, f\}$ is $A \cap B = \{c, d\}$. If a set A is a subset of B, in symbols, $A \subseteq B$, each element in set A is also in set B. For example, $\{1, 2\} \subseteq \{1, 2, 4\}$. A pair, denoted by $\{x, y\}$, is a set with two elements. For any x, y, $\{x, y\} = \{y, x\}$. An ordered pair, denoted by (x, y), means that, if $x \neq y$, $(x, y) \neq (y, x)$. The Cartesian product of two sets A and B, denoted by $A \times B$, is the set of all ordered pairs (a, b) with $a \in A$ and $b \in B$. For example, $\{1, 2, 3\} \times \{a, b\} = \{(1, a), (1, b), (2, a), (2, b), (3, a), (3, b)\}$.

For a word w in the form of ps, the word p is a prefix of w; we denote the set of prefixes of w by Pref(w). Similarly, word s is a suffix of w; we denote the set of suffixes of w by Suff(w). If we write w in the form of pqs, q is a subword of w; Sub(w) denotes the set of subwords of w. In addition, we denote the set of prefixes of w of length k by $\text{Pref}_k(w)$, i.e., for the words over alphabet A, $\text{Pref}_k(w) = A^k \cap \text{Pref}(w)$. The same notation is used for $\text{Suff}_k(w)$ and $\text{Sub}_k(w)$

Definition 1 *Let S be a language containing only words of the same length k, for some positive integer k. The subword closure $S^{\otimes}$ ($\otimes$ is pronounced as o-times) is the set*

$$\{w \in \Sigma^* \mid |w| \geq k,\ \text{Sub}_k(w) \subseteq S\}.$$

By definition, any subword of length k of a word in $S^{\otimes}$ must be in the set S. For example, given a set $C = \{abc, bca, cab\}$, *abcab* is a word in $C^{\otimes}$, but *abcb* is not, because the subword *bcb* is not in C.

In [28], the authors define $S^{\oplus}$ ($\oplus$ is pronounced as o-plus) as $S^{\oplus} \stackrel{def}{=} S^{\otimes} \cap (\Sigma^k)^+$, which is a restricted version of $S^{\otimes}$. In addition to the definition of $S^{\otimes}$, $S^{\oplus}$ requires

that the length of any word in $S^{\oplus}$ must be a multiple of the word length in S, so $S^{\oplus} \subseteq S^{\otimes}$.

DNA consists of 4 distinct bases. We consider these 4 DNA bases as 4 symbols. Therefore, the DNA alphabet, denoted by Σ, has 4 symbols, a, c, g, and t. As physical DNA sequences only consist of these 4 bases, we consider them as DNA words. For example, the physical DNA sequence 5′-*agcagtt*-3′ is represented by the DNA word *agcagtt*, thereby, we can get the concept of DNA languages. They are depicted as sets of DNA words. Formally, for $\Sigma = \{a, c, g, t\}$, Σ^* denotes the set of all DNA words, Σ^+ denotes the set of all non-empty DNA words, and Σ^k denotes the set of all DNA words of length k.

Due to the Watson-Crick complementarity property where a is the complement of t and c is the complement of g, we denote the complement of a by $\tau(a)$, i.e., $\tau(a) = t$, $\tau(c) = g$, and vice versa. We can get the complement of a DNA word by switching each DNA symbol in the word to the complementary symbol and reversing the string. We denote the complement of a DNA word w by $\tau(w)$. For example, the complement of the word $w = agcgcta$ is $\tau(w) = tagcgct$. For a DNA word set S, $\tau(S)$ denotes the set in which each word is the reverse-complement of a word in S. The function τ is called the *DNA involution.*

2.2 Theory of automata

Theory of computation is a fundamental theory in computer science and is one of the oldest research areas in computer science. It can be applied to many computer

science research areas, especially, research areas about sequential language design and analysis such as natural language processing and DNA language design. In this thesis, because we want to address problems with the theory of automata, we need to present some background knowledge about this theory.

2.2.1 Finite automata

A deterministic finite automaton is a quintuple $M = (K, \Sigma, \delta, s, F)$, where K is a finite set of states, Σ is an alphabet, $s \in K$ is the initial state, $F \subseteq K$ is the set of final states, and δ is a set of rules. Each rule in δ is of the form $\delta(q_1, \sigma) \rightarrow q_2$, where q_1 and q_2 are two states in K, and σ is a symbol in Σ. When the current state of M is q_1, if it reads in symbol σ, it goes to state q_2. If an input word can lead an automaton to reach an accepting state, we say this automaton accepts the input word. For convenience, we depict an automaton with a state diagram. For example, Figure 2.1 depicts an automaton that accepts the language $(ab)^*$. (The initial state is preceded by the symbol $>$, and the final state(s) is indicated by double circles.) In symbols, the above automaton is $M = (K, \Sigma, \delta, s, F)$, where $K = \{q_0, q_1, q_2\}$, $\Sigma = \{a, b\}$, $s = q_0$, $F = \{q_0\}$, $\delta = \{\delta(q_0, a) \rightarrow q_1, \delta(q_0, b) \rightarrow q_2, \delta(q_1, a) \rightarrow q_2, \delta(q_1, b) \rightarrow q_0, \delta(q_2, a) \rightarrow q_2, \delta(q_2, b) \rightarrow q_2\}$. For example, if the input word is $abab$, the automaton goes to states in the order of q_1, q_0, q_1, q_0. As q_0 is the accepting state, the automaton accepts this input words. If the input word is aba, the automaton stops at state q_1, so, aba is not acceptable.

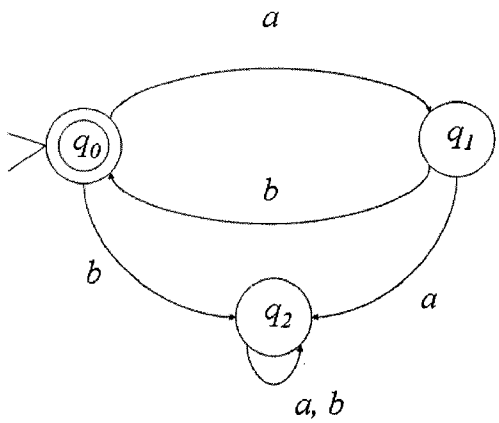

Figure 2.1: An automaton that accepts $(ab)^*$

2.2.2 Trie

A *trie*, also called *prefix tree*, is a finite automaton in the form of a tree structure. It is used for storing strings over an alphabet. The idea of this string storage is that all strings sharing a common stem or *prefix* hang off a common node. For example, in Figure 2.2, words *are* and *ark* share the same stem *ar*. The data structure for representing a trie will be depicted explicitly in Chapter 7.

2.2.3 Pushdown automata

A pushdown automaton is a sextuple $M = (K, \Sigma, \Gamma, \Delta, s, F)$, where K is a finite set of states, Σ is an alphabet (the input symbols), Γ is an alphabet (the stack symbols), $s \in K$ is the initial state, $F \subseteq K$ is the set of final states, and Δ, the transition function, is a finite subset of $(K \times (\Sigma \cup \{e\} \times \Gamma^*) \times (K \times \Gamma^*)$. In addition to a finite automaton, pushdown automata have a stack. Pushdown automata decide the next state not only based on the symbol just read but also the symbol at the top of the

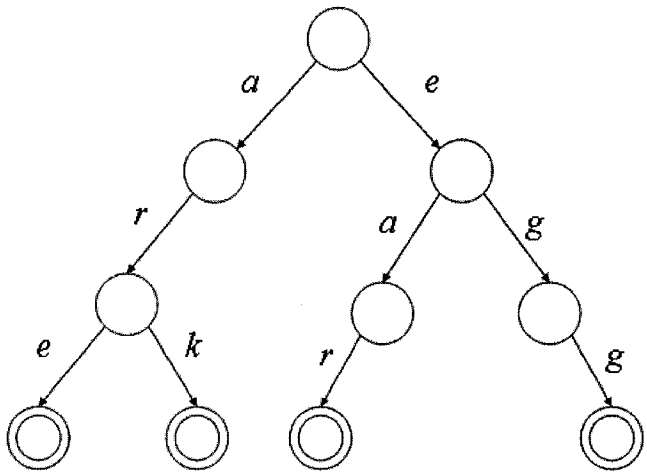

Figure 2.2: The trie recognizing words *are*, *ark*, *ear*, and *egg*

stack. They also operate the stack. The rules of pushdown automata are in the form of $((p, a, \beta), (q, \gamma))$. If the current state is p and the symbol at the top of the stack is β, a pushdown automaton may read a from the input word, replace β with γ, and enter state q. a, β, and γ could be the empty word e . The rule $((p, a, e), (q, \gamma))$ pushes γ (γ could be a word) to the top of the stack; the rule $((p, a, \beta), (q, e))$ pops a symbol β from the top of the stack. If $a = e$, a pushdown automaton does not read a symbol from the input word. A pushdown automaton accepts a word if and only if, after reading in all the symbols in the word, the pushdown automaton reaches a final state and the stack is empty.

Example 1

Let us design a pushdown automaton M_1 to accept language $L = a^n b^n$. For example, $aaabbb \in L$, but $aabbb \notin L$, and $abba \notin L$. $M_1 = (K, \Sigma, \Gamma, \Delta, s, F)$, where $K = \{s, f\}$, $\Sigma = \{a, b\}$, $\Gamma = \{a, b\}$, $F = \{f\}$, Δ contains the following rules:

1. $((s, a, e), (s, a))$

2. $((s, a, a), (s, aa))$

3. $((s, b, a), (f, e))$

4. $((f, b, a), (f, e))$

Usually, we use tables such as Table 2.1 to illustrate the procedure of the operations of pushdown automata. We illustrate the operations of M_1 on input string *aabb* in Table 2.1.

State	Unread Input	Stack	Rule used
s	$aabb$	e	-
s	abb	a	1
s	bb	aa	2
f	b	a	3
f	e	e	4

Table 2.1: The operations of M_1 on *aabb*

2.2.4 Intersection of two automata

The intersection of two sets is the set of common words in the two sets. The language accepted by an automaton is the set of all words accepted by this automaton. Therefore, the intersection of two languages accepted by two finite automata should be accepted by a finite automaton that accepts exactly the words accepted by both

of the finite automata. If $L = L_1 \cap L_2$, M_1 accepts L_1, and M_2 accepts L_2, then the *intersection automaton* $M_i = M_1 \cap M_2$ accepts L. For $M_1 = (K_1, \Sigma, \delta_1, s_1, F_1)$ and $M_2 = (K_2, \Sigma, \delta_2, s_2, F_2)$, M_i is defined as follows:

$$M_i = (K_1 \times K_2, \Sigma, \delta, (s_1, s_2), F_1 \times F_2)$$

where $\delta((p, q), a) = (\delta_1(p, a), \delta_2(q, a))$.

Each state of M_i is a pair in $K_1 \times K_2$. The initial state of M_i is the pair (s_1, s_2) such that s_1 is the initial state of M_1 and s_2 is the initial state of M_2. Since we want to accept a word if and only if it is accepted by both automata, the accepting states of M_i are pairs (p, q) such that p is an accepting state of M_1 and q is an accepting state of M_2. Each state (p, q) on an input symbol a goes to the state (x, y) such that $\delta_1(p, a) \rightarrow x$ and $\delta_2(q, a) \rightarrow y$.

Example 2

Figure 2.3 shows the automata, M_1 and M_2, such that, over $\Sigma = \{a, b\}$, M_1 accepts all the words with at least 1 a and M_2 accepts all the words with at least 1 b.

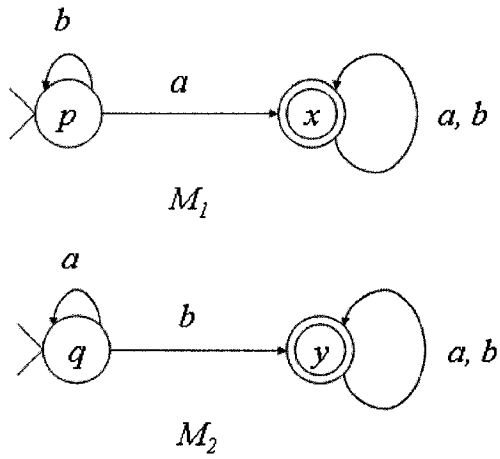

Figure 2.3: The automata M_1 and M_2

The intersection of M_1 and M_2, M_i, accepts all words with at least 1 a and at least 1 b. M_i is depicted in Figure 2.4.

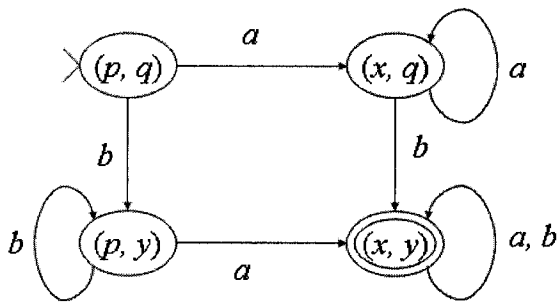

Figure 2.4: The intersection automaton of M_1 and M_2

Chapter 3

Literature review

3.1 Adleman's insight

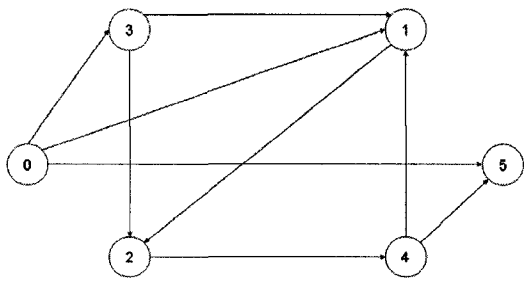

Figure 3.1: A directed graph. The Hamilton path: $0 \rightarrow 3 \rightarrow 1 \rightarrow 2 \rightarrow 4 \rightarrow 5$

In 1994, Adleman published his paper [1] on DNA computing which demonstrated the computational power of physical DNA sequences. He solved an instance of the directed Hamilton path problem [1, 2], which is an NP-complete problem. The goal of the directed Hamilton path problem is to find a path to go through a directed graph that starts and ends at specified nodes such that every node in the graph will

be visited once and only once.

As single stranded DNA sequences will automatically bind with the complementary single strands, Adleman used DNA sequences to represent the nodes and edges. The DNA sequences representing directed edges from node x to node y are designed to be able to connect the two nodes as xy. In the same way, the DNA sequences will bind to form longer DNA sequences to represent the paths through the graph. An example of nodes and edges is given as follows:

nodes	node sequences	edges	edge sequences
0	5′-*tactcatatggggttatacg*-3′	0→ 1	3′-*cccaatatgcgaggcggacc*-5′
1	5′-*ctccgcctgggcttagctta*-3′	1→ 2	3′-*cgaatcgaatctaggagaca*-5′
2	5′-*gatcctctgtttcctcagct*-3′		
3	5′-*ggctccacttactctcttgt*-3′		
4	5′-*tatgggctagcggtccggtt*-3′	4 → 5	3′-*gccaggccaacgggaacatc*-5′
5	5′-*gcccttgtagtctcgggtcc*-3′		
		0 → 3	3′-*cccaatatgcccgaggtgaa*-5′
		3 → 1	3′-*tgagagaacagaggcggacc*-5′
		3 → 2	3′-*tgagagaactctaggagact*-5′
		2 → 4	3′-*aaggagtcgaatacccgatc*-5′
		0 → 5	3′-*cccaatatgccgggaacatc*-5′
		4 → 1	3′-*gccaggccaagaggcggacc*-5′

Table 3.1: The DNA sequences representing the nodes and edges in graph 3.1.

Let us take the sequences representing node 0, node 1 and edge 0 → 1 for example:

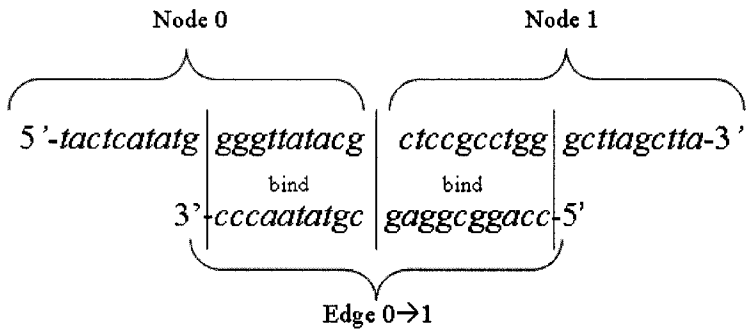

Figure 3.2: The DNA structure representing the path from node 0 to node 1.

from figure 3.2, we can see that the sequences representing node 0 and node 1 are concatenated by the sequence representing edge 0 → 1. The rest of the nodes and edges work the same way.

During the experiment, a lot of copies (approximately $3 * 10^3$) of each sequence representing nodes and edges were put together. The binding reaction among those sequences performed simultaneously and generated all possible paths in the graph. If there is a solution to the directed Hamilton path problem, the sequence representing the solution path will be generated during the reaction.

Because the growth rate of the number of strands to encode the data will be exponential when the number of the nodes that need to be processed increase, we have to fix an upper bound to the dimension of the input in order to evaluate the "feasibility" of DNA algorithms: in [5], the authors assume that 10^{21} is the upper bound to the number of DNA strands that an algorithm can treat. So, as suggested

in [16], in molecular computing the exponential barrier is the weight barrier, because it is the weight that imposes a limit to the volume of a test-tube. Thus, Adleman's experiment could theoretically be executed for a graph with up to 70 nodes, but, as shown in [16], if we have to execute the experiment for a graph with 200 nodes, we would need to manipulate DNA molecules for a weight heavier than that of the Earth.

3.2 Undesirable bonds

Adleman's experiment is based on the assumption that there is no mismatch between two DNA sequences or within one DNA sequence. It means that if the edge sequence from 1 to 2 binds with the nodes 0 and 2, the resulting sequence representing the Hamilton path will be incorrect; or if the the sequence representing node 2 bends over to form the second structure in Figure 1.1, there will be no available node 2 any more, so that the solution path cannot be generated. These mismatches are caused by undesirable bonds.

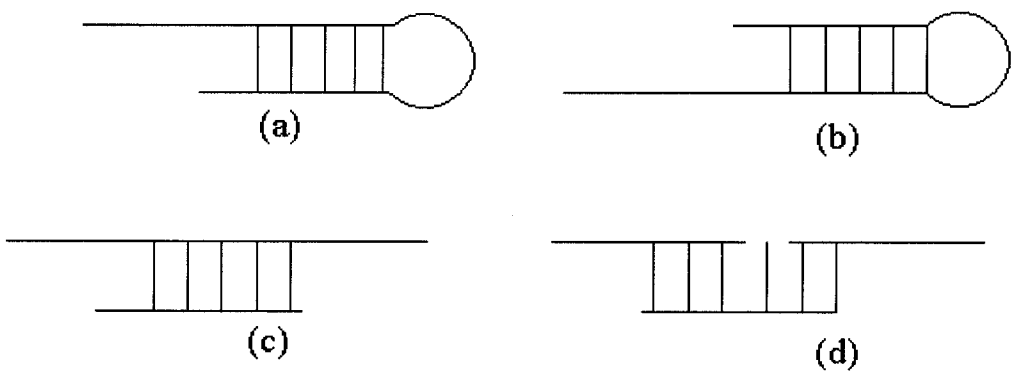

Figure 3.3: The structures of undesirable bonds

Several types of undesirable bonds may form within the DNA sequences from the initial data. These undesirable bonds can violate the accuracy of the operations. Figure 3.3 depicts several situations with undesirable bonds. In this case, the vertical bars represent the undesirable bonds, and the horizontal lines (including the circulars) represent the DNA sequences.

Undesirable bonds are a major problem in DNA computing. The goal of DNA language design is to prevent the occurrence of undesirable bonds in any molecular operations. We will review the research area of DNA language design in the next section.

3.3 DNA-based algorithm design and DNA-based computer design

3.3.1 DNA-based algorithm design

Adleman's experiment provides researchers with a new possible approach to design algorithms, parallel algorithms, even though the reliability of this approach is not sufficent at the current time. Because the DNA-based algorithms can perform tasks simultaneously, they have potential for much stronger computational efficiency. With this idea the conventional algorithms can be optimized, or DNA-based algorithms can be implemented to some problems that are hard for conventional algorithms to solve. For instance, the DNA-based computation approach to compute Dijkstra's algorithm has been studied in [20]. A DNA-based algorithm for solving an NP-

complete problem (the shortest common superstring problem) has been proposed in [14]. A DNA-based algorithm for solving image recognition problems for pattern matching has been proposed in [42].

Since Adleman initiated the research area of DNA computing [1], the contribution raises the hope to tackle NP-complete problems. However, at the same time, other researchers argue that NP-complete problems may not be the most suitable for DNA computing. A better subject for DNA computing could be large-scale evaluation of parallel computation models [37].

3.3.2 DNA-based computer design

Since DNA sequences have computational ability, they can carry out computations to replace the silicon circuits. This is the principle of DNA-based computers. The author of [8] presents a theoretical proof that DNA computing can simulate a universal Turing machine [44], which is the mathematical abstraction of a sequential computer.

The importance of a DNA-based computer is that it can carry out computations simultaneously. With this feature, artificial intelligence features can be realized by DNA-based computer [15], since an important requirement for realizing artificial intelligence is that the external information must be delivered to all computer devices simultaneously. The procedures for multiple inputs with DNA computing are proposed in [12]. Since a DNA-based computer can realize artificial intelligence, researchers proposed DNA-based computers that involve artificial intelligence. For example, a problem solving method with DNA-typed Semantic Net is proposed in [41]

to apply DNA computing to artificial intelligence. The authors of [32] present an optimal trajectory planning method of mobile robots using DNA computing. Moreover, this method is especially effective on a DNA-based computer.

DNA-based computers have not been realized, but some fundamental functions of a DNA-based computer have been studied and realized. At the hardware level, the authors of [40] find that it is possible to use easily fabricated nanocells as logic devices by setting the internal molecular switch states after the topological molecular assembly is complete. They simulate some logic devices including an inverter, a NAND gate, an XOR gate and a 1-bit adder; the simulation of Boolean circuits by finite splicing is presented in [11]; a microreactor with 20 nodes has been designed by the authors of [36], and they also improve the programmability of DNA-based computers. At the logic and arithmetic operation level, the procedure for logic operations and their time complexity are proposed in [13]. The algorithm for adding numbers with DNA is proposed in [46].

3.4 Significance of DNA language design

As we have presented, undesirable bonds can violate the accuracy of DNA computations and operations. In order to improve the reliability of DNA-based computers and DNA-based algorithms, the first thing we have to do is to design DNA languages of high quality. Those are sets of DNA words that are unlikely to form undesirable bonds with each other by hybridization. In order to prevent the problem of undesirable bonds, we have to investigate what kinds of properties the DNA languages must

have.

3.4.1 Code properties of DNA languages

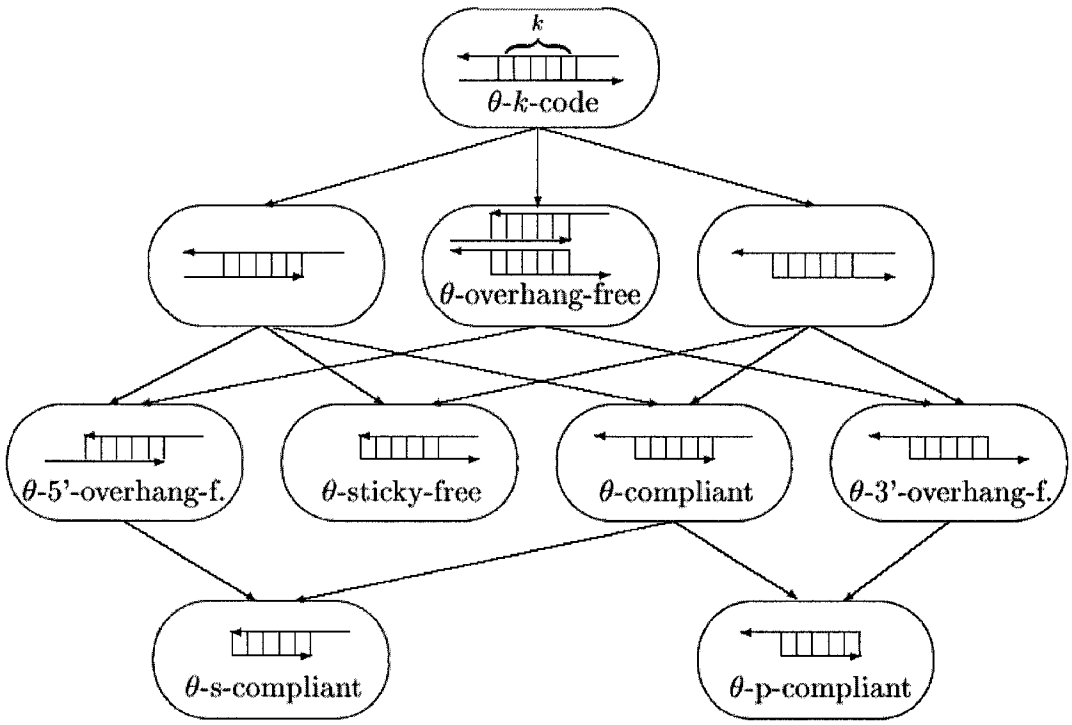

Figure 3.4: A structure of DNA with a mismatch

Figure 3.4 depicts the hierarchy of the structures of the undesirable bonds, which have been defined and analyzed in [29, 27, 19, 25]. The variable θ denotes a general involution (which could be the DNA involution). An involution θ satisfies $\theta(\theta(a)) = a$ for all letters a. More specifically, languages that are θ-k-codes, θ-compliant, θ-s-compliant and θ-p-compliant have been studied in [25]; θ-strictly-free ones have been studied in [19]; θ-3′-overhang-free, θ-5′-overhang-free and θ-overhang-free ones have been studied in [29, 27]. Each property can prevent the corresponding structure of

undesirable bonds in the same box. For example, a language L is a θ-k-code if, for any two subwords u and v of L of length k, that is $u, v \in \mathrm{Sub}_k(L)$, we have that $u \neq \theta(v)$. Based on these properties, some programs for generating DNA languages satisfying such properties have been developed in [21, 31]. The θ-s-compliant and θ-p-compliant languages are also called hairpin-free languages, whose elementary properties have been studied in [30] as well.

The structures of undesirable bonds in Figure 3.4 are related, so that the properties for preventing these structures are related. For example, if a language is θ-3′-overhang-free then it is also θ-p-compliant. The relationship among these structures also has been studied in [29, 27, 19, 25].

The above properties ensure that certain undesirable bonds can not occur when we construct the DNA language. We call them static properties. In addition, we also need some other properties such as dynamic properties. They ensure that, after a permitted bio-operation is applied to the DNA sequences, the resulting sequences also satisfy the desirable properties. The authors of [26] have studied both static and dynamic properties. The dynamic properties will be discussed in the next section.

3.4.2 Properties for generating infinite sets of words

Because none of the θ-compliance and θ-subword compliance are closed under concatenation [22], the authors of [22, 19] investigate the properties under which an infinite set of DNA words can be generated from a finite "good" set of DNA words with the same "good" properties by concatenating the DNA words in the finite sets. They

give some necessary and sufficient propositions by which the generated infinite word sets are $\theta(k, m_1, m_2)$-subword compliant, θ-compliant, and θ-free. The authors of [23] study more conditions under which more kinds of words can preserve their properties when the words are concatenated. The more general sets of words, θ-k-codes, are studied in [23].

There are two ways to generate infinite sets of words. The above method concatenates words to generate longer words. Another method that generates longer words is to use a splicing system. Splicing system is considered a computational model. Both the authors of [22, 19] consider the question that under certain kinds of properties, DNA words having good properties can keep them. That means that during or after computations, the DNA words still will not form undesirable bonds.

In [19], the authors propose an additional feature to the initial set of DNA words for preserving the good encoding properties during any computation. The authors give two propositions, which state that if the splicing base is strictly θ-free then all the words generated by splicing will not violate the property of θ-freedom.

In [22], the authors define a θ-rule, which defines the properties of splicing base and splicing rules for a finite subset of Σ^+. With these definitions, the authors give some propositions which the infinite code sets can be generated from splicing bases under the splicing rules; the resulting code sets are strictly θ-compliant, strictly θ-free, and $\theta(k, m_1, m_2)$-subword compliant.

3.4.3 The bond-free property

Theoretically, the properties proposed in [29, 27, 19, 25] can prevent many cases of undesirable bonds; however in the real world, if the language we have constructed satisfies all the properties, the undesirable bonds can still occur. For example, the structure in Figure 3.5 can be formed.

```
5'- a g c t g c -3'
    | | | |   |
3'- t c g a t g -5'
```

Figure 3.5: A structure of DNA with a mismatch

In order to prevent the occurrence of this kind of situation, DNA languages have to satisfy more properties, such as the bond-free property. Before we present the bond-free property, let us consider the following situation. For the words of the same length, if we pick a DNA word, say *aaatcc*, to be in a DNA language, due to the properties presented in Section 3.4.1, we can not have the word *ggattt* in the same language. However, because of the possibility of the above structure, words, that are different from *ggattt* at one position, such as *tgattt* and *gaattt*, can still stick to *aaatcc* and form undesirable bonds. These words are in the Hamming ball, $H_1(ggattt)$. So, it is easy to see that, in order to prevent the undesirable bonds among words, a DNA language L not only has to satisfy the properties in Section 3.4.1 but also the $(\tau, H_{d,k})$-bond-free property [28]: for any two subwords u and v of L of length k, that is $u, v \in \mathrm{Sub}_k(L)$, $\tau(u) \notin H_d(v)$.

3.4.4 Global constraints for DNA languages design

In addition to the presented properties, we need some other constraints to help us better operate the DNA sequences, such as the GC-ratio constraint which ensures that all the DNA sequences can melt at the same temperature. We often consider the following two constraints for DNA languages design.

1. The *GC-ratio constraint*: the ratio of the number of occurrences of g and c bases in a DNA word over the length of the DNA word must lie in a certain range to ensure similar thermodynamic characteristics among DNA words.

2. The *continuity constraint*: The same base should not appear continuously; otherwise, a reaction will not be well controllable since the structure of DNA will become unstable. It is not clear from the literature how long a string of equal bases should be to violate this constraint. In our system implementation, we let this be a parameter that can be specified by the user.

These two constraints require each entire DNA word to have certain properties, no matter how long the DNA words are. The bond-free property has no requirement for the word length as well. In general, we call them global constraints. In the next section, we are going to present some constraints that focus on each segment of DNA words of certain fixed length.

3.5 Local constraints for DNA language design

In the literature, the proposed methods for designing DNA languages usually require that each word w of the DNA language is made of shorter words, that is, $w = w_1 w_2 \cdots w_n$, with each w_i being of some fixed length k. When we design a DNA language, we first need to design a set S, say, of these shorter words that satisfy some *local constraints*, and then construct the desired DNA language by combining the words in S. *We shall use two major operations for combining the words in S: the concatenation closure, S^+, and the subword closure, $S^{\otimes}$.* The following local constraints for S are often used in DNA word design:

1. The *Hamming constraint* with distance parameter d means that any two words w, x in S satisfy $H(w, x) > d$.

2. The *reverse-complement constraint* with parameter d means that for all pairs of words w, x in S (where w may be the same as x), $H(\tau(w), x) > d$. This constraint is also expressed using the equation $\tau(S) \cap H_d(S) = \emptyset$.

3. The *reverse constraint* with parameter d means that for all pairs of words w, x in S, $H(w, x^R) > d$.

3.6 Construction methods

We have presented some properties and local constraints for DNA language design. We present some construction methods for DNA languages satisfying different properties and local constraints in this section.

3.6.1 Bounds and construction methods for reverse codes and reverse-complement codes

In [35], the authors define reverse codes to satisfy the Hamming constraint and the reverse constraint, and define reverse-complement codes to satisfy the Hamming constraint and the reverse-complement constraint. They denote the maximum sizes of reverse codes by $A_4^R(n,d)$, and denote the maximum sizes of reverse-complement codes by $A_4^{RC}(n,d)$, where n is the length of the words in the codes. They present a close relationship between $A_4^R(n,d)$ and $A_4^{RC}(n,d)$:

$$A_4^{RC}(n,d) = A_4^R(n,d) \qquad \text{when } n \text{ is even, and}$$

$$A_4^R(n,d+1) \leq A_4^{RC}(n,d) \leq A_4^R(n,d-1) \qquad \text{when } n \text{ is odd.}$$

In the paper, the authors propose some lower and upper bounds on reverse codes and reverse-complement codes.

Theorem 1 *[35], For $n \geq 4$,*

$$A_q^R(n,3) \leq \frac{q^{\lceil n/2 \rceil} \sum_{i=2}^{\lfloor n/2 \rfloor} \binom{\lfloor n/2 \rfloor}{i} (q-1)^i}{2(1+4(q-2)+(n-4)(q-1))}.$$

where q is the size of alphabet, in [35], $q \in \{2,4\}$.

For a specific condition, $d = 1$, the authors of [35] give a construction method for reverse code that is optimal for even n, and close to optimal for odd n. The upper bounds for this kind of codes are as follows:

Theorem 2 *[35] (d=1 Construction)*

$$A_q^R(n,2) = q^{n-1}/2 \qquad \textit{when } n \textit{ is even and } a \in \{2,4\}, \textit{ and}$$

$$A_q^R(n,2) \geq \frac{q^{n-1} - q^{\lfloor n/2 \rfloor}}{2} \qquad \textit{when } n \textit{ is odd and } q \in \{2,4\}.$$

The construction method for reverse codes, where $d = 1$ and n is even, is as follows: First partition the base case (Σ^2) into four parts: $S_1^2 = \{aa,\ cc,\ gg,\ tt\}$, $S_2^2 = \{ac,\ ca,\ gt,\ tg\}$, $S_3^2 = \{ag,\ ga,\ ct,\ tc\}$, $S_4^2 = \{at,\ ta,\ cg,\ gc\}$.

For the induction case, S_1^n contains all of the palindromes, which are words in the form of $w = xx^R$.

$$S_1^{n+2} = S_1^n \cdot S_1^2 \cup S_2^n \cdot S_2^2 \cup S_3^n \cdot S_3^2 \cup S_4^n \cdot S_4^2,$$

$$S_2^{n+2} = S_1^n \cdot S_2^2 \cup S_2^n \cdot S_3^2 \cup S_3^n \cdot S_4^2 \cup S_4^n \cdot S_1^2,$$

$$S_3^{n+2} = S_1^n \cdot S_3^2 \cup S_2^n \cdot S_4^2 \cup S_3^n \cdot S_1^2 \cup S_4^n \cdot S_2^2,$$

$$S_4^{n+2} = S_1^n \cdot S_4^2 \cup S_2^n \cdot S_1^2 \cup S_3^n \cdot S_2^2 \cup S_4^n \cdot S_3^2,$$

where $A \cdot B = \{pwq \mid w \in A, pq \in B, |p| = |q| = 1\}$. By first removing these palindromes from each subset and then dropping half of the remaining words (either a word or its reversal), we can obtain four reverse codes with $d = 1$. If we complement the second half of each word, we can obtain four reverse-complement codes with $d = 1$.

In [35], the construction methods for reverse codes and reverse-complement codes for odd n are similar to the methods for even n.

3.6.2 Template method

Because different experiments require different kinds of DNA languages, it seems impossible to design a DNA language that can be used as an all-purpose library of DNA words. The authors of [4] propose the template method that systematically generates a set of words of length k. This method is a trade-off between the tightness of those constraints and the number of words, which means that the word sets generated by

the template method are not all the words satisfying one experimental requirement but satisfy several different experimental requirements.

In [4], the authors denote by $\langle x \rangle$ the subword of x without a terminal symbol at either end. For example, $\langle x \rangle = s_2 s_3 \cdots s_{k-1}$, when $x = s_1 s_2 s_3 \cdots s_k$. They denote by $H_M(x, y)$ the minimum Hamming distance between x and the $|y| - |x| + 1$ subwords of length $|x|$ in y. They use $||x||$ to denote the minimal number among $H(x, x^R)$, $H_M(x, \langle xx \rangle)$, and $H_M(x, \langle x^R x^R \rangle)$.

They define a template as follows:

Definition 2 *[4], A template is a binary word of certain length k. Each bit of the template indicates a possible symbol of the DNA alphabet. More specifically, 1 indicates either a or t, and 0 indicates either c or g.*

The problem they want to solve is as follows:

Problem 1 *[4], Design a set S of DNA words of length k such that any word $x \in S$ or its reverse complement mismatches in at least d positions from any other word $y \in S$ $(x \neq y)$ or the overlap region of two words, i.e. $\langle yz \rangle (y, z \in S)$. Moreover, all words must share the same GC content.*

Problem 1 can be decomposed into two subproblems:

1. Find a single template x satisfying $||x|| = d$,

2. Find an error-correcting binary code E of minimum distance d.

The purpose for using an error-correcting code is to choose either a or t for the template position $x_i = 1$, and either c or g for position $x_i = 0$ $(1 \leq i \leq k)$. The

DNA words can be derived as a product $S = \{x \bullet y \mid y \in E\}$ with the encoding rule, $\{11 \rightarrow a,\ 10 \rightarrow t,\ 01 \rightarrow g,$ and $00 \rightarrow c\}$.

The second subproblem has been studied in coding theory [34], and several kinds of codes can be used, so the authors concentrate on the first subproblem. Because searching good templates of length k from $O(2^k)$ candidates is time consuming, the authors propose a upper bound for finding templates of $||x|| \geq d$, which is much less than $O(2^k)$. Therefore, it is feasible to use an exhaustive method to search good templates.

3.6.3 Stochastic local search algorithm for DNA word design

Stochastic search algorithms [10] have been used successfully to construct binary codes for more than 10 years. In order to extract the most effective general principle on the design of stochastic algorithms for DNA or RNA in the application of stochastic local search, the authors of [43] design a stochastic algorithm for generating DNA words. The authors use an empirical methodology based on run-time distribution [17] to analyze simple stochastic local search algorithm. They use their algorithm to design DNA words satisfying GC-ratio constraint, Hamming constraint and reverse-complement constraint. Their algorithm is able to find a word set whose size matches or exceeds the theoretical results of [35]. The algorithm has been proven to generate high-quality sets of DNA words that can satisfy various combinations of combinatorial constraints, but it is not accurate enough.

3.6.4 Methods for bond-free languages

The authors of [28] use the reverse-complement local constraint to construct bond-free languages:

Theorem 3 *[28] Let S be a set of words of fixed length k, then each of the languages $S^{\otimes}$ and $S^{\oplus}$ is $(\tau, H_{d,k})$-bond-free iff*

$$\tau(S) \cap H_d(S) = \emptyset.$$

The DNA languages generated by the methods in the previous sections are not necessarily maximal. When a set S, $S \subseteq \Sigma^k$, is maximal satisfying $\tau(S) \cap H_d(S) = \emptyset$, then $S^{\otimes}$ is a maximal bond-free language.

The significance of a maximal language lies in our ability to use words with the same length to represent more information than that of those non-maximal languages. In addition, the same properties that the non-maximal languages have can still be kept.

In [28], the authors also introduce some other methods that we can use in this thesis, such as

Theorem 4 *[28], Let j and q be positive integers and let L be a subset of $\Sigma^{jq}\Sigma^*$. If L is $(\tau, H_{t,q})$-bond-free, for some integer $t \geq 0$, then it is also $(\tau, H_{d,k})$-bond-free, where, $d=j(t+1)-1$ and $k=jq$.*

For example, let $j = 4$ and $q = 5$ and let L be a subset of $\Sigma^{20}\Sigma^*$. If L is a $(\tau, H_{2,5})$-bond-free language, it is also a $(\tau, H_{11,20})$-bond-free language.

3.7 An experimental construction of DNA databases

The authors of [38] construct experimental large-scale DNA databases with associative search capability. The purpose of the experiment is to measure rates of various search errors, such as false positives from near-neighbor mismatches, partial matches, non-specific binding and false negatives from limit-of-detection problems. They generate a set of DNA words that satisfy some of the constraints introduced above and generate a set of testing data by connecting those DNA words to form longer data elements. However, those DNA words used in the experimental DNA database do not carry any information at all, because there is no efficient encoding method.

Chapter 4

The subword closure operation

Given a set S of words of a fixed length k representing some desired local constraints, the language $S^{\otimes}$ is the set of words in which each segment of a word of length k is in S so that the desired local constraints can be preserved within the words in $S^{\otimes}$. Because, in this thesis, we will design data encoding methods for DNA languages generated by the subword closure operation $\otimes$, we will first investigate some general properties of languages generated by the subword closure operation in this chapter.

4.1 Background information

The properties of subword closure operation are described explicitly by the following two lemmas in [28].

Lemma 1 *[28]. Let S be a language containing only words of the same length k, for some positive integer k. The following statements hold true.*

1. $S = \mathrm{Sub}_k(S^{\otimes})$.

2. *Let S_1 be a language containing only words of the same length k. Then $S_1 \subseteq S$ iff $S_1^{\otimes} \subseteq S^{\otimes}$. This implies that, if $S_1 \neq S$ then $S_1^{\otimes} \neq S^{\otimes}$.*

3. *If $S = \mathrm{Sub}_k(L)$*, for some language L, then $L \subseteq S^{\otimes}$.

Lemma 2 *[28]. Let T be a trie accepting only words of the same length. There is a DFA $T^{\otimes}$ of size $O(|T|)$ accepting the language $L(T)^{\otimes}$. Moreover, $T^{\otimes}$ can be constructed from T in time $O(|T|)$.*

4.2 The general problem

In the need of encoding methods for a language $S^{\otimes}$, where words in S are of a fixed length k and S represents a desired local constraint, we want to be able to encode arbitrary input words. However, it might not be practical to encode all the words in an input language directly into $S^{\otimes}$. For example, if we use the English alphabet E as the input alphabet and want to encode all the English words into $S^{\otimes}$, we must have more than $|E^n|$ words in $S^{\otimes}$, where n is the maximal length of English words that we will potentially use. Even though we probably will not use all of them, we still have to have all the words in $S^{\otimes}$ for each of the possible combinations of the English symbols, such as the meaningless combination *ttwss*. Since there is no limit for n, the number of words in $S^{\otimes}$ could be infinite. Therefore, the typical approach is that, when we want to encode a word $z \in E^+$ into $S^{\otimes}$, we define blocks of length m of the input language E^+, for some $m < n$, and encode each successive block of

length m into a word in $S^{\otimes}$ and then concatenate the encoded words. In this way, we only have to pre-define and construct $|E^m|$ words in $S^{\otimes}$. On the other hand, for the convenience for decoding, we need to design the words in $S^{\otimes}$ to be of a fixed length l, for some $l > k$, because, when we decode words in $S^{\otimes}$, we cut the concatenated words into the pre-defined words that express each input block of length m. If the lengths of these pre-defined words are not equal, it is hard to know how to cut the concatenated words. The procedures for encoding and decoding are as follows:

Encoding procedure:

$$z = z_1 z_2 \cdots z_x, \text{ with } |z_i| = m,$$

$$\text{then encode } z_i \longmapsto w_i,\ w_i \in S^{\otimes} \text{ and } |w_i| = l.$$

Decoding procedure:

$$w = w_1 w_2 \cdots w_x, \text{ with } |w_i| = l,$$

$$\text{then decode } w_i \longmapsto z_i$$

But the problem is that, in general, the concatenation $w_1 w_2 \cdots w_n$ is not in $S^{\otimes}$. (We will explain the problem explicitly in the following sections.) In this thesis, we propose a solution to this problem.

4.3 Generating B-blocks

As we have presented, if we concatenate two words in $S^{\otimes}$, the resulting word might not be in the same set $S^{\otimes}$.

Example 3

Let $S_1 = \{aaa, aac, aag, aca, acc, acg, aga, agc, agg, caa, cac, gaa\}$.

Obviously, the words *aacg* and *gaaa* are in $S_1^{\otimes}$. But, the concatenation of the two words *aacggaaa* is not in $S_1^{\otimes}$, because not all the subwords of length 3 are in S_1, for example, the subword *cgg* is not in S_1.

In order to encode data into $S^{\otimes}$, with the procedure of the previous section, first of all, we need to solve this problem. The reason why the resulting word is not in $S^{\otimes}$ is as follows: When we concatenate two words, w_1 and w_2, in $S^{\otimes}$, and the length of words in S is k, each subword w_3 of the resulting word of length $2k$, with $w_3 \in \text{Suff}_k(w_1)\text{Pref}_k(w_2)$, is an element of the concatenation of S and S. By definition, SS contains all the combinations of words in S including all the words in $S^{\otimes}$ of length $2k$. Therefore, $\text{Sub}_k(SS)$ contains all the words such as w_3. Some words in SS, however, are not in $S^{\otimes}$. These words cause the problem presented above.

We propose the concept of a B-block, B, to solve this problem. Each word in B is of a fixed length $l > k$ and belongs to $S^{\otimes}$. Moreover, the concatenation of any two words in B is still in the set $S^{\otimes}$.

We need to define two new sets, EN and BE_u. For each word u in the set S, we denote by BE_u the words z of S such that uz is in $S^{\otimes}$. EN denotes a subset of S. Given a set S in which the length of words is k, we pick any subset EN of S and define, for each $u \in EN$, the set $BE_u = \{z \in S \,|\, uz \in S^{\otimes}\}$. And then, we can define the B-block in which all the words are of length l, $l \geq k$, as follows:

Definition 3 *A B-block is a set B satisfying*

$$B = S^{\otimes} \cap BE\Sigma^{l-k} \cap \Sigma^{l-k}EN, \text{ where } BE = \bigcap_{u \in EN} BE_u$$

Note that any two words in B can be concatenated freely and the resulting word is ensured to be in $S^{\otimes}$. Indeed, any two words w_1, w_2 in B can be written as $w_1 = v_1u_1$ with $u_1 \in EN$, and $w_2 = z_2y_2$ with $z_2 \in BE_{u_1}$, which implies that $v_1u_1z_2y_2$ is in $S^{\otimes}$. We wish B to be as large as possible. Note also that if EN is large then BE is small, and vice versa.

It is not necessary for l to be greater than $2k$, so we can have the following structure.

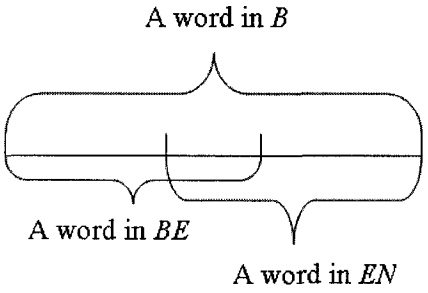

Figure 4.1: The structure of a word w, $w \in B$ and $|w| \leq 2k$

Example 4

We still use the same set, $S_1 = \{$ *aaa, caa, gaa, aca, cca, gca, aga, cga, gga, aac, cac, aag*$\}$. Table 4.1 shows, for each word u in S_1, the words of BE_u, that is all z in S such that uz is in $S_1^{\otimes}$.

u	BE_u
aaa	$\{$ aaa, caa, gaa, aca, aga, aac, aag, cac $\}$
caa	$\{$ aaa, caa, gaa, aca, aga, aac, aag, cac $\}$
gaa	$\{$ aaa, caa, gaa, aca, aga, aac, aag, cac $\}$
aca	$\{$ aaa, caa, aca, aga, aac, aag, cac $\}$
cca	$\{$ aaa, caa, aca, aga, aac, aag, cac $\}$
gca	$\{$ aaa, caa, aca, aga, aac, aag, cac $\}$
aga	$\{$ aaa, aca, aga, aac, aag, $\}$
cga	$\{$ aaa, aca, aga, aac, aag, $\}$
gga	$\{$ aaa, aca, aga, aac, aag, $\}$
aac	$\{$ aaa, aca, aac, aag, $\}$
cac	$\{$ aaa, aca, aac, aag, $\}$
aag	$\{$ aaa, aac, aag, $\}$

Table 4.1: word u and corresponding BE_u

We choose a subset EN of S_1, say $EN = \{aaa, aca, cac\}$. Then

$$\begin{aligned} BE &= \bigcap_{u \in EN} BE_u \\ &= BE_{aaa} \cap BE_{aca} \cap BE_{cac} \\ &= \{aaa, caa, gaa, aca, aga, aac, aag, cac\} \cap \{aaa, caa, aca, aga, aac, \\ &\quad aag, cac\} \cap \{aaa, aca, aac, aag\} \\ &= \{aaa, aca, aac, aag\} \end{aligned}$$

By definition of B-block, we ensure that any words beginning with words in BE can be concatenated to any words ending with word in EN.

The definition of B-block contains the following requirements about its words:

1. Are of length l and in $S^{\otimes}$,
2. Begin with words in BE, and
3. End with words in EN.

We can generate B satisfying the three requirements using a finite automaton. Figure 4.2 depicts the automaton for generating B in the above example, where $l = 5$.

4.4 Calculating the density of $S^{\otimes}$

Since $\otimes$ is a useful operation, it is nice to know some properties of this operation, such as the density of $S^{\otimes}$. The density of $S^{\otimes}$ is the function that returns the size of

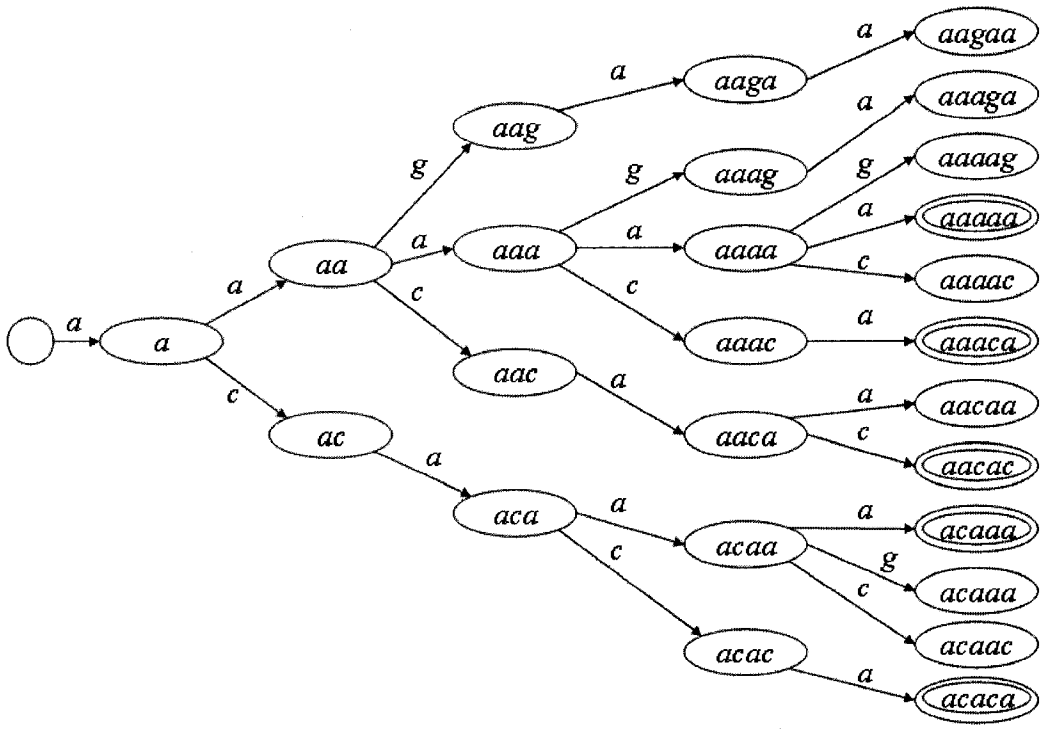

Figure 4.2: Automaton for generating words in B of length 5, where B is defined via the set S_1 and EN in Example 4

$S^{\otimes} \cap \Sigma^l$, for each $l \geq k$, where k is the length of words in S. For example, given an encoding need, we need to know how many words we can generate so that we can cover the input set. In this section, we obtain an ***exact*** recursive formula for the density of $S^{\otimes}$.

In this section, we need some notations to describe some subsets of $S^{\otimes}$. We denote by $S^{\otimes}(l)$ the set in which all the words are in $S^{\otimes}$ and of certain fixed length l, i.e., $S^{\otimes}(l) = S^{\otimes} \cap \Sigma^l$. We denote by $S^{\otimes}_w(l)$ the set in which all the words are in $S^{\otimes}$and of length l and have suffix w, i.e., $S^{\otimes}_w(l) = S^{\otimes}(l) \cap \Sigma^* w$.

$T(l)$ denotes the number of words in $S^{\otimes}$ of length l, i.e., $T(l) = |S^{\otimes} \cap \Sigma^l|$. $T_w(l)$

denotes the number of words in $S^{\otimes}$ of length l whose suffix is w, i.e., $T_w(l) = |S^{\otimes} \cap \Sigma^l \cap \Sigma^* w|$.

We let Q denote the set consisting of all the suffixes of length $k-1$ of set S, i.e. $Q = \Sigma^{k-1} \cap \text{Suff}(S)$.

Given a word $w \in S^{\otimes}$, by the definition of $S^{\otimes}$, we know that $\text{Sub}_k(w) \subseteq S$. If we want to concatenate a symbol $s \in \Sigma$ to w, we have to check the suffix of w of length $k-1$ to make sure that the concatenation of this suffix and s is in S. By checking the suffix of length $k-1$ of w of length l, we can know the number of words of length $l+1$ that are generated by concatenating symbols to w and whose suffix of length k are in the set S. This procedure is independent of the entire word, as it only depends on the suffix of words of length $k-1$. Therefore, we can apply this method recursively to generate words of desired length or of desired amount.

Example 5

Let $S_1 = \{aaa,\ caa,\ gaa,\ aca,\ cca,\ gca,\ aga,\ cga,\ gga,\ aac,\ cac,\ aag\ \}$, so that we obtain $Q_1 = \{aa,\ ca,\ ga,\ ac,\ ag\}$. Given a word, $caaa$ in $S_1^{\otimes}$, $\text{Suff}_2(caaa)$= aa, because aaa, aac, and aag are in S_1, we can concatenate a, c, and g to $caaa$ to form $caaaa$, $caaac$, and $caaag$. The generated words are in $S_1^{\otimes}$.

Since for any word w in $S^{\otimes}$, $\text{Suff}_k(w) \subseteq S$, $\text{Suff}_{k-1}(w) \subseteq Q$ holds., we can classify the words in $S^{\otimes}$ of certain length with the suffix of length $k-1$. For instance, when $l = 4$, we can partition the set $S_1^{\otimes}(4) = \{aaaa,\ aaac,\ aaag,\ caaa,\ caac,\ caag,\ gaaa,\ gaac,\ gaag,\ acaa,\ acac,\ agaa,\ aaca,\ aaga,\ caca\}$ into $S_1^{\otimes}(4) = \{aaaa,\ caaa,\ gaaa,\ agaa,\ acaa\} \cup \{aaca,\ caca\} \cup \{aaga\} \cup \{aaac,\ caac,\ gaac,\ acac\} \cup \{aaag,\ caag,\ gaag\}$.

i.e., $S_1^{\otimes}(4) = S_{1,aa}^{\otimes}(4) \cup S_{1,ca}^{\otimes}(4) \cup S_{1,ga}^{\otimes}(4) \cup S_{1,ac}^{\otimes}(4) \cup S_{1,ag}^{\otimes}(4)$. Also the cardinality of $S_1^{\otimes}(4)$ equals to the summation of $T_v(4)$, $v \in Q$, i.e., $T(4) = T_{aa}(4) + T_{ca}(4) + T_{ga}(4) + T_{ac}(4) + T_{ag}(4)$.

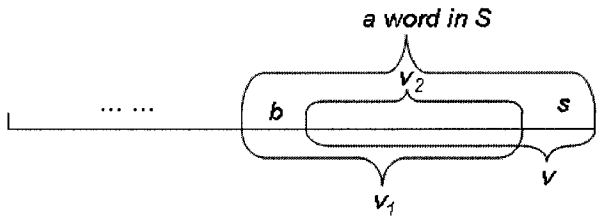

Figure 4.3: The end part of a word in $S^{\otimes}$

For each word in $S^{\otimes}$, its structure must be in the form of $\Sigma^* bv_2$, where $b \in \Sigma, bv_2 \in Q$ (Figure 4.3). If we want to concatenate another symbol s to the end of this word, $bv_2 s$ must be an element of S so that $v_2 s$ must be in Q. Therefore, words whose suffix of length $k-1$ is bv_2 can yield words whose suffix of length k are $bv_2 s \in S$. We define a set $\Sigma_{v_2 s} = \{b \in \Sigma \mid bv_2 s \in S\}$. Then, the following holds:

$$S_{v_2 s}^{\otimes}(l) = \bigcup_{b \in \Sigma_{v_2 s}} S_{bv_2}^{\otimes}(l-1)s, \text{ and}$$

$$T_{v_2 s}(l) = \sum_{b \in \Sigma_{v_2 s}} T_{bv_2}(l-1).$$

By the definitions in this section:

$$S^{\otimes}(l) = \bigcup_{v \in Q} S_v^{\otimes}(l), \text{ and}$$

$$T(l) = \sum_{v \in Q} T_v(l),$$

given a set S in which the word length is k and a length l, $l \geq k$, we can calculate the cardinality of $S^{\otimes}(l)$ recursively.

Example 6

Let $S_1 = \{\ aaa,\ caa,\ gaa,\ aca,\ cca,\ gca,\ aga,\ cga,\ gga,\ aac,\ cac,\ aag\}$. For instance, if we want to calculate $T_{aa}(l)$, we first need to find to the set Σ_{aa}. By definition of $\Sigma_{v_2 s}$, because $v_2 s = aa$, we can find symbols b such that $bv_2 \in Q$ and $bv_2 s \in S$. Because $aaa, caa, gaa \in S_1$ and $aa, ca, ga \in Q_1$, then $\Sigma_{aa} = \{a, c, g\}$. Therefore $S_{1,aa}(l) = S_{1,aa}(l-1)a \cup S_{1,ca}(l-1)a \cup S_{1,ga}(l-1)a$ and $T_{aa}(l) = T_{aa}(l-1) + T_{ca}(l-1) + T_{ga}(l-1)$. With the same method, $T_{ca}(l) = T_{ac}(l-1)$; $T_{ga}(l) = T_{ag}(l-1)$; $T_{ac}(l) = T_{ca}(l-1) + T_{aa}(l-1)$; $T_{ag}(l) = T_{aa}(l-1)$,

$$l = 3: \quad T_{aa}(3) = 3, \quad T_{ca}(3) = 3, \quad T_{ga}(3) = 3, \quad T_{ac}(3) = 2, \quad T_{ag}(3) = 1$$

$$l = 4: \quad T_{aa}(4) = 9, \quad T_{ca}(4) = 2, \quad T_{ga}(4) = 1, \quad T_{ac}(4) = 6, \quad T_{ag}(4) = 3$$

$$l = 5: \quad T_{aa}(5) = 12, \quad T_{ca}(5) = 6, \quad T_{ga}(5) = 3, \quad T_{ac}(5) = 11, \quad T_{ag}(5) = 9$$

, the size of $S_1^{\otimes}(5)$ is $T(5) = 41$.

In this thesis, we are concerned more with B, which is a subset of $S^{\otimes}$. We want to apply the recursive formula to calculate the size of B. However, because B is not only defined via S but also EN, we find that, if we randomly pick a subset of S as EN, it is difficult to calculate the size of B. The reason is as follows: we take the sets in Example 4 and the resulting formulas in Example 6 for example, where $BE = \{aaa, aca, aac, aag\}$ and $EN = \{aaa, aca, cac\}$. We can have $T_{aa}(3) = 1, T_{ca}(3) = 1, T_{ga}(3) = 0, T_{ac}(3) = 1$, and $T_{ag}(3) = 1$, and then apply the recursive formulas to calculate the size of B of a length l. We can use the same recursive formulas, because they are fixed for a particular set S and they are independent of EN. However, we have a problem in the last iteration. When we calculate $T_{aa}(l)$, we

only know how many words end with suffix *aa*. but, this information is not enough, because there two kinds of words that end with *aa*: words that end with *aaa* and words that end with *gaa* or *caa*, and we only want words that end with *aaa* to be in B. Therefore, we have to distinguish the two kinds of words from each other. It seems it is difficult to do so. Fortunately, if we choose EN with the guidelines in Section 6.4, we not only can calculate the size of B but also can have a larger set of B, which is what we wish to have.

Chapter 5

Investigating communicating cycles in automata

In Chapter 4, The method for producing the set of blocks B is non-deterministic. After obtaining a deterministic automaton for B, we do not know if the size of B is large enough to encode words in a data set. In this chapter, we propose a method for investigating if a given deterministic automaton can produce an arbitrarily large B.

5.1 Definitions

A deterministic automaton has a finite set of states K and an alphabet Σ. A *path* in a deterministic automaton is denoted as $[p_1, v_1, p_2, v_2, \cdots, p_n, v_n, p_{n+1}]$, where p_i is a state in K and v_i is a word that consist of symbols in Σ. In deterministic automata, a path denoted by $[p_1, v, p_2]$ is unique. If in the above path $p_{n+1} = p_1$, the path is called a *cycle*. If in a cycle every state is unique, this path is call a *simple cycle*. For example,

path $[p_1, x_1, p_1, x_2, p_1]$ is a cycle, but it is not a simple cycle. Two cycles $[p_1, v_1, p_1]$ and $[p_2, v_2, p_2]$ are *equivalent* if they can be written in the form $[p_1, w_1, p_2, w_2, p_1]$ and $[p_2, w_2, p_1, w_1, p_2]$ respectively.

If a deterministic automaton has two non-equivalent simple cycles $[p_1, w_1, p_1]$ and $[p_2, w_2, p_2]$ and a path $[p_1, u_1, p_2]$, these two cycles are called *a pair of communicating cycles.* Moreover, if there is no path such as $[p_2, u_2, p_1]$, the two cycles are called a pair of 1-way communicating cycles; if there exists a path $[p_2, u_2, p_1]$, the two cycles are called a pair of 2 way-communicating cycles. For example, in Figure 5.1, cycles $[p_1, v_1, p_1]$ and $[p_2, v_2, p_2]$ are a pair of 2-way communicating cycles; cycles $[p_2, v_2, p_2]$ and $[p_3, v_3, p_3]$ are a pair of 1-way communicating cycles.

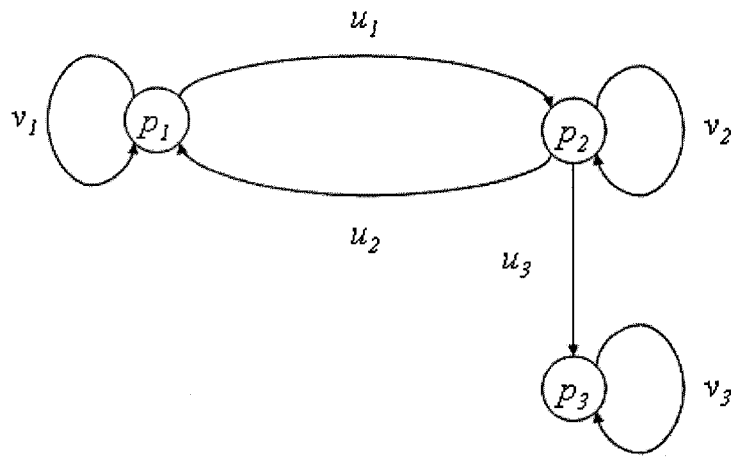

Figure 5.1: Communicating cycles

5.2 Cycle-intersection nodes

From Chapter 4, we know that B is a subset of $S^{\otimes}$ with a fixed word length. Let us consider the following theorem.

Theorem 5 *[7] If the automaton $T^{\otimes}$ has a pair of 2-way communicating cycles then the set B of a certain length can be chosen to be arbitrarily large, where T is a trie and accepts a set of words of a fixed length and satisfying a desired constraint.*

The problems we are going to investigate are the following: 1) given an automaton $T^{\otimes}$, how can we know whether the automaton has a pair of 2-way communicating cycles? 2) if an automaton $T^{\otimes}$ has a pair of 1-way communicating cycles or no communicating cycles at all, can we produce a B of a fixed length with an arbitrarily large size?

We propose a method for finding pairs of 2-way communicating cycles in a deterministic automaton as follows:

First, we define cycle-intersection nodes as follows:

Definition 4 *A cycle-intersection node is a node such that there are at least two non-equivalent simple cycles starting at the node.*

For example, in Figure 5.2, nodes p_1, p_2, and p_4 are cycle-intersection nodes; p_3 and p_5 are not. This is because, for instance, there are two non-equivalent simple cycles starting at p_1, $[p_1, v_1, p_2, v_2, p_3, v_3, p_4, v_6, p_1]$ and $[p_1, v_1, p_2, v_5, p_5, v_4, p_4, v_6, p_1]$. Therefore, p_1 is a cycle-intersection node.

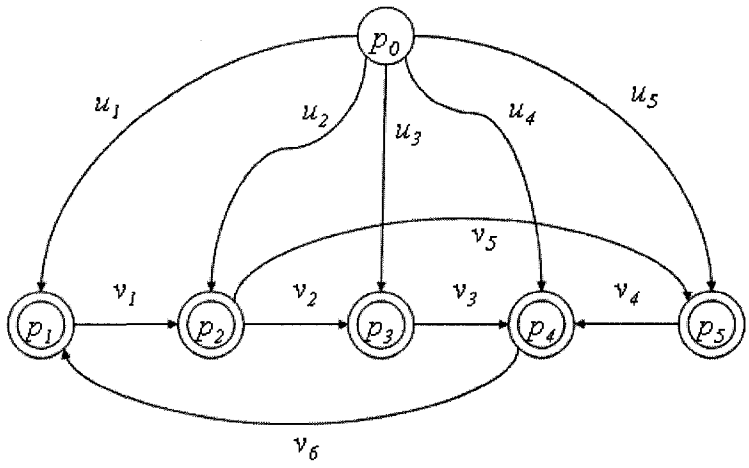

Figure 5.2: A deterministic automaton that has cycle-intersection nodes

Then, we can use the following theorem to check if a deterministic automaton contains a pair of 2-way communicating cycles.

Theorem 6 *A deterministic automaton contains a pair of 2-way communicating cycles, if and only if it has at least one cycle-intersection node.*

Proof:

The if part follows easily from the definition of the cycle-intersection node.

For the only if part, consider a pair of 2-way communicating cycles $[P, w, P]$ and $[Q, x, Q]$. There are two cases.

Case 1: The two cycles have no common state. Let $[P, u, Q, v, P]$ be a shortest cycle from P to P that passes via Q. If this cycle is simple then P is a cycle-intersection node. (as Q is not in $[P, w, P]$). If the cycle is not simple there is a state R such that the cycle is of the form

$$[P, u_1, R, u_2, Q, v_1, R, v_2, P].$$

We choose R to be the first state from left to right with this property. Then $[P, u_1, R, v_2, P]$ is a simple cycle. Let $[R, z_1, Q, z_2, R]$ be a simple cycle involving R and Q – this exists as $[R, u_2, Q, v_1, R]$ is a cycle involving R and Q. Then R is a cycle-intersection node.

Case 2: The two cycles have at least one common state. Let R be such a state. Then $[P, w, P]$ can be written as $[P, w_1, R, w_2, P]$ and $[Q, x, Q]$ as $[Q, x_1, R, x_2, Q]$. Also, these cycles are equivalent, respectively, to $[R, w_2, P, w_1, R]$ and $[R, x_2, Q, x_1, R]$. Then R is a cycle-intersection node. □

We will provide an algorithm for checking if a deterministic automaton has at least one cycle-intersection node in Section 5.3 with the time complexity analysis.

The next problem we want to consider is that, if a deterministic automaton does not contain any pair of 2-way communicating cycles, whether we can produce an arbitrarily large B. We conjecture that the following statement is true, but the proof is incomplete at this point.

Statement 1 *The set B of a certain length can be chosen to be arbitrarily large, if and only if the automaton $T^{\otimes}$ has a pair of 2-way communicating cycles, where T is the trie accepting the set of words S representing a desired constraint.*

If a $T^{\otimes}$ contains a pair of 2-way communicating cycles and the cycle-intersection node P of these two cycles is an accepting state, the following lemma ensures that we can produce an arbitrarily large B.

Lemma 3 *Let A be a deterministic automaton and n be a positive integer. If A has a pair of 2-way communicating cycles in the form of $[P_1, v, P_1]$ and $[P_1, u, P_1]$, then*

from the state P_1 to P_1, n words that are of the same length and pairwise different can be produced.

Proof:

Follows easily from the following lemma.□

Lemma 4 *[7], Let A be a deterministic automaton and n be a positive integer. If A has a pair of communicating cycles, then these cycles can be chosen to have the form $[P_1, v_1, P_1]$ and $[P_2, v_2, P_2]$, and satisfy the following property: There are n paths in A of the form*

$$[P_1, v_1^{r_i} u_1, P_2, v_2^{t_i}, P_2],$$

for i = 0, $\cdots$, n-1, such that the n words formed along these paths have the same length and are pairwise different.

Indeed, the words produced by Lemma 3 from $T^{\otimes}$ are in B, since they are recognized by $T^{\otimes}$ and of the same length. Moreover, the concatenation of any two resulting words, w_1 and w_2, is in the form of $[P, w_1, P, w_2, P]$ where w_1 and w_2 start at the cycle-intersection node P and P is an accepting state. As a result, this concatenation will also be recognized by $T^{\otimes}$.

5.3 An algorithm for checking cycle-intersection nodes and the time complexity analysis

In this section, we provide an algorithm for checking if there is at least one cycle-intersection node in a given deterministic automaton. The correctness of the algo-

rithm depends on the following lemma.

Lemma 5 *If a deterministic automaton has two different simple paths that start at the same state, then there is an edge that occurs in one of these paths but not in the other.*

Proof:

Let $P = [p_0, a_1, p_1, \cdots, a_k, p_k]$ and $P' = [p'_0, a'_1, p'_1, \cdots, a'_l, p'_l]$ be two different simple paths with $p_0 = p'_0$. As the paths are simple, each edge $[p_{i-1}, a_i, p_i]$ of P is unique, and similarly for P'. If $k \neq l$ then one of the paths is longer and the statement is true. So assume that $k = l$, but suppose that the two paths consist of the same edges. Let i be the first index with $[p_{i-1}, a_i, p_i] \neq [p'_{i-1}, a'_i, p'_i]$. The edge $[p'_{i-1}, a'_i, p'_i]$ must appear as $[p_{j-1}, a_j, p_j]$ in P, for some index $j > i$. Then the two paths can be written as

$$P = [p_0, x, p_{i-1}, a_i, p_i, \cdots, p_{j-1}, a_j, p_j, y, p_k] \text{ and } P' = [p'_0, x', p_{i-1}, a'_i, p'_i, y', p'_k],$$

such that $x = x'$ and $p_{i-1} = p'_{i-1}$ – otherwise, $i > 1$ and $[p_{i-2}, a_{i-1}, p_{i-1}] \neq [p'_{i-2}, a'_{i-1}, p'_{i-1}]$. But then $p_{i-1} = p_{j-1}$, which contradicts the fact that P is simple. □

The algorithm is as follows:

Input: a deterministic automaton A.

Output: Yes, if there is at least one cycle-intersection node; No, there is no cycle-intersection node.

for each node n in the automaton **do**

make a copy of A and call it A'

add a new node n' to A' and re-direct all the incoming labels of n to n' in A'

apply a search algorithm (e.g. breadth-first or depth-first) to find a path from n to n'

if there is no such a path

there is no cycle containing n

continue

for each edge e_i in the resulting path **do**

make a copy of A' and call it A'_i

remove e_i from A'_i and apply the search algorithm to find a path from n to n'

if there is a path from n to n'

n is a cycle-intersection node

return Yes

return No

We provide a time complexity analysis of the above algorithm based on the Breadth-first search algorithm. The time complexity of the Breadth-first search algorithm is $O(N+E)$ [6], where N is the number of nodes in the deterministic automaton and E is the number of edges in the automaton. However, the time comlexity of the algorithm, in our case, is $O(E)$, since the automaton in this research will always be connected.

In the algorithm, we check to see if there is at least a cycle-intersection node. We have to check all the nodes in a deterministic automaton. For each node n in the automaton, we first try to find a path going from node n to node n'. The length of this

path is no greater than N, since this path should be a simple path. If there is such a path, we temporarily remove each edge e_i from the automaton in a for-loop, and then, apply Breadth-first search again to find another path going from node n to node n'. Therefore, the time complexity of the algorithm is $O(N(E + NE)) = O(N^2E)$.

Chapter 6

Construction methods for DNA languages with local constraints

Theory of automata is a powerful tool for sequential language design. In the literature review section, we reviewed some theoretical designs and construction methods for DNA languages. There are many valuable methods and ideas. However, few of them address problems using the theory of automata. In this chapter, we will address problems using the theory of automata and demonstrate the significance of theory of automata in DNA language designs and constructions. We will propose some methods for constructing DNA languages that can be used for encoding purposes.

Using the subword closure operation, we can construct a DNA language satisfying a local constraint that is expressed by a set of words of a fixed length. In order to construct these DNA languages, we need to construct these sets for different local constraints. Moreover, suppose we want a language L that should satisfy the local

constraints expressed in sets S_1 and S_2, that is, L should be a subset of $S_1^{\otimes}$ and $S_2^{\otimes}$. If L_1 is a subset of $S_1^{\otimes}$ (resulting from some construction methods) and L_2 is subset of $S^{\otimes}$, then the desired language L is $L_1 \cap L_2$. Since, in this chapter, all the sets that express local constraints can be accepted by finite automata, L can be accepted by the intersection automaton of the finite automata accepting L_1 and L_2 (referring to Section 2.2.4).

In this chapter, all the languages are over the DNA alphabet.

6.1 An algorithm for generating set S for the bond-free property

Since we consider the bond-free property in this section, S has to satisfy $\tau(S) \cap H_d(S) = \emptyset$, so that $S^{\otimes}$ is the desired DNA language. We propose the following algorithm to generate S satisfying $\tau(S) \cap H_d(S) = \emptyset$. When the algorithm initializes DNA words, for word length in S being k, it needs to keep all the words in Σ^k in the memory. There are 4^k DNA words, so we can create an array with 4^k elements and use the index number of the array to represent each DNA word, since each index number has $2k$ binary bits. The method is as follows: we use 2 bits of binary codes to represent 1 DNA symbol, such as $00 \rightarrow a, 01 \rightarrow c, 10 \rightarrow g$, and $11 \rightarrow t$.

Example 7

Let $k = 2$, the index numbers (binary code) and the DNA words are:

Index number	DNA word	Index number	DNA word	Index number	DNA word	Index number	DNA word
0000	aa	0100	ca	1000	ga	1100	ta
0001	ac	0101	cc	1001	gc	1101	tc
0010	ag	0110	cg	1010	gg	1110	tg
0011	at	0111	ct	1011	gt	1111	tt

This method consumes the minimal memory space and has a very good memory access efficiency. The algorithm is as follows:

Procedure: generate a set S satisfying $\tau(S) \cap H_d(S) = \emptyset$

Input: the length of words (k), the Hamming distance of any two words in S (d), a start word (w).

Output: all the elements in S

$K := \Sigma^k$

Break the array into two parts; w is at the beginning of the second part, exchange the two parts

for $i := 1$ to $|K|$ **do**

if the word[i] is not flagged **then**

generate all the words whose Hamming distance is less than or equal to d from word[i].

complement these words and reverse them

compare the complemented and reversed words with words in K, flag the words in K that are the same as the complemented and reversed words

end if

end for

return S, the non-flagged words

Note: for the same parameters: the word length k and the Hamming distance d, if we begin with different words in the set Σ^k to generate the first Hamming ball, the resulting sets and the cardinalities of these sets will be different.

Remark:

The set S_1 used in Chapter 4 is generated by the above algorithm with parameters: $k = 3$, $d = 1$, and $w = aaa$.

6.2 Construction methods for languages satisfying GC-ratio constraints

The GC-ratio constraint is that, the ratio of the summation of occurrences of gs and cs in a DNA word over the length of the DNA word must lie in a certain range to ensure similar thermodynamic characteristics among DNA words. We denote the number of gs and cs over a DNA word w by $N(w)_{g,c}$; similarly, the number of as and ts over a DNA word w is denoted by $N(w)_{a,t}$. Thereby, the GC-ratio constraints are

$$r_1 \leq \frac{N(w)_{g,c}}{l} \leq r_2, \text{ for any word } w$$

where r_1 and r_2 are the lower bound and upper bound of the required GC-ratio, and $|w| = l$.

6.2.1 Construction method for languages with general GC-ratio

Generally, we want a GC-ratio to be in a certain range from r_1 to r_2. We propose the following method to construct languages with a GC-ratio in the range from r_1 to r_2.

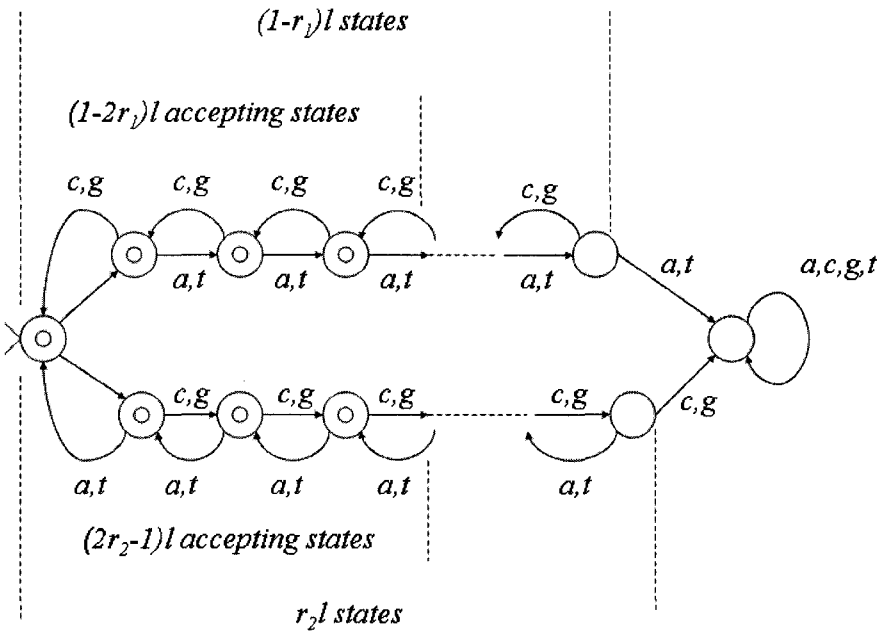

Figure 6.1: The automaton that accepts $F_{r_1,r_2,l}$

First, we define a finite automaton (Figure 6.1) that accepts the language $F_{r_1,r_2,l} = \{w \mid -(1-2r_1)l \leq N(w)_{g,c} - N(w)_{a,t} \leq (2r_2-1)l\}$. $F_{r_1,r_2,l}$ contains all the words w with the property that the difference between $N(w)_{g,c}$ and $N(w)_{a,t}$ is in the range from $-(1-2r_1)l$ to $(2r_2-1)l$. The word lengths are not necessarily the same. We define a set $R_{r_1,r_2,l} = \Sigma^l \cap F_{r_1,r_2,l}$. Then

Proposition 1 *$R_{r_1,r_2,l}$ satisfies the GC-ratio constraint.*

Proof:

For *GC*-ratio constraint, we want $r_1 \leq N(w)_{g,c}/l \leq r_2$, where $|w| = l$. That is $r_1 l \leq N(w)_{g,c} \leq r_2 l$, where $|w| = l$.

From the left part, we have

$$r_1 l \leq N(w)_{g,c} \ (1),$$

As $N(w)_{a,t} = l - N(w)_{g,c}$, we have

$$N(w)_{a,t} \leq l - r_1 l \ (2).$$

By adding (1) and (2), we get $r_1 l + N(w)_{a,t} \leq N(w)_{g,c} + l - r_1 l$, that is $-(1-2r_1)l \leq N(w)_{g,c} - N(w)_{a,t}$.

From the right part,

$$N(w)_{g,c} \leq r_2 l \ (3),$$

As $N(w)_{a,t} = l - N(w)_{g,c}$, we have

$$l - r_2 l \leq N(w)_{a,t} \ (4).$$

By adding (3) and (4), we get $N(w)_{g,c} + l - r_2 l \leq r_2 l + N(w)_{a,t}$, that is $N(w)_{g,c} - N(w)_{a,t} \leq (2r_2 - 1)l$.

Therefore, the condition that all the words in $R_{r_1,r_2,l}$ satisfy is the same as the condition required by the *GC*-ratio constraint. So, words in $R_{r_1,r_2,l}$ are all the desired words that are of length l and whose *GC*-ratio lies in the desired range. □

In Figure 6.1, we use $(1 - r_1)l$ states in the top branch and use $r_2 l$ states in the bottom branch. The reason is as follows: for a word of length l, when the lower bound of the *GC*-ratio is r_1, the maximal number of *a*s or *t*s is $(1 - r_1)l$. There could

be $(1-r_1)l$ continuous as and ts. They will take $(1-r_1)l$ states. If a word satisfies the GC-ratio constraint, the following symbols must be either cs or gs so that, in the automaton, these symbols can make the finite automaton go back to an accepting state. This is also the reason why, after the $(1-r_1)l$th state, if we read one more a or t , we will get into the sink state. Similarly, we need $r_2 l$ states in the bottom branch.

After constructing $R_{r_1,r_2,l}$, we can simply generate languages satisfying the GC-ratio constraint with $\otimes$ operation. $R^{\otimes}_{r_1,r_2,l}$ is the language that we want. By definition of $\otimes$ operation, every segment of length l of a word in $R^{\otimes}_{r_1,r_2,l}$ is in $R_{r_1,r_2,l}$. For convenience, let us take a word in $R^{\otimes}_{r_1,r_2,l}$ of length ml for example. We can break it into m segments, each of which is in $R_{r_1,r_2,l}$. Each segment has at most $r_2 l$ gs and cs and has at least $r_1 l$ gs and cs. Therefore, these m segments have at most $mr_2 l$ gs and cs and at least $mr_1 l$ gs and cs. The GC-ratio of this word in $R^{\otimes}_{r_1,r_2,l}$ is from $mr_1 l/ml$ to $mr_2 l/ml$, that is from r_1 to r_2, therefore, all the words in $R^{\otimes}_{r_1,r_2,l}$ satisfy GC-ratio constraint.

6.2.2 Construction method for GC-ratio of 50%

Sometimes, in practice, we want DNA languages of GC-ratio of exact by 50%. That will have $N(w)_{g,c} = N(w)_{a,t}$; since, when regarding the GC-ratio constraint, we consider g and c to be the same and a and t to be the same. In this case, pushdown automata are more convenient than finite automata for constructing DNA languages with a GC-ratio of 50%. The automaton that accepts languages such that the GC-ratio of any word is 50% is defined as follows:

$M_2 = (K, \Sigma, \Gamma, \Delta, s, F)$, where $K = \{s, q, f\}$, $\Sigma = \{a, c, g, t\}$, $\Gamma = \{\beta, \lambda\}$, $F = \{f\}$, Δ contains the following rules.

1. $((s, c, e), (q, \beta))$	2. $((s, g, e), (q, \beta))$	3. $((s, a, e), (q, \lambda))$
4. $((s, t, e), (q, \lambda))$	5. $((q, c, \beta), (q, \beta\beta))$	6. $((q, g, \beta), (q, \beta\beta))$
7. $((q, a, \beta), (q, e))$	8. $((q, t, \beta), (q, e))$	9. $((q, a, e), (q, \lambda))$
10. $((q, t, e), (q, \lambda))$	11. $((q, c, e), (q, \beta))$	12. $((q, g, e), (q, \beta))$
13. $((q, c, \lambda), (q, e))$	14. $((q, g, \lambda), (q, e))$	15. $((q, a, \lambda), (q, \lambda\lambda))$
16. $((q, t, \lambda), (q, \lambda\lambda))$	17. $((q, e, e), (f, e))$	

The language accepted by M_2 is $L_{50\%} = \{w \mid N(w)_{g,c} = N(w)_{a,t}\}$. For example, $agtccgt \notin L_{50\%}, agttcg \in L_{50\%}$. The set of words of a required length l in this language is $R_{50\%,l} = \Sigma^l \cap L_{50\%}$.

Example 8

Let us use a word *gcattagg* to illustrate how the above pushdown automaton works (Table 6.1).

State	Unread Input	Stack	Rule used
s	$gacttagg$	e	-
q	$acttagg$	β	2
q	$cttagg$	e	7
q	$ttagg$	β	11
q	$tagg$	e	8
q	agg	λ	10
q	gg	$\lambda\lambda$	15
q	g	λ	14
q	e	e	14
f	e	e	17

Table 6.1: The rules used for accepting *gacttagg*

6.3 Construction method for DNA language satisfying continuity constraints

The continuity constraint is that, the same base should not appear continuously, otherwise, a reaction will not be well controllable since the structure of DNA will become unstable [39]. It is convenient to again use the theory of automata to construct these kind of languages, because, in the theory of automata, if we have an automaton to accept a language with a certain property, when we complement the automaton, we can obtain the automaton that rejects the language with the property. Complementing a

deterministic automaton is done by switching the accepting states into non-accepting states and the non-accepting states into accepting states. We omit the construction for an automaton that accepts the language in which each word contains as least one segment of q continuous same symbols. The complemented automaton is depicted in Figure 6.2, which accepts languages $L_q = \{w \mid w$ does not contain q continuous same symbols$\}$.

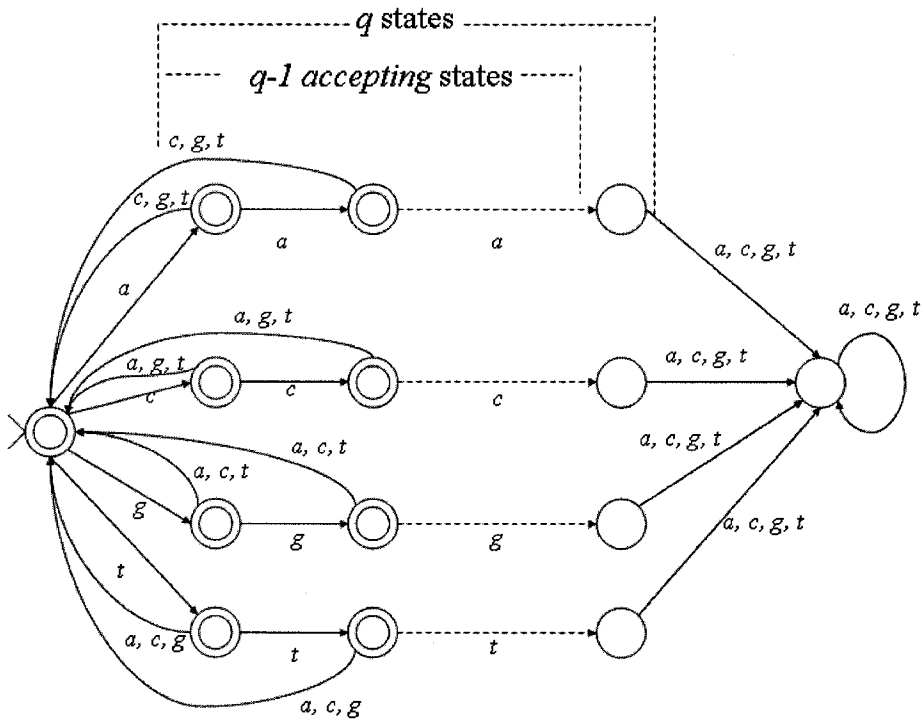

Figure 6.2: The automaton that accepts L_q

All words of length l that do not contain q continuous same symbols are in set $X_{q,l} = \Sigma^l \cap L_q$. So, any word in $X_{q,l}^{\otimes}$ satisfies the continuity constraint.

6.4 Guidelines for constructing B-blocks and Encoding methods for bond-free languages

As we have mentioned, if EN is large, BE is small, and vice versa. If we pick an arbitrary subset of S to be EN, it is difficult to calculate the size of B and the size of B might not be large enough. Since EN is a subset of S, which is an element of the power set 2^S, we have $2^{|S|} - 1$ choices (excluding $\emptyset$). However, in order to find a large set B generated from S, only a few ENs are appropriate.

If we look at the pairs of EN and BEs in detail, we can find that it is the suffixes of words in EN that decide the words in BE. Let us use the set generated in Section 4.3, $S_1 = \{aaa, aac, aag, aca, acc, acg, aga, agc, agg, caa, cac, gaa\}$, and the corresponding Table 4.1 to explain this. If we pick the word aac, $BE_{aac} = \{w \mid \text{Suff}_1(aca)\text{Pref}_2(w) \in S \text{ and } \text{Suff}_2(aca)\text{Pref}_1(w) \in S,\ w \in S\}$, i.e., $BE_{aac} = \{aaa, aca, aac, aag\}$. We can find that when we construct BE_us, we do not refer to the first symbol of u. Therefore, BE_us where us have the same suffix of length $k-1$ should be the same. We can see this result from Table 4.1. So, the first guideline is that,

If we pick a word u to be in EN, we also pick other words whose suffix of length k-1 is the same as $\text{Suff}_{k-1}(u)$,

so that we can have larger size of B. For example, if we only pick the word $u = aac$ to be in EN, the set $BE = \{aaa, aca, aac, aag\}$. This implies any word in B should begin with words in $\{aaa, aca, aac, aag\}$ and end with aac. If we add word cac into

EN, we will not delete any existing words. In addition, we can have words that begin with words in $\{aaa, aca, aac, aag\}$ and end with cac.

Another useful guideline needs to refer to Table 4.1:

After sorting the sets by the size of BE_us, it is recommended to choose continuous words from the top to the bottom in the first column in tables such as Table 4.1.

Example 9

The words from the top to the bottom in the first column of Table 4.1 are $\{aaa,\ caa,\ gaa,\ aca,\ cca,\ gca,\ aga,\ cga,\ gga,\ aac,\ cac,\ aag\}$. When we pick words for EN, using the first guideline, it is better to choose words such as $EN = \{aaa,\ caa,\ gaa\}$, $EN = \{aaa,\ caa,\ gaa,\ aca,\ cca,\ gca\}$, *or* $EN = \{aaa,\ caa,\ gaa,\ aca,\ cca,\ gca,\ aga, cga,\ gga\}$. The reason is as follows. If we pick EN such as $EN = \{aaa, caa, gaa, aag\}$, we add one more word in EN and lose 5 possible words in BE. If we roughly calculate the sizes of the two different Bs, we can find that the second guideline provides us more words in B. (We compare the sizes of Bs with different choices of EN in Table 6.2). Note: when we construct the sets ENs and BEs, we can not always obtain tables as Table 4.1, in which each BE_u is a subset of the BE_us above it.

With the first guideline, we have already solved the problem in Section 4.4. The problem was that it is hard to calculate the size of B, because we can decide whether a suffix belongs to an acceptable word. In Section 4.4, we classify B into several disjointed partitions by the suffix of length $k-1$. The first guideline suggests that, if we pick a word u to be in EN, we also pick other words whose suffix of length $k-1$

are the same as $\text{Suff}_{k-1}(u)$. Thereby, we pick all the words in one partition. So, when we calculate the size of B with the two guidelines, we start with words in BE, and, at the end, we simply sum up the size of partitions in which words in EN are.

Pairs	EN	BE	$T(10)$
Pair 1	$\{aaa, caa, gaa, aca, cca, gca, aga, cga, gga, aac, cac, aag\}$	$\{aaa, aac, aag\}$	150
Pair 2	$\{aaa, caa, gaa, aca, cca, gca, aga, cga, gga, aac, cac\}$	$\{aaa, aca, aac, aag\}$	180
Pair 3	$\{aaa, caa, gaa, aca, cca, gca, aga, cga, gga\}$	$\{aaa, aca, aga, aac, aag\}$	150
Pair 4	$\{aaa, caa, gaa, aca, cca, gca\}$	$\{aaa, caa, aca, aga, aac, aag, cac\}$	180
Pair 5	$\{aaa, caa, gaa\}$	$\{aaa, caa, gaa, aca, aga, aac, aag, cac\}$	150
Pair 6*	$\{aaa, caa, gaa, aga, cga, gga\}$	$\{aaa, aca, aga, aac, aag\}$	111

Table 6.2: The comparison of the sizes of Bs with different ENs

$\star$: Pairs 6 does not follow the second guideline, so the size of B of this pair is much smaller than the other pairs.

With the two guidelines, we have a few choices of pairs of EN and BE. For each pair, we can calculate the size of B of desired length. One of the results is the maximal size of a desired length.

At last, we complete our encoding method for bond-free languages. Given an encoding need, such as the number of input words, n, and the Hamming distance d. We can construct a B-block with equal word length satisfying $|B| \geq n$ and the Hamming constraint. Then, for each input segment of certain length, x, we can set a one-to-one encoding rule: $f : x \mapsto w$, $w \in B$, such that $f(x_1x_2\cdots x_n) = f(x_1)f(x_2)\cdots f(x_n) = w_1w_2\cdots w_n$, then $f(x_1x_2\cdots x_n) \in S^{\otimes}$. Indeed, if $f(x_1\ x_2\ \cdots\ x_{i-1}) \in S^{\otimes}$, then $f(x_1x_2\cdots x_{i-1})f(x_i) \in S^{\otimes}$, because $f(x_1x_2\cdots x_{i-1})$ ends with some $u \in EN$ and $f(x_i)$ starts with some z in BE, which implies that $uz \in S^{\otimes}$.

Since all the words in B are of a fixed length k and the encoding rules are one-to-one, when we decode the words in $S^{\otimes}$, we cut the words in $S^{\otimes}$ into segments of length k and decode each segment into an input segment.

Also, we can combine other constraints into our design. We summarize the general methods in the next section.

6.5 Combining local constraints

In this chapter, we have proposed several methods for constructing DNA languages satisfying different constraints. We should have already noticed that we can clearly express properties of DNA languages with the $\otimes$ operator. For instance, DNA languages with the bond-free property can easily be denoted by $S^{\otimes}$ with S satisfying $\tau(S) \cap H_d(S) = \emptyset$; DNA languages satisfying the GC-ratio constraint can be denoted as $R^{\otimes}_{r_1,r_2,l}$ with three parameters: the lower bound for the GC-ratio, the upper bound for the GC-ratio, and the length of words in R; also, we can easily construct DNA

languages satisfying continuity constraint, which is $X^{\otimes}_{q,l}$ with the maximal number of the continuous same symbols, q. When we want a language satisfying a combination of constraints, we don't have to reconstruct any new languages. We just take the intersection of some existing languages. For example, if we want a language satisfying bond-free property and also satisfying GC-ratio constraint, conceptually, we can have a language such as $L = S^{\otimes} \cap R^{\otimes}_{r_1,r_2,l}$. But, this method might not be very efficient, since the intersection automaton of two finite automata with K_1 and K_2 states has $|K_1| \times |K_2|$ states. When K_1 and K_2 are large, the intersection automaton is even larger. So, we need to implement this language in an indirect way, which is guided by $L = S^{\otimes} \cap R^{\otimes}_{r_1,r_2,l}$. For example, when we construct languages satisfying bond-free property and GC-ratio constraint, we first take the intersection of S and $R_{r_1,r_2,l}$ and then construct the language with $\otimes$ operation, that is $(S \cap R_{r_1,r_2,l})^{\otimes}$.

In this thesis, we address problems using the theory of automata. Because DNA languages are sets satisfying different combinations of constraints, it is very convenient to use operations in the theory of computation to manipulate words and languages.

Chapter 7

Implementation of the code generating system and experimental results

In this thesis, we design a websystem that produces DNA words satisfying desirable constraints. We have two implementations that produce DNA words. One applies the methods proposed in this thesis. We call it *the subword closure operation encoding system.* The other one applies Theorem 9 in [28]. We call it *the direct encoding system.*

Because the methods proposed in Chapter 4 are general and can be used for many kinds of local constraints, in the system, users are allowed to specify a set of fixed length DNA words representing a desired local constraint. The set can be specified in two ways. The first one is that users can upload a file containing the set of DNA

words. The second one is that users can produce the set by specifying parameters of constraints. As, in this thesis, we focus on the code design for DNA computing, we only provide an interface for specifying parameters of the DNA computing related constraints: the bond-free constraint, the GC-ratio constraint, and the continuity constraint. If the users want languages satisfying constraints other than these ones, they can write the local constraint set in a file and use the first approach to produce DNA words. Figure 7.1 depicts the interface for specifying the parameters of the DNA computing related constraints.

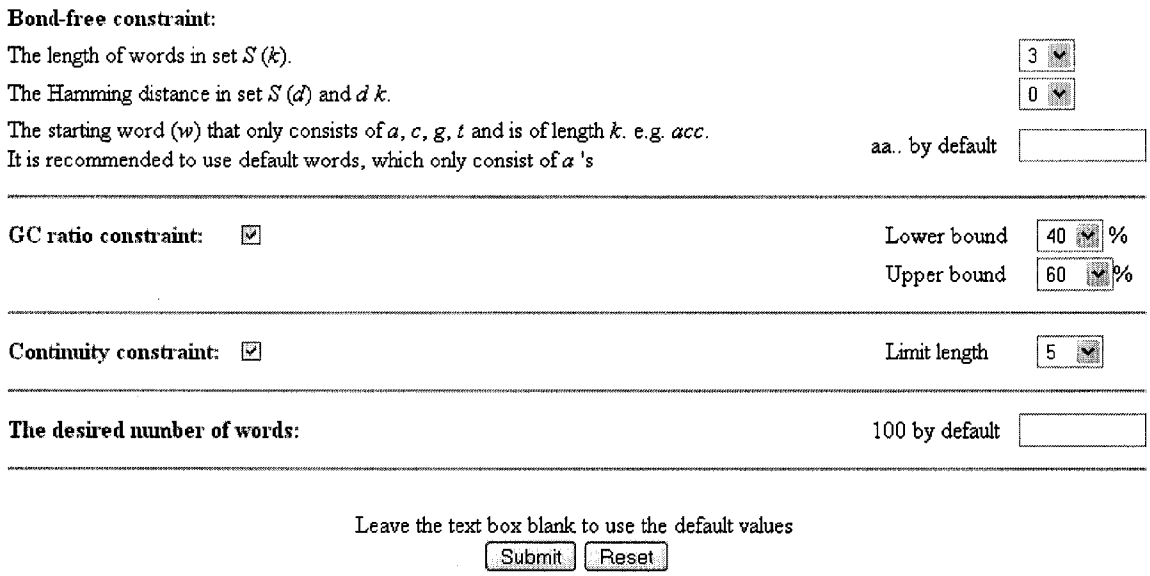

Figure 7.1: The web interface for specifying parameters of the DNA computing related constraints

As Theorem 9 in [28] works only for the DNA computing related constraints, the interface for the direct encoding system is similar with the above one.

In this chapter, we will present how the system works and the algorithms we use

in the system in detail. The URL of the system is http://cs.smu.ca/~b_cui/Thesis.

7.1 The flow of the subword closure operation encoding system

In our websystem, words representing arbitrary data are called *data words*, and the DNA words used for encoding data words are called *codewords*, or *blocks*. We use the symbol B for the set of these blocks. As we solve problems by using the theory of automata, in the implementation of the subword closure operation encoding system, we need to use systems that enable us to manipulate automata. Grail[1] is a well designed system and can perform almost all the manipulations on finite automata. Therefore, we want to integrate Grail into our system. Because Grail is implemented in C++ and it can be used as a library, we need to implement our system in C++ as well. Also, we want to have a web interface to allow users access our system through Internet. The architecture of our system, therefore, consists of two parts, a PHP implementation and a C++ implementation.

Since PHP provides us with a method to invoke executables, we can let the PHP implementation read parameters from users and pass the parameters to the C++ implementation, which does all the computations. Figure 7.2 depicts the flow chart of the subword closure operation encoding system.

[1]Grail is designed by the Department of Computer Science, University of Western Ontario

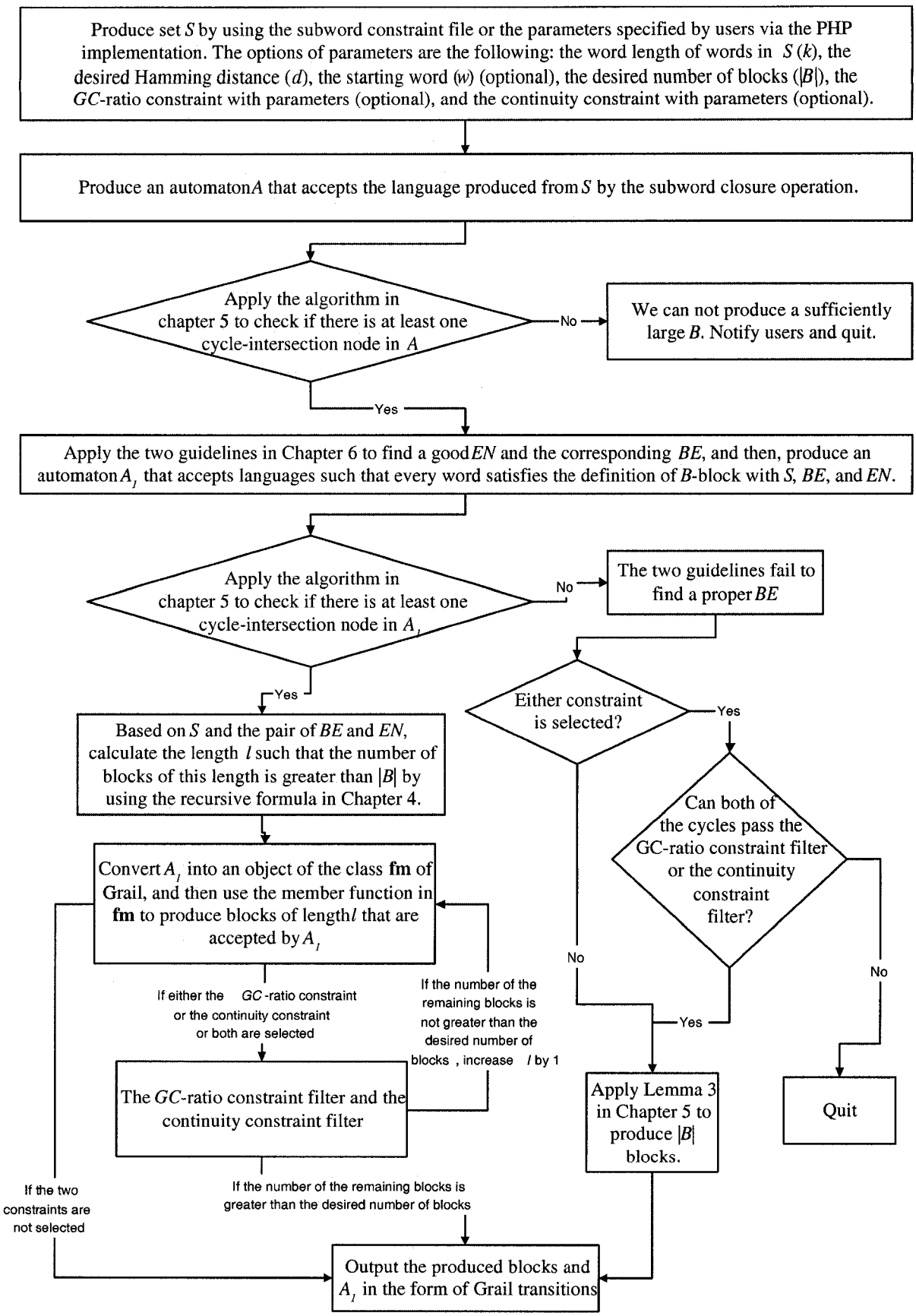

Figure 7.2: The flow chart of the subword closure operation encoding system

The PHP implementation simply provides some options for users to specify the parameters. After users have entered the values for the parameters, the PHP implementation passes the parameters to the C++ implementation by using the ` ` operator. If we write a command in between the two quotes of this operator, PHP will execute the command and assign the output of the command to a variable in PHP code as a string. For example, if we have the following files in the current directory in a Linux system,

a.out backup process.php project.html

the statement *$result = `ls`;* in process.php assigns the string *a.out backup process.php project.html* to the variable *$result.* Then, we can simply write the statement *echo $result;* to display the output of the Linux command to a web page.

In the C++ implementation, we first need to generate a set S, which can be specified in two ways. The first way is that we simply initialize S by using the words in the file that is provided by users as a local constraint set. The second way is that we initialize S to satisfy the bond-free constraint by using the parameters specified by users. In this way, we use the algorithm in Section 6.1 to generate the set.

After producing S, we construct a $T^{\otimes}$ for accepting $S^{\otimes}$ according to Lemma 2, and then, apply the method in Chapter 5 to check if there is a cycle-intersection node in $T^{\otimes}$. If there is no cycle-intersection node at all, we can not produce an arbitrarily large DNA word set to encode data words. In such a case, we will notify the user and quit from the system. If there is at least one cycle-intersection node, we will apply the two guidelines in Chapter 6 to produce DNA blocks. However, the guidelines

might not work all the times. If there is at least one cycle-intersection and the two guidelines fail, we will produce DNA words by using the pair of 2-way communicating cycles that contain the cycle-intersection node. In this way, according to Lemma 3 in Chapter 5, we will be able to produce an arbitrarily large B. However, this only works for the bond-free constraint. If the users require any additional constraint, we have to make sure that all the DNA words produced by Lemma 3 satisfy the additional constraints. In such a case, we know that all the words produced by Lemma 3 come from two words, u and v, that are recognized by the cycles in the pair of 2-way communicating cycles respectively. If the continuity constraint with a parameter q is required by the users, we know that, if either u or v or both do not satisfy the continuity constraint, all the DNA words produced by Lemma 3 will not satisfy the continuity constraint, since u and v will be segments of these DNA words. Also, we know the fact that any concatenation of two words satisfying the GC-ratio constraint with two parameters r_1 and r_2 will satisfy the same GC-ratio constraint. Therefore, if the GC-ratio constraint with parameters is required by the users, we have to pass u and v into the GC-ratio constraint filter. If either u or v or both do not satisfy the desired GC-ratio constraint, all the DNA words produced by Lemma 3 will not satisfy the desired GC-ratio constraint. This is why in the flow chart, if the users require any additional constraint, both u and v have to pass the filters, otherwise, we quit from the system.

When we produce a pair of BE and EN based on S, we refer to the two guidelines in Chapter 6. In this step, we need an auxiliary table that is similar with Table 4.1.

In theory, we can evaluate all the possible pairs of BE and ENs and pick the pair that provides the maximum number of blocks. However, this takes a huge amount of computations and might not be practical when the size of S is large. Therefore, we only evaluate some pairs of BE and ENs. The algorithm for picking up pairs of BE and ENs will be introduced later on.

In Figure 7.2, the fourth input parameter is the desired number of blocks, because this parameter is straightforward to users. We could set the parameter to be a desired length of DNA words, which makes the system much easier to implement and reduces the amount of computation significantly. However, this is not convenient for users, because the users, obviously, do not like to try many lengths until they find a length such that the number of DNA words of this length is greater than what they want. Therefore, after the users entered a desired number of blocks, we need to find a suitable length for them. As we have presented, if we select the pairs of BE and ENs following the two guidelines in Section 6.4, we can apply the recursive formula in Section 4.4 to calculate the size of B. And, thereby, we can know the suitable length l for the desired number of words.

In order to generate DNA words in B, we need to generate an automaton that accepts DNA words satisfying the definition of B-block. The algorithm for generating the automaton is described in the next section.

Once we obtain the automaton, we can convert it into an object of the class **fm** of Grail. We can use the functions in the class to generate DNA words of length l. As a result, these resulting DNA words satisfy the definition of B-block and are of

the same length l. Moreover, the number of these DNA blocks is greater than the desired number $|B|$.

Up to this point, the DNA words generated by the system only satisfy the initial local constraint S. If the users do not require the GC-ratio constraint and the continuity constraint, the system will display all the resulting DNA words on the web interface. However, since the GC-ratio constraint is a very important constraint in DNA language design, and the continuity constraint, sometimes, is also taken into the language designers' consideration, we need to be able to generate DNA words satisfying the two constraints. We design a filter for the two constraints. In Figure 7.2, we can see that, if either constraint or both constraints are required by the users, we need to pass the resulting DNA words of the previous steps through the filter. Figure 7.3 depicts how the filter works for the two constraints.

As we can see, there are two sub filters. The GC-ratio constraint filter simply removes all the DNA words that do not satisfy the GC-ratio constraint with the parameters specified by the users. Because all the DNA words passed into the filter are of the same length l, when we apply the method in Section 6.2.1, we only need two parameters, the lower bound r_1 and the upper bound r_2. The continuity constraint filter applies the method in Section 6.2. It generates a finite deterministic automaton to check if a DNA word satisfies the continuity constraint with the parameter specified by the users. If not, we remove the DNA word from the word set.

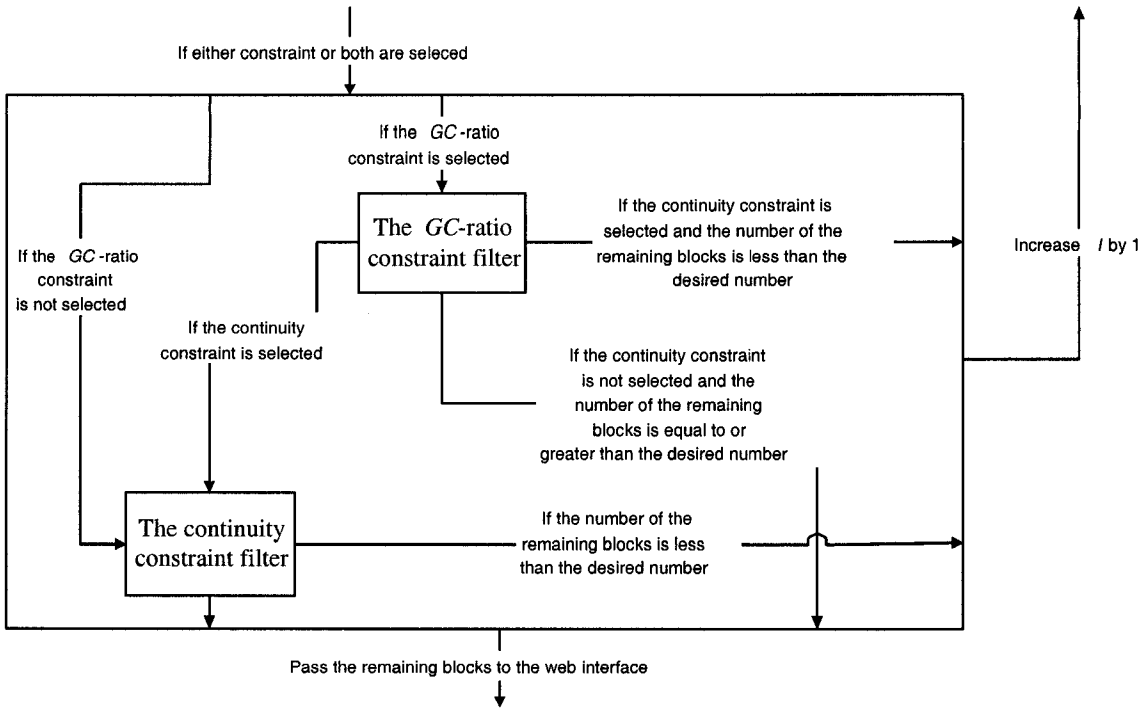

Figure 7.3: The filter for the *GC*-ratio constraint and the continuity constraint

7.2 The algorithms used in the implementation of the subword closure operation encoding system

In the C++ implementation of the system, we define the following classes, **S**, **Table**, **Pairs**, **Automaton**, **Continuity**, and **GCratio**, and integrate the following classes in Grail to our implementation, **state** and template classes: **array**, **String**, **inst**, **list**, **set** and **fm**. In this section, we are going to present and explain the algorithms in these classes.

Class **S** uses the algorithm in Section 6.1 to produce S and provides some accessors

for us to get the DNA words in the set. The algorithm for checking a cycle-intersection node is implemented as a member function of Class **S**. The data structure for holding a $T^{\otimes}$ is a linked list structure, which will be introduced later on.

Class **Table** maintains tables in the same format as Table 4.1. If we concatenate a word in a BE_u in the second column to the word in front of the set, the resulting word is in $S^{\otimes}$. The algorithm is simply the following: for each word u in S, we concatenate each word v in S to u. If all the segments of length k of the resulting word uv are in $S^{\otimes}$, we put v into BE_u. We sort the BE_us in the second column in the tables by the sizes of the sets.

After we obtain a table for S with specified parameters, in class **Pairs**, we pick pairs of BE and ENs according to the two guidelines in Section 6.4. Then, we evaluate the pairs to choose the one that can generate the largest set of words of a certain fixed length. Let us recall the guidelines for choosing pairs of BE and ENs. The first one is that, if we pick a word u to be in EN, we also pick other words whose suffix of length k-1 is the same as $\text{Suff}_{k-1}(u)$. The second guideline is that, after sorting the sets in tables such as Table 4.1 by the size of BE_us, it is recommended to choose continuous words from the top to the bottom in the first column. As we can see, the two guidelines are only for choosing ENs, because once we have a EN, the corresponding BE can be simply obtained by intersecting all the BE_us where $u \in EN$. Because the auxiliary table of S has as many columns as the size of S, when the size of S is large, the table has many columns as well. Therefore, even though we follow the two guidelines, the number of choices of possible ENs will still

be large. In order to improve the efficiency of our system, we do not generate all the possible pairs of BE and ENs based on the two guidelines. Instead, we use the following algorithm to generate a portion of them.

Procedure: based on S, select a pair of BE and EN that gives us a relatively large set B

Input: the auxiliary table generated based on S

Output: a pair of BE and EN

$index = 0$

$current = 0$

$bestBE$ and $bestEN$ are two empty sets

r is the number of rows

k is the length of words in S

if the size of BE_u in the first row is empty

    exit

else

    push the word in the first column in *current* row into EN

    push all the words in BE_u in *current* row into BE

    $index = current{+}{+}$

    while the $index <= r$ **and** the word in *index* row has the same suffix as the suffix of the word in *current* row

        push the word in the first column in *index* row into EN

        $index{+}{+}$

assign BE and EN to the $bestBE$ and $bestEN$, respectively

while $index <= r$ **and** BE_u in $index$ row is not empty

while $index <= r$ **and** BE_u in $index$ row is not a subset of BE

$index{+}{+}$

if $index <= r$ **and** BE_u in $index$ row is not empty

$current = index$

push the word in the first column in *current* row into EN

clear BE and push all the words in BE_u in *current* row into BE

while the $index <= r$ **and** the word in $index$ row has the same suffix as the suffix of the word in *current* row

push the word in the first column in $index$ row into EN

$index{+}{+}$

evaluate the pair of BE and EN by using the recursive formula in Section 4.4, If the current pair of BE and EN can produce a larger set of B with a fixed word length than the size of B with the same word length produced by $BestBE$ and $BestEN$, assign BE and EN to be $BestBE$ and $BestEN$

In this algorithm, we only use each word in the first column of the auxiliary table once. As we go through all the words, we keep pushing qualified words into EN. A qualified word must satisfy the following requirement: in the table, the BE_u after it must be a subset of all the BE_us of all the qualified words above it except for the

pair in the first row. In this way, we will lose some good choices of BE and ENs. We use the following example to describe the situations where the above algorithm misses good choices of BE and ENs.

Example 10

u	BE_u	The size of BE_u
cc	{ cc, ct, tg, tt, }	4
ct	{ ga, gt, tg, tt, }	4
gt	{ ga, gt, tg, tt, }	4
tt	{ ga, gt, tg, tt, }	4
tg	{ tg, tt, }	2
ga	{ }	0

Table 7.1: The auxiliary table for S_3 with $k = 2$, $d = 0$, $w = cc$

First, we generate a set S with the following parameters: $k = 2$, $d = 0$, and the start word $w = cc$. We obtain $S_3 = \{cc, ct, gt, tt, tg, ga\}$. Based on S_3, we obtain the auxiliary table in Table 7.1.

The above algorithm first pushes cc into EN and put cc, ct, tg, and tt into BE. Then, the algorithm evaluates the pair by using the recursive formula in Section 4.4. The result of the calculation shows that the size of B with a word length 10 is 1. Since, the BE_us in the Row 2, Row 3, and Row 4 are not subsets of the first BE_u, the algorithm skips the us in Row 2, Row 3, and Row 4. The BE_u in Row 5, $\{tg, tt\}$, is a subset of $\{cc, ct, tg, tt\}$, therefore, the u in Row 5, tg, is a qualified word.

The algorithm pushes tg into EN and updates BE to be $\{tg, tt\}$. The algorithm evaluates the new pair of BE and EN. The result of the calculation shows that the size of the new B with a word length 10 is 34. The BE_u in Row 6 is empty, therefore the algorithm is terminated. Between the two pairs of BE and ENs, the algorithm returns the second pair, because it can produce a larger B. However, if we follow the two guidelines, we can have a larger B. Because the BE_us in Rows 2, 3, and 4 are not subsets of the BE_u in Row 1, the algorithm simply skipped the us in these rows. But, if we refer to the two guidelines, we need to push the us in Rows 2, 3, and 4 into EN and intersect the BE_us in the first 4 rows. As a result, we will have the following two sets: $BE = \{tg, tt\}$ and $EN = \{cc, ct, gt, tt\}$. Because $S_3 = \{cc, ct, gt, tt, tg, ga\}$, we can get the following recursive formulas: $T_a(l) = T_g(l-1)$; $T_g(l) = T_t(l-1)$; $T_c(l) = T_c(l-1)$; $T_t(l) = T_c(l-1) + T_t(l-1) + T_g(l-1)$. The following example calculates the size of B with a word length 10, that is produced with $BE = \{tg, tt\}$ and $EN = \{cc, ct, gt, tt\}$.

$$
\begin{array}{llllll}
l = 2: & T_a(2) = 0, & T_t(2) = 1, & T_g(2) = 1, & T_c(2) = 0 \\
l = 3: & T_a(3) = 1, & T_t(3) = 2, & T_g(3) = 1, & T_c(3) = 0 \\
l = 4: & T_a(4) = 1, & T_t(4) = 3, & T_g(4) = 2, & T_c(4) = 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots \\
l = 10: & T_a(10) = 21, & T_t(10) = 55, & T_g(10) = 34, & T_c(10) = 0
\end{array}
$$

Since $EN = \{cc, ct, gt, tt\}$, it is clear that, among all the words in S_3, we only want DNA words ending with cs and ts. Therefore, we have $T_t(10) + T_c(10) = 55$ DNA words of length 10 in B. The size of this B is greater than the best B generated by

the above algorithm. Therefore, sometimes, the above algorithm misses some good choices of pairs of BE and ENs.

Actually, if we do not follow the two guidelines, we could have some even larger B with certain word lengths. We still use S_3 and pick $\{ct, gt, tt\}$ to be EN. Thereby, $BN = \{ga, gt, tg, tt\}$. Since we are using the same S_3, the recursive formulas remain the same. We can calculate the size of B with a word length 10 as follows:

$$\begin{array}{lllll} l=2: & T_a(2)=1, & T_t(2)=2, & T_g(2)=1, & T_c(2)=0 \\ l=3: & T_a(3)=1, & T_t(3)=3, & T_g(3)=2, & T_c(3)=0 \\ l=4: & T_a(4)=2, & T_t(4)=5, & T_g(4)=3, & T_c(4)=0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ l=10: & T_a(10)=34, & T_t(10)=89, & T_g(10)=55, & T_c(10)=0 \end{array}$$

Since $EN = \{ct, gt, tt\}$, we want words ending with t in S_3. Therefore, the size of B with a word length 10 is $T_t(10) = 89$. From this example, we can see that the two guidelines and the algorithm can not always provide us with the largest B, however, in most of the cases, the pairs of BE and ENs generated by the two guidelines and the algorithm do provide the largest B. The algorithm is a tradeoff between the optimal size of B and the ability to systematically and efficiently generate pairs of BE and ENs.

After obtaining a pair of BE and EN, class **Automaton** generates a deterministic automaton accepting words beginning with words in the BE and ending with words in the EN. Moreover, any segment of length k of the words accepted by the automaton is in the S that corresponds to the BE and EN. We should notice that, if $T^{\otimes}$ contains

at least one cycle-intersection node, but this automaton generated from BE and EN can not produce an arbitrarily large B, then this implies that the two guidelines failed to find a proper EN. We need to explain why we need to define a class that generates and stores deterministic automata as opposed to using the class **fm** of Grail. That is because we need to dynamically build up automata from BE and ENs. During the construction, we need more information about each state of automata than what class **fm** can provide. For example, as we will see in the next example, when we construct an automaton, we create some states of the automaton, construct a trie of the DNA words in BE, and then add links among the states that contain the words in S. In order to add the links, we need to store a DNA word in each state temporarily. But the states in Grail are only numbers. They can not hold any information about DNA words. This is why we need to define the class **Automaton** to help us construct automata. We can easily convert the automata stored in objects of class **Automaton** into objects of class **fm** afterwards.

We use the following example to explain how we construct an automaton from a set S and a pair of BE and EN and provide the algorithm for doing the construction later on.

Example 11

We generate $S_2 = \{aa, ca, ga, ac, cc, ag\}$ with the following parameters: $k = 2$, $d = 0$, and $w = aa$. We apply the algorithm for selecting a good pair BE_2 and EN_2 on S_2: $BE_2 = \{aa, ac, ag, ca, cc\}$ and $EN_2 = \{aa, ac, ga, ca, cc\}$. Since it is required that all the words accepted by the deterministic automaton must start with words in

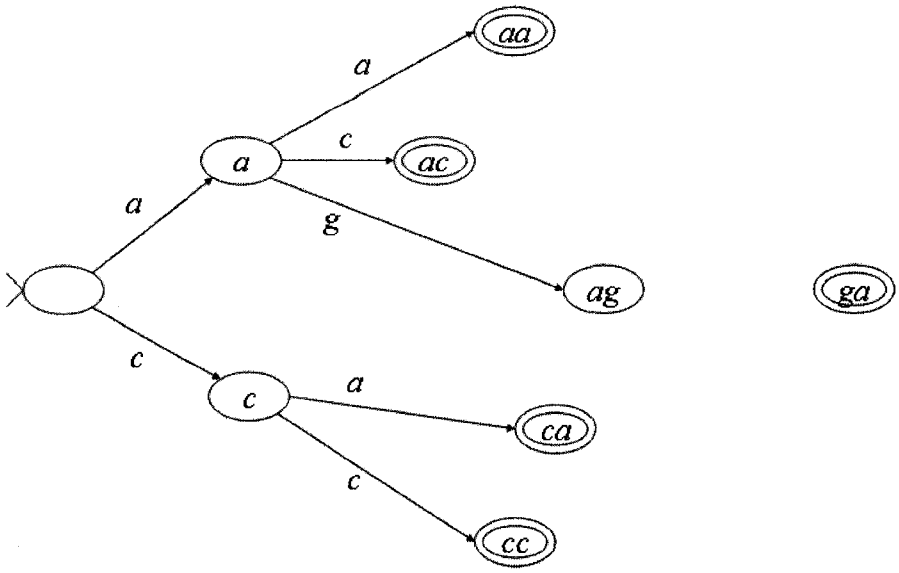

Figure 7.4: A trie for words in BE_2 and State for the other words in S_2

BE_2, we construct a trie with words in BE_2 and make the initial state of the trie to be the only initial state of the desired deterministic automaton. The next step is to make all the words in S_2, except for the words in BE_2, to be states in the automaton. In this case, we only have one word ga. The result of the above two steps is depicted in Figure 7.4.

The words in the states indicate that, if a word or a prefix of a word accepted by the automaton can reach a state, it must end with the word in the state. Since all the leaf states of the trie and the states with no link are words in S_2, if we link these states with proper labels, the result will be that any segment of length 2 of the words accepted by the automaton is in set S_2. The principle for adding links and labels is the following: we concatenate a symbol s in DNA alphabet to the word w in one state St_1. If $\text{Suff}_k(ws)$ is the word in another state St_2, we add a link from St_1 to

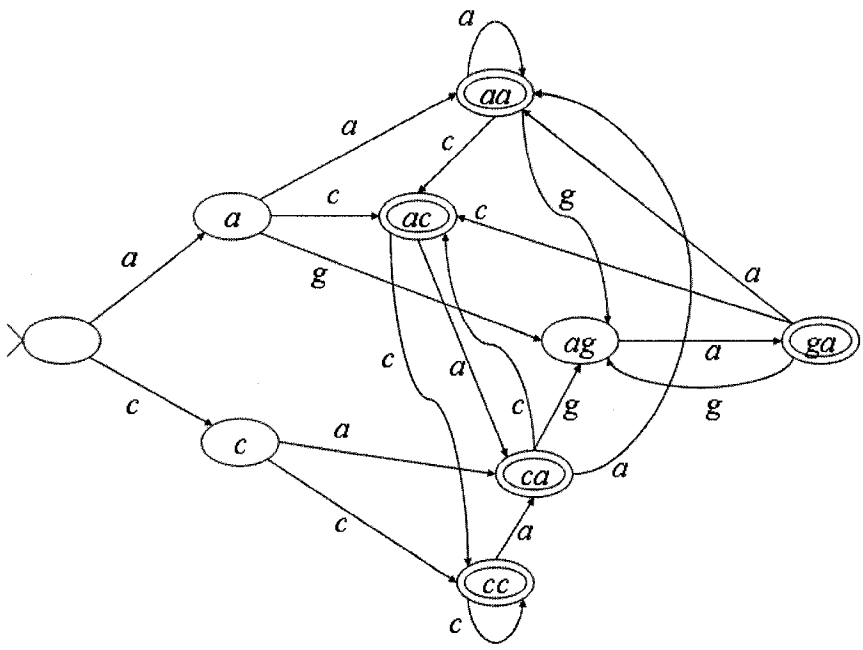

Figure 7.5: The automaton for accepting words such that each word begins with words in BE_2 and ends with words in EN_2, and moreover, any segment of length k is a word in S_2

St_2 with label s. For example, in Figure 7.4, we concatenate symbol c to the word aa in one of the states. Because $\text{Suff}_2(aac) = ac$ is the word in another state, we add a link from the state with aa to the state with ac and put label c on the link. After adding links, we make states containing words in EN_2 to be accepting states. As a result, any word accepted by the automaton ends with a word in EN. The resulting automaton for S_2, BE_2, and EN_2 is shown in Figure 7.5.

Before we introduce the algorithm following the above method for constructing automata, we need to introduce the data structure for storing automata. In this project, we use a linked structure for storing automata. There are two kinds of

nodes: state nodes and link nodes. State nodes are used for storing information about the states in automata. The data structure of state nodes is the following:

```
struct statenode
{
    int index;
    string word;
    bool initialstate;
    bool acceptingstate;
    linknode* nextlink;
    statenode* nextstate;
}
```

Link nodes are used for storing information of links from one state to another one. The data structure of link nodes is the following:

```
struct linknode
{
    char label;
    int index;
    linknode* nextlink;
}
```

Note: in the above figure, the first boolean variable in the state nodes is for the initial state, and the second boolean variable is for the accepting states.

In each state node, there are two node pointers. Nextstate is used for linking all

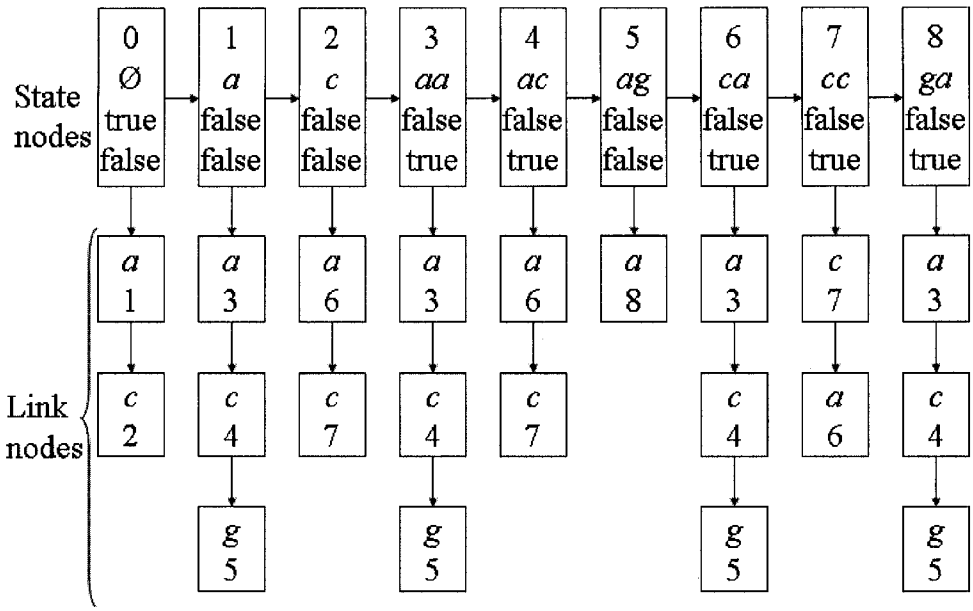

Figure 7.6: The linked structure for storing the automaton in Figure 7.5

the states together, so that we can easily go through all the states and quickly find a specific state. Nextlink maintains a list of link nodes that represent outgoing links from a state. For example, in Figure 7.6, we have a state node with index number 0 and two link nodes that represent the outgoing links from State 0. In the first link node, letter a and number 1 represent that there is an outgoing link from State 0 to State 1.

In the implementation, we use the following algorithm to construct a deterministic automaton from a set S and a pair of BE and EN.

Procedure: given a set S, and a pair of BE and EN, generate a deterministic automaton for accepting words such that each word begins with words in BE and ends with words in EN, and, moreover, any segment of length k is a word in S

Input: S, BE, and EN

Output: a deterministic automaton in a linked structure that represents the desired automaton

create state nodes for each word in S, and make the state an accepting state if the word in the state is in EN

we now construct a trie for the words in BE. At first, we put the prefixes of length $k-1$ of words in BE into a vector. In the vector, each word has to be unique.

while the length of the words in the vector is greater than 0

create state nodes for the words in the vector and link the newly created state nodes in front of the existing linked list of state nodes.

clear the vector and put the prefixes of length $length-1$ of the newly created state nodes into the vector, each word in the vector has to be unique.

$length--$

add the unique initial node, because each trie only has one initial node, and link it in front of the existing linked list of state nodes

add links among the states, if there is one outgoing link from a state represented by a state node, we add a link node to the linked list of link nodes maintained by the state node.

Reminder: recall that automaton $T^{\otimes}$ for recognizing words in $S^{\otimes}$ is also of a linked list structure, but the algorithm for constructing $T^{\otimes}$ from a set S is much simpler. We just construct a trie from S and add links among the leaf states. The algorithm is similar with the above one and is omitted.

Because we want to apply the existing functions in Grail to manipulate automata,

we need to convert the automata obtained from the above process into an object of class **fm**. In Grail, a deterministic automaton is stored as three sets: a set of initial states, a set of accepting states, and a set of instructions. An instruction is a transition from one state to another state on an input letter. States and instructions of an automaton are objects of class **state** and class **inst** respectively. Class **set** is a subclass of class **array**, which is a dynamic array template class and allows an unlimited number of items to be added. For example,

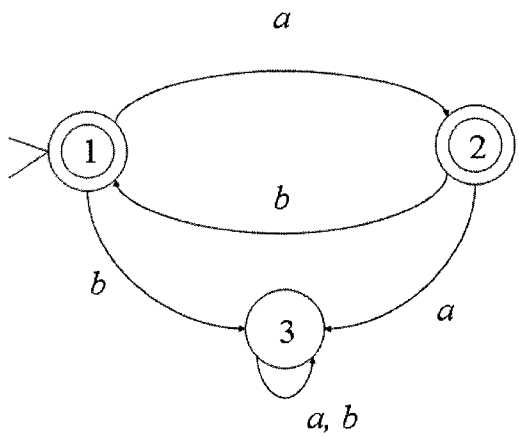

Figure 7.7: The automaton accepting the language $(ab)^*$

the automaton in Figure 7.7 is stored as follows:

initial state set = {State 1}

Accepting state set = {State 1, State 2}

Instruction set = {1 *a* 2, 1 *b* 3, 2 *a* 3, 2 *b* 1, 3 *a* 3, 3 *b* 3}

Converting a deterministic automaton from a linked structure to an object of class **fm** can be easily done, so we are not going to go into the details. The next step is to generate DNA words that are accepted by a deterministic automaton and of a

desired length. Class **fm** provides a member function, **enumerate**, to enumerate words accepted by the automaton stored in an object of the class. Each word is an object of class **String**. The words are sorted by the lengths of the words and the quantity of the resulting words is specified by one of the parameters of the function. In our project, we need words of a certain fixed length, therefore, we add a new member function, **enumerateDNA**, to enumerate DNA words of the certain fixed length. The length is specified by one of the parameters. We use the automaton in Figure 7.5 as an example to compare the difference between the two functions. We convert the automaton in the format shown in Figure 7.6 into an object of class **fm**. The results of the function calls with different parameters are shown in Table 7.2.

Up to this point, the words generated by function **enumerateDNA** of an object of class **fm** are words in B. They only satisfy the bond-free property. As we have presented, to enable words generated by our system to meet more experimental requirements, we design two filters for the GC-ratio constraint and the continuity constraint. In the GC-ratio constraint filter, we apply the method in Section 6.2.1. We construct deterministic automata with three parameters, r_1, r_2, and l, to remove words that do not satisfy the GC-ratio constraint from B. Similarly, in the continuity constraint filter, we apply the method in Section 6.2. We construct deterministic automata with two parameters, q and l, to remove words that do not satisfy the continuity constraint from B.

Function	enumerate	enumerateDNA
Parameter	10	3
Resulting words	aa	caa
	ac	cac
	ca	cca
	cc	ccc
	aaa	aaa
	aac	aac
	aca	aca
	acc	acc
	aga	aga
	caa	

Table 7.2: The outputs of the functions **enumerate** and **enumerateDNA** of an object of class **fm** with parameter 10 and parameter 3 respectively

7.3 Implementation of the direct encoding system

In this method, we apply Theorem 9 in [28] (shown further below) to produce DNA codewords (because both the input and the output are DNA words, we call the input DNA words the data words and call output DNA words the DNA codewords). Since the DNA codewords produced by this theorem only satisfy the bond-free constraint, in addition, we will pass the produced DNA codewords into the GC-ratio constraint filter and the continuity constraint filter depending on whether users want the DNA

codewords to satisfy these constraints.

Theorem 7 *in [28], Let I be a nonempty subset of $\{1, \ldots, k\}$ of cardinality $\lfloor k/2 \rfloor + 1 + \lfloor (d + k\%2)/2 \rfloor$. Then the language B^+ is $(\tau, H_{d,k})$-bond-free, where*

$$B = \{v \in \Sigma^k \mid \text{if } i \in I \text{ then } v[i] \in \{a, c\}\}.$$

The advantage of some codes defined by this theorem is that they can be used for encoding and decoding data in linear time. The authors of [28] provide an instance of the code B as follows:

$$B = \Sigma^{k-l}\{a, c\}^l$$

such that l is even, where $l = \lfloor k/2 \rfloor + 1 + \lfloor (d + k\%2)/2 \rfloor$. For example, this is true when d and k are even and $k+d+2$ is a multiple of 4. Let n be the quantity $k - l/2$. Every word $a_1 \cdots a_n$ in Σ^n can be encoded with a codeword in B as follows. Each symbol a_i is encoded as a_i, for $i = 1, \ldots, k - l$, and each symbol a_j is encoded as

$$aa, \text{ if } a_j = a;$$

$$ac, \text{ if } a_j = c;$$

$$ca, \text{ if } a_j = g;$$

$$cc, \text{ if } a_j = t;$$

for $j = k - l + 1, \ldots, n$. For example, if $k = 10$ and $d = 4$ then $l = 8$ and $n = 6$. This way the word *acctga* will be encoded as *acaccccaaa*.

Since, in the web system, users might select a pair of k and d such that $k + d + 2$ is not a multiple of 4. In this case, the value of l will not be an even number. As a result, the quantity of n will not be an integer. To avoid this kind of situation, when

calculating l, we will round it up to the nearest even number. This rounding will affect the number of the positions where the symbols remain the same, especially for small ks. For instance, if $k = 3$ and $d = 1$, then $l = \lfloor 3/2 \rfloor + 1 + \lfloor (1 + 3\%2)/2 \rfloor = \lfloor 1.5 \rfloor + 1 + \lfloor (1 + 1)/2 \rfloor = 1 + 1 + 1 = 3$, which is not an even number. If we round it up to the nearest even number 4, the number of the positions where the symbols remain the same will be $k - l = 3 - 4 = -1$. For this reason, in the implementation, we exclude $k = 3$ from the options of the value for k.

After users specified the values for k and d, we will calculate l and n. If the users do not require any additional constraints, we will simply follow the encoding/decoding rules presented above. If the users require the GC-ratio constraint or the continuity constraint, we produce all the words in Σ^n as data words, apply the encoding rules to produce the encoded DNA codewords, pass the DNA codewords through the filters, and then, list the remaining data words and DNA codewords in a look-up table. The users will only be allowed to enter combinations of the data words or the DNA codewords in the look-up table to perform encoding or decoding.

7.4 Experimental results of the subword closure operation encoding system

In this section, we will make use of our system with values of parameters and collect some data from the results.

To generate B-blocks for DNA computing related constraints, we need to construct

a set S. When we construct S, we need three parameters: the word length k, the Hamming distance d, and the starting word w, which is required by the algorithm in Section 6.1. In the algorithm, we generate all the DNA words of a fixed length and list them alphabetically. For example, all the words of length 2 are listed as follows: aa, ac, ag, at, ca, cc, cg, ct, ga, gc, gg, gt, ta, tc, tg, tt.

Here, we need to notice two things. The first one is that the starting word affects the resulting set. Let us explain the reason with the above example. If the starting word is aa, and the Hamming distance is set to be 0, we have to remove tt from the list, because $\tau(tt) = aa$. As a result, tt would not appear in S. Obviously, if the starting word is set to be tt, we will not have aa in S, for the same reason. The other thing we need to notice is that the order of removing words that violate the bond-free property affects the resulting set. The reason is almost the same as the reason for the starting word. When we construct S, we keep removing DNA words from the list that contains all the DNA words of a fixed length. If we have two words w and v and the Hamming distance between w^R and v is less than d, if we remove w first, v might appear in S; in the opposite way, if we remove v first, w might appear in S.

In the implementation of the algorithm in Section 6.1, we pick words sequentially from the beginning of lists as shown above, generate Hamming balls of the words, complement and reverse words in the Hamming balls, and remove the complemented and reversed words from the lists. After the procedure, it is very possible that the words in the front of the list stay in set S. They are the words that contain one or more as. For this reason, we suggest to use words that only consist of as as the

starting words. We also can see that words consisting of only as can provide larger sets of Bs – see Table 7.3 and Table 7.4.

w	4	5	6	w	4	5	6	w	4	5	6	w	4	5	6
aa	20	45	101	ca	1	1	1	ga	20	45	101	ta	1	1	1
ac	6	10	19	cc	2	3	5	gc	16	32	64	tc	1	1	1
ag	1	1	1	cg	16	32	64	gg	16	32	64	tg	1	1	1
at	1	1	1	ct	16	32	64	gt	5	8	13	tt	6	10	19

Table 7.3: The sizes of Bs of length 4, 5, and 6 generated with different starting words w, where k =2 and $d = 0$.

In Table 7.5, we compare the sizes of B and the densities of $S^{\otimes}$ of different lengths produced from different Ss satisfying values of parameters.

w	5	6	7	w	5	6	7	w	5	6	7	w	5	6	7
aaa	55	148	412	*caa*	4	6	9	*gaa*	28	60	136	*taa*	4	6	9
aac	21	64	172	*cac*	4	6	9	*gac*	26	57	129	*tac*	4	6	9
aag	29	70	172	*cag*	4	6	9	*gag*	27	59	134	*tag*	4	6	9
aat	26	60	139	*cat*	3	5	8	*gat*	17	38	87	*tat*	0	6	0
aca	26	59	136	*cca*	4	7	12	*gca*	17	37	83	*tca*	2	3	5
acc	15	34	81	*ccc*	3	1	4	*gcc*	17	37	83	*tcc*	2	3	5
acg	9	18	39	*ccg*	26	58	127	*gcg*	17	37	83	*tcg*	1	1	1
act	7	13	24	*cct*	26	58	127	*gct*	17	37	83	*tct*	1	1	1
aga	7	13	23	*cga*	26	58	127	*gga*	17	37	83	*tga*	1	1	1
agc	7	13	23	*cgc*	17	37	83	*ggc*	17	37	83	*tgc*	0	1	2
agg	4	6	9	*cgg*	17	37	83	*ggg*	17	37	83	*tgg*	0	1	3
agt	4	6	9	*cgt*	26	58	127	*ggt*	14	28	59	*tgt*	15	34	81
ata	4	6	9	*cta*	26	58	127	*gta*	8	14	24	*tta*	29	59	136
atc	4	6	9	*ctc*	27	61	136	*gtc*	8	13	21	*ttc*	26	60	139
atg	4	6	9	*ctg*	27	59	134	*gtg*	8	13	21	*ttg*	29	70	172
att	4	6	9	*ctt*	26	57	129	*gtt*	4	6	9	*ttt*	21	64	172

Table 7.4: The sizes of Bs of length 5, 6, and 7 generated with different starting words w, where k =3 and $d = 0$.

	Sizes \ Lengths	3	4	5	6	7	8	9	10	11	12
k=2	$\|B\|$	9	20	45	101	227	510	1146	2575	5786	13001
d=0	$\|S^{\otimes}(l)\|$	14	31	70	157	353	793	1782	4004	8997	20216
k=3	$\|B\|$	8	21	55	148	412	1125	3057	8346	22820	62285
d=0	$\|S^{\otimes}(l)\|$	32	80	232	640	1706	4654	12786	34869	95022	259573
k=3	$\|B\|$	3	6	10	18	32	57	101	180	320	569
d=1	$\|S^{\otimes}(l)\|$	12	21	41	71	124	224	398	705	1256	2235
k=4	$\|B\|$		9	26	88	225	577	1563	4369	11988	32854
d=0	$\|S^{\otimes}(l)\|$		120	270	695	2032	5692	15588	41388	112205	306605
k=4	$\|B\|$		12	23	50	101	213	438	914	1891	3932
d=1	$\|S^{\otimes}(l)\|$		69	159	312	672	1365	2868	5910	12318	25503
k=4	$\|B\|$		3	5	8	12	19	31	49	77	122
d=2	$\|S^{\otimes}(l)\|$		21	30	54	92	140	215	343	554	879
k=5	$\|B\|$			10	30	88	303	778	2095	5374	15060
d=0	$\|S^{\otimes}(l)\|$			512	1200	2920	7588	22794	66504	192132	543506
k=5	$\|B\|$			22	51	118	299	728	1800	4407	10798
d=1	$\|S^{\otimes}(l)\|$			276	669	1662	4169	10271	24967	61026	149807
k=5	$\|B\|$			14	26	49	93	175	331	625	1180
d=2	$\|S^{\otimes}(l)\|$			156	309	588	1101	2100	3945	7455	14088
k=5	$\|B\|$			3	5	7	10	14	21	32	48
d=3	$\|S^{\otimes}(l)\|$			39	48	75	135	209	293	407	590

Table 7.5: The sizes of Bs and the density of $S^{\otimes}$ of different lengths generated with the parameters k, d, and w, where w only consists of as.

Parameters\ Lengths	3	4	5	6	7	8	9	10	11	12
k=2, d=0	7	8	30	83	133	384	603	1790	4691	8406
k=3, d=0	6	9	36	116	226	798	1486	5385	17356	37210
k=3, d=1	2	1	3	9	4	14	5	20	64	27
k=4, d=0		3	17	70	121	403	766	2854	9220	19882
k=4, d=1		1	9	23	20	66	44	176	536	448
k=4, d=2		0	0	1	0	0	0	0	0	0
k=5, d=0			5	22	47	212	352	1248	3867	8568
k=5, d=1			14	38	66	202	342	1071	3065	5683
k=5, d=2			5	14	7	26	9	42	140	62
k=5, d=3			0	0	0	0	0	0	0	0

Table 7.6: The sizes of Bs of different lengths generated with the parameters k, d, and w, where w only consists of as. In addition, all the words satisfy the GC-ratio constraint, where $r_1 = 40\%$ and $r_2 = 60\%$.

In Table 7.6, we list the sizes of Bs of different lengths produced from different pairs of k and ds and satisfying only the GC-ratio constraint. The lower bound and the upper bound of the GC-ratio are 40% and 60%.

Note that all DNA words in Bs in Table 7.6, 7.7, and 7.8 satisfy the bond-free constraint by default.

Parameters \ Lengths	3	4	5	6	7	8	9	10	11	12
k=2, d=0	9	20	43	95	209	460	1011	2224	4891	10757
k=3, d=0	8	21	54	145	398	1073	2890	7808	21113	57027
k=3, d=1	3	6	9	16	27	46	78	132	225	382
k=4, d=0		9	25	84	212	530	1423	3950	10753	29071
k=4, d=1		9	19	36	76	147	308	612	1247	2509
k=4, d=2		3	4	6	8	12	18	25	35	51
k=5, d=0			9	28	82	284	697	1823	4553	12641
k=5, d=1			21	48	110	274	653	1602	3865	9331
k=5, d=2			13	24	42	77	137	248	447	802
k=5, d=3			2	3	3	4	5	6	8	9

Table 7.7: The sizes of Bs of different lengths generated with the parameters k, d, and w, where w only consists of as. In addition, all the words satisfy the continuity constraint, where $q = 5$.

In Table 7.7, we list the sizes of Bs of different lengths produced from different pairs of k and ds and satisfying only the continuity constraint. The length limit of the continuity constraint is 5.

Parameters \ Lengths	3	4	5	6	7	8	9	10	11	12
k=2, d=0	7	8	30	83	133	376	593	1710	4315	7812
k=3, d=0	6	9	36	116	226	790	1474	5266	16683	35846
k=3, d=1	2	1	3	9	4	14	5	20	64	27
k=4, d=0		3	17	70	121	392	756	2760	8730	18896
k=4, d=1		1	9	23	20	66	44	176	536	448
k=4, d=2		0	0	1	0	0	0	0	0	0
k=5, d=0			5	22	47	207	342	1173	3531	7923
k=5, d=1			14	38	66	202	342	1063	2973	5614
k=5, d=2			5	14	7	26	9	42	140	62
k=5, d=3			0	0	0	0	0	0	0	0

Table 7.8: The sizes of Bs of different lengths generated with the parameters k, d, and w, where w only consists of as. In addition, all the words satisfy both the GC-ratio constraint and the continuity constraint, where $r_1 = 40\%$, $r_2 = 60\%$ and $q = 5$.

In Table 7.8, we list the sizes of Bs of different lengths produced from different pairs of k and ds and satisfying both the GC-ratio constraint and the continuity constraint. The lower bound and the upper bound of the GC-ratio are 40% and 60%. The length limit of the continuity constraint is 5.

In Table 7.9, we can see that the third set of parameters is $k = 4$, $d = 0$, $r_1 = 40\%$, $r_2 = 60\%$, and $q = 5$, which are the same as the parameters in the fourth row in Table 7.8. This is the reason that, when we want 50 DNA words, the system generates

Parameters	Sample codes
$k = 3$, $d = 0$, q =3 $r_1 = 50\%$, r_2= 50% Desired number = 40	aacagc aaccga aacgca aagcca acaagc acacca acacga acagca acagga accaca accaga accgaa acgaca acgcaa agaagc agacca agacga agcaca agcaga agccaa aggaca caacca caacga caagca caagga cacaca cacaga caccaa cacgaa cactca cagaca cagcaa ccaaca ccaaga ccacaa ccagaa cgaaca cgaaga cgacaa caggaa
$k = 4$, $d = 0$, $q = 3$ $r_1 = 50\%$, $r_2 = 50\%$ Desired number = 20	aacacca aacgcaa aacgcca aactcca aagacca aagccaa aaggcaa acaacca acaccaa acagcaa acagcca acatcca accacaa accacca accgcaa acctcaa acctcca acgacaa acgacca acgccaa
$k = 4$, $d = 0$, $q = 5$ $r_1 = 40\%$, $r_2 = 60\%$ Desired number = 50	aaacca aaccaa aaccca aagcaa aagcca aatcca acacaa acacca accaaa acccaa acccca acgaaa acgcaa acgcca actcaa actcca agacaa agcaaa agccaa agccca aggaaa aggcaa agtcaa caacaa cagcca catcaa catcca ccaaaa ccccaa cctaaa cctcaa cctcca (a portion of words in the resulting set)
$k = 4$, $d = 1$, $q = 5$ $r_1 = 40\%$, $r_2 = 60\%$ Desired number = 50	aacacaca aagacaca acaacaca acacaaca acacaaga acacacaa acacagaa acagaaca acagaaga acagacaa acagagaa agaacaca agacaaca agacaaga agacacaa agacagaa agagaaca agagaaga agagacaa caaacaca caacaaca caacaaga caacacaa caacagaa caagaaca caagaaga caagacaa caagagaa cacaaaca cacaaaga (a portion of words in the resulting set)

Table 7.9: Sample codes generated with specified parameters

words of length 6, since the number of the words of length 6 covers the number of the range from 18 to 70. The fourth set of parameters in Table 7.9 is another example, which refers to the fifth row in Table 7.8.

7.5 Experimental results of the direct encoding system

In this section, we produce DNA codewords – because both the data words and the codewords are DNA words, we use the terms DNA datawords and DNA codewords to make a distinction – by using the same parameters as what are used in the last section. In the direct encoding method, if we consider only the bond-free constraint, we actually encode each symbol in the DNA datawords. The number of DNA datawords depends on the parameters that the users want the DNA codewords to satisfy. We have already presented this fact in Section 7.3. Therefore, given a fixed pair of k (the length of codewords) and d (the Hamming distance), the number of DNA datawords (say $|N|$, where $N = \Sigma^n$) can be calculated by using the methods in Section 7.3. However, if we want to encode more DNA datawords into DNA codewords produced with the same k and d, we can use DNA codewords in N^i, where $i \geq 2$. Similarly, when considering additional constraints, we still can use DNA codewords in N^i, where $i \geq 2$ and $N \subset \Sigma^n$.

In the following tables, we obtain the sizes of some DNA codeword sets satisfying different constraints, and using values of parameters.

$k \backslash d$	0	1	2	3	4	5	6	7
4	4^2	4^2	4^2					
5	4^3	4^3	4^3					
6	4^4	4^4	4^3	4^3				
7	4^5	4^4	4^4	4^4				
8	4^5	4^5	4^5	4^5	4^4			
9	4^6	4^6	4^6	4^5	4^5			
10	4^7	4^7	4^6	4^6	4^6	4^6		
11	4^8	4^7	4^7	4^7	4^7	4^6		
12	4^8	4^8	4^8	4^8	4^7	4^7	4^7	
13	4^9	4^9	4^9	4^8	4^8	4^8	4^8	
14	4^{10}	4^{10}	4^9	4^9	4^9	4^9	4^8	4^8

Table 7.10: The sizes of DNA codeword sets calculated in response to different pairs of k and ds. The DNA codewords only satisfy the bond-free constraint.

In Table 7.10, we list the sizes of DNA codeword sets produced from different pairs of k and ds and only satisfying the bond-free constraint.

$k \backslash d$	0	1	2	3	4	5	6	7
4	6	6	6					
5	40	40	40					
6	200	200	50	50				
7	560	140	140	140				
8	728	728	728	728	182			
9	2016	2016	2016	504	504			
10	10752	10752	2688	2688	2688	2688		
11	50688	12672	12672	12672	12672	3168		
12	40128	40128	40128	40128	10032	10032	10032	
13	192192	192192	192192	48048	48048	48048	48048	
14	604032	604032	151008	151008	151008	151008	37752	37752

Table 7.11: The sizes of DNA codeword sets calculated in response to different pairs of k and ds. The DNA codewords satisfy the bond-free constraint and the GC-ratio constraint, where $r_1 = 40\%$ and $r_2 = 60\%$.

In Table 7.11, we list the sizes of DNA codeword sets produced from different pairs of k and ds and satisfying the bond-free constraint and the GC-ratio constraint, where $r_1 = 40\%$ and $r_2 = 60\%$.

$k \backslash d$	0	1	2	3	4	5	6	7
4	16	16	16					
5	62	62	62					
6	246	246	58	58				
7	980	228	228	228				
8	904	904	904	904	216			
9	3600	3600	3600	848	848			
10	14370	14370	3362	3362	3362	3362		
11	57306	13390	13390	13390	13390	3150		
12	53448	53448	53448	53448	12488	12488	12488	
13	213144	213144	213144	49736	49736	49736	49736	
14	850032	850032	198528	198528	198528	198528	46400	46400

Table 7.12: The sizes of DNA codeword sets calculated in response to different pairs of k and ds. The DNA codewords satisfy the bond-free constraint and the continuity constraint, where $q = 5$.

In Table 7.12, we list the sizes of DNA codeword sets produced from different pairs of k and ds and satisfying the bond-free constraint and the continuity constraint, where $q = 5$.

$k \backslash d$	0	1	2	3	4	5	6	7
4	6	6	6					
5	40	40	40					
6	200	200	50	50				
7	560	140	140	140				
8	706	706	706	706	174			
9	1970	1970	1970	486	486			
10	10252	10252	2492	2492	2492	2492		
11	47238	11340	11340	11340	11340	2740		
12	36486	36486	36486	36486	8846	8846	8846	
13	170094	170094	170094	40800	40800	40800	40800	
14	540672	540672	130356	130356	130356	130356	31524	31524

Table 7.13: The sizes of DNA codeword sets calculated in response to different pairs of k and ds. The DNA codewords satisfy the bond-free constraint, and GC-ratio constraint, and the continuity constraint, where $r_1 = 40\%$, $r_2 = 60\%$, and $q = 5$.

In Table 7.13, we list the sizes of DNA codeword sets produced from different pairs of k and ds and satisfying the bond-free constraint, the GC-ratio constraint, and the continuity constraint, where $r_1 = 40\%$, $r_2 = 60\%$, and $q = 5$.

7.6 Comparison between the results of the two systems

We have two implementations that produce words for desired constraints. The advantage of the subword closure operation encoding system is that the method is general enough for arbitrary local constraints. Users can provide a set S in which the words are of a fixed length and preserve a certain property. The method applied in the subword closure operation encoding system are able to produce words in $S^{\otimes}$. As a result, the property will be kept within words whose lengths are longer than the word length in S. This system can also be used for producing DNA words satisfying the DNA computing related constraints. The drawback of this system is the efficiency. However, it is possible to improve the efficiency by improving the methods for producing EN. We discuss this in the next chapter as a possible future work of this research. The direct encoding system can only produce DNA codewords satisfying the DNA computing related constraints, but the advantage of this system is that it can perform encoding and decoding in linear time. A drawback of this method is that, when considering either the GC-ratio constraint or the continuity constraint or both, we can not encode arbitrary data. Here, the arbitrary data really means all the DNA datawords of a certain length.

From the results in the previous two sections, we can see that the direct encoding system produces better results when producing DNA words satisfying the DNA computing related constraints. For example, in Table 7.6 and Table 7.11, when $k = 4$ and

$d = 0$ and the length of the DNA codewords is 4, the direct encoding system allows us to encode 6 datawords as opposed to encoding 3 datawords that is allowed by the subword closure operation encoding system. However, if we want to encode more than 6 datawords into a language satisfying the same constraints with the same parameters, the subword closure operation encoding system might provide a better information ratio. For example, if we want to encode 10 datawords into a language that satisfies the bond-free constraint with $k = 4$ and $d = 0$ and the GC-ratio constraint with $r_1 = 40\%$ and $r_2 = 60\%$, we still have to look at Table 7.6 and Table 7.11. This time, the subword closure operation encoding system allows us to use DNA codewords of length 5 to encode the 10 datawords, but we have to use DNA codewords of length 8 to encode these 10 datawords, since $|N| < 10 < |N^2|$, where N contains the 6 datawords shown in Table 7.11.

In another case, in Table 7.6 and Table 7.11, if we use the same constraints with the same parameters, the subword closure operation encoding system allows us to encode up to 403 datawords when the codeword length is 8; the direct encoding method allows us to encode up to 36 words when the codeword length is 8, since $|N^2| = 36$. It seems that, in this case, the subword closure operation encoding system provides a better result. However, by using Theorem 4 in Section 3.6.4, we know that $(\tau, H_{0,4})$-bond-free is the same as $(\tau, H_{1,8})$-bond-free. Therefore, if we look at Table 7.11, we can encode up to 728 datawords into the language satisfying the same constraints with the same parameters.

In all, the two systems provide us many ways to produce DNA codewords satisfying

desirable constraints. That which method can provide a better result really depends on what kind of codeword set the users want to produce.

7.7 Discussion

In the subword closure operation encoding system, we apply the two guidelines in Chapter 6 and the recursive formula in Chapter 4 to choose an EN and apply the B-block method to produce DNA words. The two guidelines are non-deterministic and might not always work. The reason we still use them is that, if they work, this method produces more codewords than or at least equal to the result produced by Lemma 3. We say this because the method for checking if the two guidelines worked is to check if there is at least one cycle-intersection node in the deterministic automaton produced from the EN and the corresponding BE produced by the two guidelines. The worst case that the two guidelines succeeded is that there is only one cycle-intersection node. Therefore, the result obtained by using the two guidelines will be better or at least equal to the result produced by Lemma 3, since there may be other cycles that are involved to produce DNA words.

Actually, the two guidelines worked very well when we tested the system by using many different sets of parameters. We have used 336 different sets of parameters: all the DNA words of length 2 as the starting word and $d = 0$ (16 sets of parameters), all the DNA words of length 3 as the starting word and $d = 0$ (64 sets of parameters), and all the DNA words of length 4 as the starting word and $d = 0$ (256 sets of parameters). The $T^{\otimes}$s produced for accepting $S^{\otimes}$s do not contain any cycle-intersection node when

Ss are produced with the following parameters: $k = 2$, $d = 0$, and $w \in \{ag, at, ca, ta, tc, tg\}$. In the remaining 330 cases, the $T^{\otimes}$s contain cycle-intersection nodes and the automata produced from ENs and the corresponding BEs produced by the two guidelines also contain cycle-intersection nodes.

Chapter 8

Conclusion and future work

8.1 Conclusion and discussion

In this thesis, we introduced a method for encoding arbitrary data into DNA languages that satisfy important constraints in DNA computing. Moreover, this method can be applied to encode arbitrary data into languages produced by the subword operation applied on a word set satisfying any arbitrary local constraint.

Along with the research, we investigated properties of the subword closure operation and proposed the concept of B-block to address the problem that not any concatenation of two words in $S^{\otimes}$ is in the same $S^{\otimes}$. Moreover, we introduced a recursive formula to calculate the density of languages produced by the subword closure operation.

To answer the question of whether we can produce an arbitrarily large B of a fixed length or not, in Chapter 5, we investigated the properties of cycles in automata and

introduced methods for checking if an automaton $T^{\otimes}$ can produce an arbitrarily large B, where $T^{\otimes}$ is the automaton for accepting $S^{\otimes}$.

We apply the subword closure operation to construct languages from word sets satisfying desirable local constraints. In Chapter 6, we proposed some construction methods for producing word sets that satisfy the desirable local constraints. We should notice that we convert the GC-ratio constraint from a global constraint to a local constraint, since any concatenation of any DNA words that are of a fixed length and satisfy a GC-ratio constraint satisfies the same GC-ratio constraint in which the two parameters r_1 and r_2 remain the same. We proved this in Section 6.2.1. After obtaining a set S representing an individual local constraint, we can produce a language $S^{\otimes}$. Also, we can produce a B of a fixed length that is a subset of $S^{\otimes}$. However, sometimes, we need a B in which all the words satisfy more than one local constraint. Theoretically, we can construct several languages each of which satisfies one desired local constraint, for example $S_1^{\otimes} \cap S_2^{\otimes} \cap \cdots \cap S_n^{\otimes}$, where each S_i represents one desired local constraint. However, this might not be practical, since every time we intersect two languages, the number of states of the automaton for accepting the intersection language is the product of the numbers of the states of the two automata that accept the two languages. In this thesis, automata used for accepting $S^{\otimes}$s can be large, and automata for accepting intersection languages can be even larger. Therefore, we used a trade-off method to make sure that the produced languages satisfy the bond-free constraint and the system can work efficiently, since the bond-free constraint is the major concern in DNA computing. The definition

of B-block ensures that $B^+ \subseteq S^{\otimes}$, so we first produce a B satisfying the bond-free constraint, and then, if additional constraints are required by users, pass all the words in B through additional filters. As a result, the remaining words will satisfy the desired constraints. However, if the continuity constraint is required by the users and we concatenate two words among the remaining words, not every concatenation will satisfy the same continuity constraint.

In Chapter 7, we implemented a system to produce DNA languages satisfying desirable constraints for encoding arbitrary data. Some experimental results were obtained in response to values of parameters. Also, we briefly compared some results obtained from the two subsystems.

8.2 Future work

As we are gaining more understanding about the properties of cycles in automata, we realize that there is a lot more that can be done to improve the methods proposed in this thesis. In this section, we present a few ideas for future research.

One aspect that we can improve is the algorithm for choosing EN in Section 7.2. As we see in Example 10, the algorithm introduced in that section does not directly follow the two guidelines proposed in Chapter 6 and can not always provide the best choice of EN. In the further discussion in the same example, we see that, even if we follow the two guidelines, we still might not be able to obtain the best choice of EN. The reason we use this algorithm in the system is that it works efficiently and provide good choices of EN in most of the times. On the other hand, we believe there is more

that can be done to improve both the accuracy and the efficiency of this algorithm.

In this thesis, to be able to produce larger B from the same S, we use the two guidelines in Chapter 6 and the recursive formula in Chapter 4 to evaluate some possible ENs. However, this method might not always give us the best choice. As we see in Chapter 5, the real property that allows us to produce longer words and arbitrarily large Bs is the communicating cycles in deterministic automata. Currently, we have the concept of 2-way communicating cycles and cycle-intersection nodes to make sure that can we produce arbitrarily large Bs. But this is not enough. So another research direction is to investigate the properties of n-way communicating cycles and n-cycle-intersection nodes that allow us to improve the information ratio of Bs and improve the efficiency of the system. It is possible that the higher the n is, the larger B we can produce, which is produced from the n cycles starting from the n-cycle-intersection node. If we are able to evaluate the complexity of cycles in deterministic automata, we do not need the two guidelines any more, because the evaluation will be deterministic and we will be able to know which EN is the best choice.

As stated in the previous section, we use a trade-off method to produce DNA languages. We not only make sure that the resulting languages satisfy the bond-free constraint, the major constraint in DNA computing, but also make sure the system works efficiently. However, the continuity constraint sometimes will be violated in the concatenations of the DNA words in B that passed through the continuity constraint filter. This is a problem we can address in the future.

When we test the effectiveness of the two guidelines, we used 336 different sets of parameters. We did not find a case such that a $T^{\otimes}$ contains at least one cycle-intersection node and the two guidelines fail to find a proper EN. To cover a wide range of situations, it is necessary to test more data sets in the future research.

Bibliography

[1] L. M. Adleman, "Molecular Computation of Solutions to Combinatorial Problem" *Science* Vol. **226** (Nov. 1994), pp. 1021-1024.

[2] L. M. Adleman, "Computing with DNA" *Scientific American* **279(2)** (August 1998), pp.54–61.

[3] M. Amos, G. Păun, G. Rozenberg, A. Salomaa, "Topics in the theory of DNA computing" *Theoretical Computer Science* **287** (2002), pp.3-38.

[4] M. Arita, S. Kobayashi, "DNA Sequence Design Using Templates" *New Generation Computing* **20** (2002), pp. 263-277.

[5] D. Boneh, C. Dunworth, R. Lipton, J. Sgall, "On the Computational Power of DNA" *Technical Report TR-499-95, Princeton University, USA*, October 1995.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein *Introduction to Algorithms*, MIT Press and McGraw-Hill, Second Edition, 2001.

[7] B. Cui, S. Konstantinidis, "DNA Coding Using the Subword Closure Operation" *The 13th International Meeting on DNA Computing, Memphis, Tennessee, USA, June 4-8, 2007* .

[8] J. H. M. Dassen, "DNA Computing: Promises, Problems and Perspective" *IEEE Potentials* Vol.**16** (1997-1998), pp. 27-28.

[9] R. Deaton, M. Garzon, J. Rose, D.R. Franceschetti, S.E. Stevens, Jr., "DNA Computing: A Review" *Fundamenta Informaticae* Vol.**30** (1997), pp.23-41.

[10] A. A. El Gamal, L. A. Hemachandra, I. Shperling, V. K. Wei,"Using Simulated Annealing to Design Good Codes" *IEEE Transactions on Information Theory* Vol. **IT-33**, No. **1**(January 1987), pp. 116-123.

[11] K. Erk, "Simulating Boolean Circuits by Finite Splicing" *Proc. of Congress on Evolutionary Computation, 1999*, pp.1279-1285.

[12] A. Fujiwara, S. Kamio, "Procedures for Multiple Input Functions with DNA Strands" *Proc. of the 18th International Parallel and Distributed Processing Symposium, 26-30 April 2004*, pp.173-181.

[13] A. Fujiwara, K.Matsumoto, W. Chen, "Addressable Procedures for Logic and Arithmetic Operations with DNA Strands" *Proc. of the International Parallel and Distributed Processing Symposium, 2003*, pp. 162-170.

[14] G. Gloor, L. Kari, M. Gaasenbeek, S. Yu, "Towards a DNA Solution to the Shortest Common Superstring Problem" *Proc. of 1998 IEEE International Joint*

Symposia on Intelligence and Systems, Rockville, Maryland, May 1998, pp. 111-116.

[15] O. N. Granichin, S. S. Sysoev, "About Some Characteristics of Computers of New Generation" *Proc. of the International Conference on Physics and Control, St. Petersburg, Russia, 2003*, Vol. **3**, pp. 804-807.

[16] J. Hartmanis, "On the Weight of Computation" *Bullettin of the European Association for Theoretical Computer Science* Vol. **55** (Feb 1995), pp. 136-138.

[17] H. H. Hoos, T. Stützle, "Evaluating Las Vegas Algorithms - Pitfalls and Remedies" *Proc. of the 4th Conference on Uncertainty in Artificial Intelligence UAI-98, 1998*, pp. 238-245.

[18] J. E. Hopcroft, R. Motwani, J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison Wesley, Second Edition, Toronto, Canada, 2001.

[19] S. Hussini, L. Kari, S. Konstantinidis, "Coding Properties of DNA Languages" *Theoretical Computer Science* **290** (2003) pp. 1557-1579.

[20] Z. Ibrahim, Y. Tsuboi, O. Ono, M. Khalid, "Molecular Computation Approach to Compete Dijkstra's Algorithm" *The 5th Asian Control Conference, 20-23 June 2004*, Vol.**1**, pp. 635-642.

[21] N. Jonoska, D. Kephart, K. Mahalingam, "Generating DNA Code Words" *Congressus Numerantium* Vol. **156** (2002), pp. 99-110.

[22] N. Jonoska, K. Mahalingam, "Languages of DNA Based Code Words" *Pre-proc. of DNA, Madison, Wisconsin, 9 June 2003*, pp.58-68.

[23] N. Jonoska, K. Mahalingam, J. Chen, "Involution Codes: With Application to DNA Coded Languages" *Natural Computing: An International Journal* Vol. **4**, No. **2** (June 2005), pp. 141-162.

[24] H. Jürgensen, S. Konstantinidis, N. H. Lâm "Asymptotically Optimal Low-Cost Solid Codes" *Technical Report No. 2002-7, Saint Mary's University, Halifax, NS, Canada*,April, 2002.

[25] L. Kari, R. Kitto, G. Thierrin, "Codes, Involutions and DNA encoding" W. Bauer, H. Ehrig, J. Karhumäki, A. Salomaa (Eds.), *Formal and Natural Computing* LNCS 2300 (2002) pp. 376-339.

[26] L. Kari, S. Konstantinidis, "Static and Dynamic Properties of DNA Languages" *Proc. of the 25th Annual International Conference of the IEEE EMBS, Cancun, Mexico, September 17-21, 2003*, pp. 3846-3849.

[27] L. Kari, S. Konstantinidis, E. Losseva, G. Wozniak, "Sticky-free and Overhang-free DNA Languages" *Acta Informatica* **40** (2003), pp. 119-157.

[28] L. Kari, S. Konstantinidis, P. Sosík, "Bond-Free Languages: Formalizations, Maximality, and Construction Methods" *International Journal of Foundations of Computer Science* Vol.**16** (2005), pp.1039-1070.

[29] L. Kari, S. Konstantinidis, P. Sosík, "On Properties of Bond-free DNA Languages" *Theoretical Computer Science* Vol.**334** (2005), pp.131-159.

[30] L. Kari, S. Konstantinidis, P. Sosík, G. Thierrin, "On Hairpin-free Words and Languages" *Proc. of the 9th International Conference on Developments in Language Ttheory, Palermo, Italy, 4-8 July 2005*, LNCS 3572, pp.296-307.

[31] D. Kephart, J. Lefevre, "Codegen: The Generation and Testing of DNA Code Words" *Proc. of the 2004 IEEE Congress on Evolutionary Computation, Honolulu, Hawaii, 2002*, pp. 1865-1873.

[32] K. Kiguchi, K. Watanabe, T. Fukuda, "Trajectory Planning of Mobile Robots Using DNA Computing" *Proc. of 2001 IEEE International Symposium on Computational Intelligence in Robotics and Automation, Banff, Alberta, Canada, 29 July-1 August, 2001*, pp.380-385.

[33] H. R. Lewis, C. H. Papadimitriou, *Elements of the Theory of Computation*, Prentice-Hall, Second Edition, Upper Saddle River, New Jersey, August 7, 1997.

[34] F. J. MacWilliams, N. J. A. Sloane, *The Theory of Error-Correcting Codes*, North-Holland, 1978.

[35] A. Marathe, A. E. Condon, R. M. Corn, "On Combinatorial DNA Word Design" *Journal Computational Biology* Vol.**8** (2001), pp. 201-220.

[36] D. V. Noort, F. U. Gast, J. S. McCaskill, "DNA Computing in Microreactors" *The 7th International Workshop on DNA-based Computer, Tampa, Florida, 2001*, LNCS 2340, pp.33-46.

[37] M. Ogihara, A. Ray, "Executing Parallel Logical Operations with DNA" *Proc. of IEEE Congress On Evolutionary Computation, Piscataway, NJ, 1999*, pp. 972-979.

[38] J. H. Reif, T. H. LaBean, M. Pirrung, V. S. Rana, B. Guo, C. Kingsford, G. S. Wichham, "Experimental Construction of Very Large Scale DNA Databases with Associative Search Capability". *Proc. 7th Workshop on DNA-Based Computers, Tampa, Florida, 2002.* LNCS 2340, pp.231-247.

[39] F. Tanaka, M. Nakatsugawa, M. Yamamoto, T. Shiba, A.Ohuchi, "Developing Support System for Sequence Design in DNA Computing" *Proc. 7th Workshop on DNA-Based Computers, Tampa, Florida, 2002.* LNCS 2340, pp.129-137.

[40] J. M. Tour, W. L. V. Zandt, C. P. Husband, S. M. Husband, L. S. Wilson, P. D. Franzon, D. P. Nackashi, "Nanocell Logic Gates for Molecular Computing" *IEEE Transactions on Nanotechnology* Vol. **1**, No. **2** (June 2002), pp. 100-109.

[41] Y. Tsuboi, Z. Ibrahim, O. Ono, "Problem solving Method with Semantic Net Based on DNA Computing in Artificial Intelligence" *The 5th Asian Control Conference, Melbourne, Australia, 20-23 July, 2004*, pp. 652-657.

[42] Y. Tsuboi, O. Ono, "Pattern Matching Algorithm for Engineering Problems by Using DNA Computing" *Proc. of the 2003 IEEE/ASME International Con-*

ference on Advanced Intelligent Mechatronics, Kobe, Japan, 20-24 July 2003, pp.1005-1008.

[43] D. Tulpan, H. Hoos, A. E. Condon, "Stochastic Local Dearch Algorithms for DNA Word Design" *Proc. 8th Workshop on DNA-based Computers, Saeporo, Japan, 2002.* LNCS 2568.2002.

[44] A. M. Turing, "On Computable Numbers, with an Application to the Entcheidungproblem" *Proc. of the London Mathematical Society*, 1936, II Ser.42, pp. 230-265.

[45] J, Xu, "Formalizations of Error Models With Applications to Spelling Error Correction", *MASc Thesis, Saint Mary's University, Halifax, NS, Canada*, 2004.

[46] P. Wasiewicz, J. J. Mulawka, W. R. Rudnicki, B. Lesyng, "Adding Numbers with DNA" *2000 IEEE International Conference on System, Man, and Cybernetics, 2000*, Vol.1, pp.265-270.

[47] M. Watson, "Practical Artificial Intelligence Programming in Java", www.markwatson.com.