Grammars and a Random Generator for Deterministic Chain Regular Expressions

Xinyu Chu\textsuperscript{1,2}, Ping Lu\textsuperscript{3}, and Haiming Chen\textsuperscript{1}\textsuperscript{*}

\textsuperscript{1} State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China
\textsuperscript{2} University of Chinese Academy of Sciences
\{chuxy, chm\}@ios.ac.cn
\textsuperscript{3} Beijing Advanced Innovation Center for Big Data and Brain Computing, Beihang University, Beijing 100191, China
luping@buaa.edu.cn

Abstract. Deterministic regular expressions (DREs) are a core part of XML Schema and widely used in other applications. There are context-free grammars for DREs, which are not efficient for being used and also lead to the problem of being unable to generate expressions of the given length in an efficient manner. Deterministic chain regular expressions (dCHAREs) are a very practical subclass of DREs. In many practical situations, dCHAREs are more suitable than standard or deterministic regular expressions. But dCHAREs do not have a simple syntax, which puts a burden on the applications of these expressions. In this paper, we propose derivation rules and give regular grammars for dCHAREs which are more succinct than the grammars of DREs. Based on the grammars, we further design an algorithm to randomly generate dCHAREs of the given length, which fills the gap in the generation of DREs. Experimental results demonstrate that our generator is efficient in terms of running time and complexity.

Keywords: Deterministic Chain Regular Expressions · Regular Grammars · Efficient Random Generation Algorithm

1 Introduction

1.1 Motivation

Deterministic regular expressions (DREs) and its subclasses are a core part of XML Schema \cite{33, 34} and widely used in other applications (e.g., \cite{19, 23}). They have been extensively studied in the literature \cite{3, 7, 8, 10–12, 15, 17, 21, 24, 27–29, 31}, also under the name of one-unambiguous regular expressions. But they used to be a mystery for users because they were defined only by a semantic manner

\textsuperscript{*} Corresponding author. Work supported by the National Natural Science Foundation of China under Grant Nos. 61872339 and Nos. 61472405.

Copyright \textcopyright{} 2020 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).
for a long time. To solve this problem, Xu et al. [35] proposed a syntax for DREs, which made DREs better understood and used more widely. But the grammars describing the syntax of DREs are context-free grammars, which is not efficient when the alphabet is large or the expression length is long and there remains a problem on random generating DREs by their grammars, i.e., the generation algorithm of [35] only guarantees that the length of a generated expression is not longer than the given length, e.g. when you want to generate a DRE of length 10, the expression it gives may be 9 or 8 in length. So it is necessary to put forward more concise grammars, which can generate expressions of the given length.

Practical research shows that 79.54% of DREs in practice are deterministic chain regular expressions (dCHAREs), which is one of the most practical sub-classes of DREs [5, 22]. In some specific applications, dCHAREs can be more suitable than DREs. But they are still a mystery to users, because they were defined only by a semantic manner, which put a burden on the applications of these expressions. Hence, it is necessary to study grammars for dCHAREs and their applications. By considering dCHAREs, it is possible to have simpler grammars, and give random generation algorithms which can generate expressions with the given length, a challenge we consider in this paper.

1.2 Contributions
• We propose derivation rules and regular grammars for dCHAREs.
• Based on the grammars of dCHAREs, we design a generation algorithm, which can randomly generate expressions of the given length.
• Compared to generation algorithms in [35], our generator performs better with longer given lengths and larger alphabet sizes.
• We experimentally evaluate that our generation algorithm is efficient in terms of running time and complexity.

1.3 Overview
We concentrate our attention on dCHAREs, which is proved to be a widely used subclass of DREs in practice. In Section 3, we introduce the necessary definitions. In Section 4, we discuss the structure properties of dCHAREs, based on which we propose a derivation system and regular grammars for dCHAREs. Then we design an algorithm to generate dCHAREs based on the grammars, which can generate expressions exactly with the given length in Section 5. The analysis and comparison experiments of our generation algorithm are in Section 6. We summarized this paper in Section 7.

2 Related Work
Deterministic regular expressions and its subclasses. There has been a lot of work to decide the determinism of regular expressions and its subclasses, such as for standard regular expressions [7], for expressions with counting [11, 17, 20], or for expressions with interleaving [27]. There has been a lot of work on inferring DREs, Bex et al. gave algorithms to learn deterministic $k$-occurrence REs based on the Hidden Markov Model [4], Freydenberger and Kötzing [14] gave linear time algorithms to infer subclasses of DREs. To study the practicability of DREs and its subclasses, Li et al. [22] harvested a large-scale real data from
the Web, which indicates that dCHAREs is one of the most practical subclass of
DREs. Another problem is random generating expressions, on which we focused
in this paper.

Random generation. Randomly generating expressions find applications in
hardware and software testing, coding theory, bioinformatics. Hanford et al. [32]
and Denise et al. [13] separately proposed algorithms for generating sentences
randomly from context-free languages. Arnold and Sleep [1] considered the uni-
formity, which means that all expressions of the given length are generated by
the grammars equally likely, and presented an algorithm to generate balanced
parenthesis strings. Hickey et al. [18] also proposed methods based on parse tree
to uniformly generate strings in a context-free language. Bernardi et al. proposed
the first linear algorithm for random sampling from a regular language by a de-
terministic finite automaton [2]. Researches proposed several random generation
algorithms to improve either the time and space bounds [16, 25] or the prepro-
cessing [26]. Enumeration algorithms can be used as random generators [36], but
due to the large memory required by the maintenance of the whole grammar and
the information required by the generation, they are restricted when used for
random generation. Xu et al. [35] proposed a generation algorithm based on the
grammars of DREs, however, there are still problems unsolved in their generator
due to the production form and the large scale of grammars of DREs.

We consider to solve the random generation problem by a bottom-up algo-
rithm based on grammars of dCHAREs, by taking full advantage of the chain
structure and the information that the grammars take, our algorithm needs no
preprocessing and can fill a gap in [35], i.e., randomly generate dCHAREs exactly
equal to the given length and more efficient in the case of long given lengths
and large alphabet sizes.

3 Definitions
Let \( \Sigma \) be an alphabet of symbols. The set of finite words over \( \Sigma \) is denoted
by \( \Sigma^* \). Note: \( \varepsilon \) represents the empty word, \( \emptyset \) represents the empty set. For an
expression \( r \) over \( \Sigma \), the language specified by \( r \) is denoted by \( L(r) \).

**Definition 1. Regular Expressions (REs).** A RE over \( \Sigma \) is \( \varepsilon \) or \( a \in \Sigma \), or
the union \( r_1 \| r_2 \), the concatenation \( r_1 \cdot r_2 \), the plus \( r_1^+ \), the kleene star \( r_1^* \) or
the question mark \( r_1^? \) for REs \( r_1 \) and \( r_2 \).

**Definition 2. Simple Regular Expressions [5].** A base symbol is a RE \( a \),
\( a^+ \), or \( a^* \) where \( a \in \Sigma \), a factor is of the form \( e \), \( e^+ \), or \( e^* \) where \( e \) is a disjunction
of base symbols. A simple regular expression is \( \varepsilon \), \( \emptyset \) or a sequence of factors.

To define deterministic regular expressions, we need some notations. For a reg-
ular expression we can mark symbols with subscripts such that in the marked
expression each marked symbol occurs only once. Without loss of generality,
we use the positions as the marked subscripts. For example, a marking of the
expression \((a|b)^+ab(a|b)\) is \((a_1|b_2)^+a_3b_1(a_5|b_6)\). The marking of an expression \( r \)
is denoted by \( r^\# \). Accordingly, the result of dropping off the subscripts from a
marked expression \( r \) is denoted by \( r^\Psi \). Then we have \((r^\#)^\Psi = r\).
Definition 3. Deterministic Regular Expressions (DREs) [8]. An expression \( r \) is deterministic if it satisfies the condition: for any two words \( u \) and \( v \), \( u \langle r \rangle v \) is in \( L(r) \) if and only if \( u \langle r \rangle v \) is in \( L(r) \). A regular language is deterministic if it can be denoted by some deterministic expression.

Definition 4. Chain Regular Expressions (CHAREs). A CHARE is a RE (Def. 1) of the form \( f_1 \cdots f_n \), where every \( f_k \) (\( 1 \leq k \leq n \)) is a factor of the form \( e \), \( e^* \), or \( e^+ \). A base symbol is a RE \( a \), \( a^* \), \( a^+ \) where \( a \in \Sigma \), and factor \( e \) is a disjunction of base symbols with same unary operators.

E.g., \( (c^*)^+ \cdot (a^+|b^+)^* \cdot (a^*|b^*|c^*)^+ \) is a CHARE while \( (ab|c)^* \) and \( (a|b)^+ \cdot (c^+|d^+) \) are not CHAREs because their structure does not conform to the definition.

Definition 5. Deterministic Chain Regular Expressions (dCHAREs).

A dCHARE is a CHARE (Def. 4) which is deterministic.

E.g., \( (a^*|b^+)^* \cdot (c^+|d^+) \) is a dCHARE, \( (a^*|b^+)^* \cdot (a^*|b^*|c^*)^+ \) is not a dCHARE because it is not deterministic. Let \( S = \{ S, \alpha, \beta \} = \{ \alpha_1, \cdots, \alpha_n \} \), \( 1 \leq i \leq n \), \( a_i \in \Sigma \), \( a_i \neq a_j \) for \( i \neq j \), \( n \geq 1 \), \( \alpha, \beta \in \{ , \cdot, ?, * \} \). Denote \( r^\alpha = r \).

Note that dCHAREs we defined here are different from the chain regular expressions [6] which is a subclass of SOREs [6] in which each alphabet symbol can occur at most once. The dCHAREs we defined support multiple occurrences of alphabet symbols.

To construct the grammars, we define the following sets and the function \( \lambda \):

\[
\text{First}(r) = \{ a | \omega \in L(r), a \in \Sigma, \omega \in \Sigma^* \}
\]

\[
\text{followLast}(r) = \{ b | \nu \in L(r), \nu \neq \varepsilon, \nu \omega \in L(r), b \in \Sigma, \omega \in \Sigma^* \}
\]

\[
\lambda(r) = \text{true}, \text{ if } \varepsilon \in L(r); \lambda(r) = \text{false}, \text{ otherwise}
\]

The First set and function \( \lambda \) for any expression can be computed as follows:

First(\( r \)) = \{ \}
First(\( a \)) = \{ a \}, a \in \Sigma
First(s) = First(s) \cup First(s)
First(\( r \cdot s \)) = \begin{cases} First(\( r \)) \cup First(\( s \)) & \text{if } \lambda(\( r \)) = \text{true} \\ First(\( r \)) & \text{otherwise} \end{cases}
First(\( r^+ \)) = First(\( r^* \)) \cup First(\( r \))
\lambda(\varepsilon) = \text{true}; \lambda(a) = \text{false}, a \in \Sigma
\lambda(\( r \)) = \lambda(\( r \)) \lor \lambda(\( s \)); \lambda(\( r \cdot s \)) = \lambda(\( r \)) \land \lambda(\( s \))
\lambda(\( r^+ \)) = \lambda(\( r^* \)) = \lambda(\( r \))

For REs, the followLast set can be computed as follows [11]:
followLast(\( r \)) = followLast(\( a \)) = \{ \}, a \in \Sigma
followLast(\( r \cdot s \)) = followLast(\( r \)) \cup followLast(\( s \))
followLast(\( r^+ \)) = followLast(\( r^* \)) \cup followLast(\( r \))
followLast(\( r^* \)) = followLast(\( r^+ \)) \cup followLast(\( r \))
followLast(\( r^+ \)) = followLast(\( r^* \)) \cup followLast(\( r \))

4 Grammars for dCHAREs

In this section, we first exhibit a derivation system for characterizing and recognizing dCHAREs. Then we give regular grammars for dCHAREs and propose an algorithm to construct the grammars.
4.1 Derivation System for dCHAREs

We give the derivation system by exploiting the structure of dCHAREs. The following lemmas can be easily obtained from Def. 3 and Def. 4.

Lemma 1 ([8, 10, 20]). Let E be a regular expression. E = E₁ · E₂: If L(E) = ∅, then E is deterministic. If L(E) ≠ ∅ and ε ∈ L(E₁), then E is deterministic iff E₁ and E₂ are deterministic, First(E₁) ∩ First(E₂) = ∅, and followLast(E₁) ∩ First(E₂) = ∅. If L(E) ≠ ∅ and ε ∉ L(E₁), then E is deterministic iff E₁ and E₂ are deterministic, and followLast(E₁) ∩ First(E₂) = ∅.

Lemma 2. Let r = (S, α, β). Then First(r) = S. α ∈ {?, *} or β ∈ {?, *} iff λ(r) = true.

Lemma 3. If r = (S₁, α₁, β₁) · · · (Sₙ, αₙ, βₙ) is a dCHARE. When αₙ = ε and βₙ = o, then followLast(r) = ∅. When αₙ = + or βₙ = +, and αₙ, βₙ ∉ {?, *}, then followLast(r) = Sₙ.

Let ⊨ r means the expression r is a dCHARE. A derivation rule is of the form:

\[ ⊨ r₁ · · · rₙ (c₁ · · · cₘ) \]

which means if r₁, · · · , rₙ are dCHAREs, and the conditions c₁, · · · , cₘ hold, then r is a dCHARE. We obtain the derivation system DC as follows:

(Fac) \[ ⊨ (S, α, β) \]

(SeqA) \[ ⊨ r = (S₁, α₁, β₁) · · · (Sₘ, αₘ, βₘ) (m ≥ 1) \quad αₘ, βₘ = o \]

(SeqB) \[ \begin{align*}
αₘ &= + ∨ βₘ = + \land (αₘ, βₘ ∉ {?, *}) \\
Sₘ ∩ S &= ∅ \\
\end{align*} \]

(SeqC) \[ \begin{align*}
t &= i, · · · , m \quad i = 1 ∨ (i > 1, αᵢ−₁ βᵢ−₁ = o) \\
\alphaᵢ ∈ {?, *} ∨ βᵢ ∈ {?, *} \\
Sᵢ ∩ S &= ∅ \\
\end{align*} \]

(SeqD) \[ \begin{align*}
t &= i, · · · , m \quad i > 1 \quad Sᵢ₋₁ ∩ S = ∅ \quad Sᵢ ∩ S = ∅ \\
\end{align*} \]

According to those derivation steps, we can concatenate a factor behind a dCHARE r to make r · (S, α, β) a new dCHARE.

Theorem 1. (Soundness and completeness) An expression r is a dCHARE iff r is derivable from DC.

Proof. According to the chain structure of dCHAREs, lemma 1 guarantees that any length of dCHAREs can be generated in the form of concatenating sub-expressions. The rules in DC cover all candidate value of α and β. We say r is derivable if there is a derivation tree in DC whose root is r. If r is derivable in DC, r is clearly a dCHARE by the lemmas listed above. On the other hand, for a dCHARE r, we can construct a derivation tree, which is isomorphic to the structure of r. Thus r is derivable in DC. In conclusion, the derivation system DC is sound and complete.
4.2 Grammars for dCHAREs

The grammars for dCHAREs can be constructed by simulating the computations in the derivation system DC. To simplify the grammar, we define the function \( fl \) for a dCHAR \( r = (S, \alpha, \beta) \cdot r_1 \):

\[
fl(r) = \begin{cases} 
S & \lambda(r) = \text{true} \\
\emptyset & \text{otherwise}
\end{cases}
\]

This function makes the representation of nonterminal more concise. Otherwise, we have to use two instead of only one variable for the \( First \) set and the \( \lambda \) function. This will be clear later. The function \( fl \) can be computed as follows:

\[
fl((S, \alpha, \beta)) = \begin{cases} 
S & \alpha \in \{?, +\} \lor \beta \in \{?, +\} \\
\emptyset & \text{otherwise}
\end{cases}
\]

\[
fl(r \cdot (S, \alpha, \beta)) = \begin{cases} 
fl(r) & \alpha \in \{?, +\} \lor \beta \in \{?, +\} \\
\emptyset & \text{otherwise}
\end{cases}
\]

The correctness of the computation comes from the computation of \( \lambda \) and \( First \). Then we have the following property of \( fl \):

**Lemma 4.** Let \( r \) be a dCHAR. If \( \lambda(r) = \text{true} \), then we have \( First(r) = fl(r) \cup followLast(r) \).

**Proof.** For a dCHAR \( r = (S_0, \alpha_0, \beta_0) \cdot (S_1, \alpha_1, \beta_1) \cdots (S_n, \alpha_n, \beta_n) \), when \( \lambda(r) = \text{true} \):

when \( n = 1 \), \( fl(r) = S_0 \), \( First(r) = S_0 \), \( followLast(r) = \emptyset \) or \( S_0 \), then \( First(r) = fl(r) \cup followLast(r) \); when \( n \geq 2 \), \( fl(r) = S_0 \), \( First(r) = S_0 \cup S_i \), \( followLast(r) = \cup S_i \), \( S_i \) are the terminal sets which satisfies \( \lambda((S_0, \alpha_0, \beta_0)\cdot(S_1, \alpha_1, \beta_1)\cdots(S_{n-1}, \alpha_{n-1}, \beta_{n-1})) = \text{true} \), then \( First(r) = fl(r) \cup followLast(r) \). So we have \( First(r) = fl(r) \cup followLast(r) \) for that \( \lambda(r) = \text{true} \). 

Now consider how to construct grammars for dCHAREs. Let \( \Sigma = \{a_1, \cdots, a_n\} \).

Suppose there is a finite set \( X \) of nonterminals. Each nonterminal is of the form \( X^{R,F,\alpha,\beta} \), where \( R, F \subseteq \Sigma, \alpha, \beta \in \{a, +, ?, *\} \). \( X^{R,F,\alpha,\beta} \) is intended to define the language \( L(X^{R,F,\alpha,\beta}) = \{ r \in dCHAREs \mid followLast(r) = R, fl(r) = F, r = r_1 \cdot (S, \alpha, \beta), r_1 \in dCHAREs \} \) for fixed \( R, F, \alpha, \beta \). Denote the grammars by \( G_c \), the grammar rules of \( G_c \) are:

**Fac:**

\[
X^{R,F,\alpha,\beta} \rightarrow \bigcup (S, \alpha, \beta)
\]

\[
(\alpha \in \{+, *\} \lor \beta \in \{+, *\}) \rightarrow (R = S)
\]

\[
(\alpha \notin \{+, *\} \land \beta \notin \{+, *\}) \rightarrow (R = \emptyset)
\]

\[
(\alpha \in \{?, *\} \lor \beta \in \{?, *\}) \rightarrow (F = S)
\]

\[
(\alpha \notin \{?, *\} \land \beta \notin \{?, *\}) \rightarrow (F = \emptyset)
\]

**Seq:**

\[
X^{R,F,\alpha,\beta} \rightarrow \bigcup X_{R_1,F_1,\alpha_1,\beta_1} \cdot (S, \alpha, \beta)
\]

\[
(\alpha \in \{?, *\} \lor \beta \in \{?, *\}) \rightarrow (R = R_1 \cup S)
\]

\[
(\alpha \notin \{?, *\} \land \beta \notin \{?, *\} \land \alpha = + \lor \beta = +) \rightarrow (R = S)
\]

\[
(\alpha = o \land \beta = a) \rightarrow (R = \emptyset)
\]

\[
(\alpha \in \{?, *\} \lor \beta \in \{?, *\}) \rightarrow (F = F_1)
\]

\[
(\alpha \notin \{?, *\} \land \beta \notin \{?, *\}) \rightarrow (F = \emptyset)
\]

\[
R_1 \cap S = \emptyset
\]

\[
F_1 \cap S = \emptyset
\]
The conditions of these rules are used to check the determinism of dCHAREs. Hence the productions for grammars of dCHAREs can be constructed as follows: The productions in the Fac case are straightforward, the conditions in the rule correspond to compute the followLast and fl sets. For the productions in the Seq case, the first five conditions are used to compute followLast and fl sets, the sixth condition checks that for the concatenation expression \( r = s \cdot t \), whether followLast\((s) \cap \text{first}(t) = \emptyset \), the seventh condition checks that for the concatenation expression \( r = s \cdot t \), if \( \lambda(s) = \text{true} \), then first\((s) \cap \text{first}(t) = \emptyset \) must hold. Thanks to fl, we only need to check whether fl\((s) \cap \text{first}(t) = \emptyset \).

The form of productions in grammars \( G_c \) conform to the definition of left-linear grammar [30], i.e., \( G_c \) are regular, then we come to the theorem:

**Theorem 2.** dCHAREs can be defined by regular grammars.

### 4.3 Scale of Grammars

Given an alphabet \( \Sigma \), the terminals in grammars \( G_c \) are all the factors over \( \Sigma \). Let us consider the size of \( G_c \). Since there are \( 2^{2|\Sigma|} \) different fl and followLast sets, there are \( 2^{|\Sigma|} \cdot 2^{|\Sigma|} \cdot 4 \cdot 4 = 2^{2|\Sigma|+1} \) different nonterminals. Given a set \( S \), an \( \alpha \) and a \( \beta \), because all possible permutations of the symbols in \( S \) can form a factor, there are \(|S|!\) possible factors. For example, given \( S = a, b, c \), \( \alpha = o \) and \( \beta = * \), we have the following factors: \((a|b|c)^*\), \((a|c|b)^*\), \((b|a|c)^*\), \((b|c|a)^*\), \((c|a|b)^*\) and \((c|b|a)^*\). Each production in the grammars uses at most two nonterminals and one terminal, then the number of the productions is \( O(2^{|\Sigma|} \cdot 2^{|\Sigma|} \cdot |\Sigma|!) = O(2^{|\Sigma|^2} \cdot |\Sigma|!) \).

| \(|\Sigma|\) | \(|G_d|\) | \(|G_c|\) |
|-------|-------|-------|
| 1     | 46    | 12    |
| 2     | 815   | 897   |
| 3     | 14904 | 3104  |
| 4     | 240481| 12480 |
| 5     | 3520010| 44192 |
| 6     | 48369939 | 146496 |
| 7     | 638241628 | 467744 |

For a nonterminal \( X^{R,F,\alpha,\beta} \), we call it is useless if there not exists dCHAREs \( r \) such that \( X^{R,F,\alpha,\beta} \Longrightarrow^* r \). For a production in grammars of dCHAREs, if it contains useless nonterminal, the production is useless. The useless productions accounts for a large proportion in grammars of DREs [35], but the grammars of dCHAREs directly produce the useful productions according to DC. To verify the conciseness of dCHAREs grammars, we compared the number of productions in grammars of DREs \( (G_d) \) and grammars of dCHAREs \( (G_c) \) in Table 1. Result shows that \( G_c \) are smaller in order of magnitude than \( G_d \).

### 5 Random Generation Algorithm

This section shows how to use the grammars to randomly generate dCHAREs whose length is exactly equal to the given value. Considering the chain structure of dCHAREs, each chain factor has two nested unary operators and the inner operators are identical, besides, the right side of the productions are a nonterminal connected to a factor, so we adopt a bottom-up manner to generate dCHAREs.

The generation algorithm is shown in Algorithm 2, which takes an alphabet \( \Sigma \) and a length \( N \) as input, returns a dCHARE. The variables have the following
meaning: $S$ for the list of symbols in each factor, \textit{divide} records the partition of $N$, i.e. the length of each factor, \textit{termin} records random symbols of the current factor. $R, F, A, B$ stand for the set $R, F, \alpha, \beta$ of nonterminals respectively, and unary operators $\alpha, \beta \in \{?, *, +, o\}$. \textit{ranDivide}(\(N, |\Sigma|\)) randomly writes $N$ as a sum of positive integers like integer partition [9], but each integer is limited to no more than the alphabet size $|\Sigma|$, and the order of integers is considered (line 2). That ensures the length of the dCHARE we generate is equal to $N$ and the correctness of the generation algorithm. \textit{ranSample}(\(A, n\)) randomly chooses $n$ different elements from $A$ (line 4, 11 and 15). The generator produces the left-most factor according to the value of $\alpha, \beta, S$ and \textit{Fac} productions, calculate $R$ and $F$ by grammar rules to obtain the nonterminals and productions (line 7), then generate the rest factors from left to right, based on $Seq$ and the terminals that has been generated, generating the dCHARE at the same time of randomly selecting grammar productions(line 8-20). If the grammatical restriction interrupts a generation process, the algorithm will randomly re-divide $N$ then generate a new dCHARE (re-generate, line 22). Experiment shows that when $N \leq 6|\Sigma|$, the success rate of generate one dCHARE without re-generate can reach to 100%, in other cases, the algorithm can also terminate in a limited time and generate a dCHARE, so the finiteness and effectiveness of the algorithm can be guaranteed.

\textbf{Algorithm 1:} \textit{ranDivide}

\begin{verbatim}
Input: a positive integer $N$ and the upper bound of each portion $\mathcal{M}$
Output: a random divide list of $N$

1  $res = []$; $current, now = 0$
2  \textbf{while} $current < N$ \textbf{do}
3    $res$.Append($now$)
4    now = a random integer range from 1 to $\mathcal{M}$
5    $current+$ = now
6  $current-$ = now; $res$.Append($N - current$)
7  \textbf{return} $res$
\end{verbatim}

\textbf{Example:} input $\Sigma = \{a, b, c\}$, $N = 5$, the output of Algorithm 2 is $(S, A, B)$, where $S = [\{c\}, [b, a], [c], [a]]$, $A = [o, o, o, +], B = [+, *, +, +]$. So the dCHARE generated is $(c)^+ \cdot (b(a)^* \cdot (c)^+ \cdot (a^+)^*)$. Grammar production used in the derivation are: $X(\{c\}, o, o, +) \rightarrow (c)^+$, $X\{a, b, c\}, o, o, * \rightarrow X\{c\}, o, o, + \cdot (b)^{o^*}, X\{c\}, o, o, + \rightarrow X\{a, b, c\}, o, o, * \cdot (c)^{o^*}, X\{a, c\}, o, +, * \rightarrow X\{c\}, o, o, + \cdot (a^+)^*$. 

\textbf{Theorem 3.} Complexity of our generation algorithm is $O(|\Sigma|^2 \log^2(N))$ in time and $O(|\Sigma| \log(N))$ in space.

\textbf{Proof.} Unlike the classical recursive method, there is no preprocessing. The time of functions \textit{append} is $O(1)$, and we randomly choose an integer in $O(1)$ time, so the time complexity of Algorithm 1 is $O(\log(N))$. The time complexities of calculating the subtraction and union set are $O(|\Sigma|)$ and $O(|R_{i-1}| + |S_i|) = O(|\Sigma|)$. $S_i$ is a random list ($|S_i| = \text{divide}[i]$) generated from the candidate symbol set $SC$, which is a subset of $\Sigma$, so the time complexity of generating $S$ is $O(|SC_i| * O(|\Sigma|) * O(\text{len(divide)}) = O(|\Sigma|^2 \log(N))$. Then the time of generating one factor is $O(|\Sigma|^2 \log(N)) + O(|\Sigma|)$, so the time complexity of Algorithm 2 is $O(\log(N)) * O(|\Sigma|^2 \log(N) + |\Sigma|) = O(|\Sigma|^2 \log^2(N))$. Counting the listed variables, we obtain the space complexity is $O(|\Sigma| \log(N))$. $\square$
Algorithm 2: Random Generate dCHAREs

Input: an alphabet $\Sigma$, a length $N$
Output: a dCHARE of length $N$

divide, $S, R, F, A, B = []$; is,oo, flag = true

divide = ranDivide($N, |\Sigma|$)

$A[0], B[0] = \text{randomly chooses two operators from } \{?, \ast, +, o\}$

$\text{termin} = \text{ranSample}(\Sigma, \text{divide}[0]); S[0] = \text{termin}$

if $(A[0], B[0]) = (o, o)$ then

is,oo = true

Assign $S[0]$ or $\emptyset$ to $R[0]$ and $F[0]$ according to Fac of $G_c$

for $i \in \{1, \ldots, \text{divide} - 1\}$ do

$A[i], B[i] = \text{randomly chooses two operators from } \{?, \ast, +, o\}$

if is,oo = false then

$\text{termin} = \text{ranSample}(\Sigma, \text{divide}[i])$

else

if divide[i] > $|\Sigma| - |S[i] - 1|$ then

flag = false; break

$\text{termin} = \text{ranSample}(\Sigma - S[i - 1], \text{divide}[i])$

$S[i] = \text{termin}$

if $(A[i], B[i]) = (o, o)$ then

is,oo = true

else

Calculate $R[i]$ and $F[i]$ according to Seq of $G_c$

if flag = false then

Random Generate dCHAREs($N, \Sigma$)

else

return $(S, A, B)$

6 Experiments

6.1 Constructing Grammars for dCHAREs

Table 2 shows the time of constructing grammars with small alphabets, where $|G_c|$ denotes the number of productions in the grammars $G_c$, $\text{Time}(ms)$ and $\text{Avg}_T(\mu s)$ denote the constructing time for grammars and the average time for each production respectively. Although grammars can be constructed easily for small alphabets, the result shows that the construction time is exponential in $|\Sigma|$. This is consistent with the number of productions given in Section 4.3. The average time for constructing one production of grammars is shorter than 1 $\mu s$, which enables us to efficiently generate dCHAREs by only constructing the productions when they are needed.

6.2 Randomly Generating Expressions with the Given Length

In this section, we present some experiments to evaluate our random generation algorithm for dCHAREs.
Table 2. Grammars construction

| $|\Sigma|$ | $G_c$ | Time (ms) | Avg.T ($\mu$s) |
|---|---|---|---|
| 1 | 32 | 0.075 | 1.219 |
| 2 | 576 | 0.425 | 0.518 |
| 3 | 3104 | 1.358 | 0.361 |
| 4 | 12480 | 7.096 | 0.407 |
| 5 | 44192 | 97.223 | 1.012 |
| 6 | 146496 | 382.624 | 1.094 |
| 7 | 467744 | 1030.292 | 1.076 |
| 8 | 1460160 | 3597.424 | 1.232 |
| 9 | 4494752 | 24703.156 | 1.374 |
| 10 | 13713216 | 12944.305 | 1.555 |

Fig. 1. Generation time on various $|\Sigma|, N$

We conduct experiments on grammars with different alphabet sizes ($3 \leq |\Sigma| \leq 10$) to generate dCHAREs with different lengths ($1 \leq N \leq 22$). Fig. 1 exhibits the average time of generating one dCHARE for the given $N$ and $|\Sigma|$, the generation time is less than $2 \text{ ms}$ which is acceptable in practice. It also implies that if the user requires a short generation time, the relative value of $|\Sigma|$ and $N$ should be taken into account.

Our generator is more efficient and practical comparing with algorithm in [35]. The comparison on average time of generating one expression with $|\Sigma| = 20$ is shown in Fig. 2. The size of the grammars supported by our algorithm is much larger than that of Xu et al., because the running time of their algorithm increases rapidly with the increase of $|\Sigma|$.

Table 3. Generation time on large $|\Sigma|, N$

| $|\Sigma|$ | $N$ | Time (ms) |
|---|---|---|
| 50 | 100 | 0.783 |
| 100 | 0.127 |
| 300 | 0.165 |
| 500 | 0.1342 | 6.846 |
| 50 | 0.817 | 4.055 |
| 200 | 0.266 | 6.379 |

The largest $|\Sigma|$ given in [35] is 27, which takes an average of 58.3 s to generate one expression with $N \leq 500$. We would also like to see how our algorithm performs when generating long dCHAREs from large grammars. So we respectively generate 100 dCHAREs with expressions length ($N$) equals 100, 200, 300, 500 from grammars whose alphabet size ($|\Sigma|$) equals 50, 100, 200. Table 3 shows the average time of generating a dCHARE.

7 Conclusion

In this paper, we have given regular grammars for dCHAREs. We constructed the grammars, then designed a random generator for dCHAREs which solved the problem of randomly generate expressions of the given length in [35]. With the grammars and the generator for dCHAREs, a series of problems can be solved concisely. For instance, dCHAREs defined in this paper had no inference algorithms before and can be inferred based on the grammars we proposed. On the other hand, we can generate a large amount of expressions randomly to be used as input to various test programs. The grammars and generator can also help to determine whether a regular expression can be converted to
dCHAREs (see example in Sec 5). Since the grammars we proposed are regular, our generator can be extended to other subclasses that can be expressed by regular grammars.

**Future work.** (1) Optimizing the random generation process based on the distribution parameters of generated expressions. The purpose of the optimization is to approximate the randomly generated expression to a uniform distribution. (2) Other applications of the grammars. For example, further develop a tool to help users writing dCHAREs.

**References**

34. W3C: Unique Particle Attribution. https://www.w3.org/wiki/UniqueParticleAttribution, 27 September 2005