国際化正規表現ライブラリの開発

長谷川 勇 <isamu@yamato.ibm.com>
日本アイ・ビー・エム株式会社 ソフトウェア開発研究所

平成 14 年 7 月 31 日

概要: GNU C Library (以下 glibc) には、すでに国際化アプリケーションを作成するために必要な API が用意されている。しかし、glibc-2.2.x の正規表現関数においては、複数文字照合要素の扱いなど、国際化の観点からはまだ十分ではない。我々は glibc のメンテナ Ulrich Drepper 氏の協力のもと国際化正規表現ライブラリの実装を行った。我々の実装は POSIX 準拠の国際化、DFA ベースの実装などの特徴を持ち、現在 glibc-2.3 の CVS ツリーにマージされている。本論文では、国際化正規表現の解説を行い、従来の正規表現ライブラリの問題点と我々の実装の特徴、成果を述べる。

1 はじめに

近年、 Linux の普及に伴うオープンソースソフトウェアの需要の増加に従い、それらの国際化や多言語化の要求が高まっている [1]。

Linux においては、コアとなる GNU C Library (以下 glibc)[2] に、POSIX ロケールモデル [3] に基づいた国際化ソフトウェアを作成するために必要な API が用意されている。しかし、glibc version 2.2 系列までの正規表現関数 [4] においては、マルチバイトキャラクタへの対応はなされているが、国際化はまだ十分ではない。

我々は glibc のメンテナ Ulrich Drepper 氏の協力のもと国際化正規表現ライブラリの実装を行った。

本論文では、まず正規表現と国際化に関して説明を行う。次に現在の glibc の正規表現ライブラリの問題点、我々が実装を行う際の問題点と解決方法を述べる。最後にその成果を示す。

2 正規表現

2.1 ソフトウェアにおける正規表現

正規表現とは文字列中のパターンを発見/操作するためのパターンマッチングの機構であり、 sed, grep, awk を始めとするテキスト処理ツールに広く利用されている。例えば、正規表現 “`(ab)*`” は “`ab`” の 0 回以上の繰り返し (“”, “`ab`”, “`abab`”, ...) にマッチし、“`[acd]`” は “`a`” または “`c`” または “`d`” にマッチする。

2.2 正規表現の評価器

正規表現の評価器には有限状態オートマトンを用いる。有限状態オートマトンとは、開始ノードと停止ノードを持つラベルつきの有向グラフであり、非決定性有限状態オートマトン (以下 NFA) と、決定性有限状態オートマトン (以下 DFA) の 2 つに分類できる。DFA は処理速度に優れており、GNU grep, GNU awk などのツールが採用している。

本節では、NFA と DFA について解説する。

NFA ここでは、NFA を 5 つ組 (K_N, T, t_N, k_1, F_N) で定義する。ただし、

- K_N : 状態の有限集合
- T : 入力文字の有限集合
- t_N : 現在の状態, 入力, 遷移先状態の関係の有限集合
  $t_N = \{(s_s, C, s_d) | s_s \in K_N, C \subset T, L \subset K_N,$ s_s は $c \in C$ を受理して s_d に遷移可能 $\}$

($C = \phi$ を許す[1])

- k_1 : 開始状態
- F_N : 停止状態の集合 ($F_N \subset K_N$)

例えば、正規表現 “`([ab]c)*ac`” からは以下の NFA を構成できる (図 1)。

$$
\begin{aligned}
&(\{0,1,2,3\}, \{a,b,c\}, \\
&\ \{ (0,\{a,b\},\{1\}), (1,\{c\},\{0\}), (0,\{a\},\{2\}), \\
&\quad (2,\{c\},\{3\})\}, \\
&\ 0, \{3\})
\end{aligned}
$$

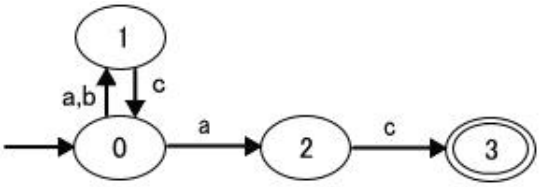

図 1: NFA “`([ab]c)*ac`”

有限状態オートマトンの状態遷移　このオートマトンに対して、入力 “`acbcac`” を与えると、以下のように状態遷移を行うことで、停止状態 3 に遷移する。

$$0 \xrightarrow{a} 1 \xrightarrow{c} 0 \xrightarrow{b} 1 \xrightarrow{c} 0 \xrightarrow{a} 2 \xrightarrow{c} 3$$

このため、このオートマトンは入力を受理でき、正規表現 “`([ab]c)*ac`” は、入力 “`acbcac`” とマッチする。

一方、入力 “`aa`” に対しては、いかなる順序で状態遷移を行っても終了状態に遷移できず途中で停止する。このため、このオートマトンは入力を受理することができず、正規表現 “`([ab]c)*ac`” は入力 “`aa`” とマッチしない。

なお、t は、ある状態、入力に対して複数の遷移先を持ち得る。図 1 の例では、状態 0 から 入力 a に対し、遷移先の状態が、1, 2 の 2 通りある。このため、NFA は “非決定性” である。

DFA の生成　DFA は、正規表現から NFA を構成し、そこから DFA に変換することで構成する。

[1] $C = \phi$ の場合、文字を受理せず状態遷移し、これを ϵ 遷移とよぶ

ここでは、 NFA (K_N, T, t_N, k_1, F_N) を DFA $(K_D, T, t_D, K_{D1}, F_D)$ に変換する手順を示す。ただし、t_D は、$t_D = \{(k_s, a, k_d) | k_s \in K_D, a \in T, k_d \in K_D, k_s$ は a を受理して k_d に状態遷移可能 $\}$ である。

1. 次の手順で K_{D1} を求め、 $M = K_{D1}, K_D = \{M\}, t_D = \phi$ とする。
   a. $K_{D1} = \{k_1\}$
   b. ${}^{\forall}s(s \in K_N)$ に対して、$(s_s, \phi, s) \in t_N \wedge s_s \in K_{D1}$ ならば、$K_{D1} \cup \{s\}$ を新たな K_{D1} とする。
   c. K_{D1} の要素が増えなくなるまで b を繰り返す。
2. ${}^{\forall}a \in T$ に対して、
   a. $L = \{s_d | (s_s, N, s_d) \in t_N, s_s \in M, a \in N\}$
   b. $K_D \cup \{L\}$ を新たな K_D とする。
   c. $t_D \cup \{(M, a, L)\}$ を新たな t_D とする
3. 2 で増えた K_D の要素すべてを M とし 2, 3 を適用
4. $F_D = \{S | S \in K_D, {}^{\exists}k(k \in S, k \in F_N)\}$

ここで定義する DFA の t_D に対して、

$$
\begin{aligned}
&{}^{\forall}t_1 {}^{\forall}t_2 (s_1 = s_2 \cap a_1 = a_2 \rightarrow t_1 = t_2) \\
&\qquad (t_i = (s_i, a_i, d_i) \in t_D)
\end{aligned}
$$

であり、現在の状態と入力文字から一意に遷移先の状態が決定するため、このオートマトンは “決定性” である。

図 1 の NFA を変換すると次の DFA が得られる (図 2)。

$$
\begin{aligned}
&(\{\{0\},\{1\},\{1,2\},\{0,3\}\}, \{a,b,c\}, \\
&\ \{ (\{0\},a,\{1,2\}), (\{0\},b,\{1\}), \\
&\quad (\{1\},c,\{0\}), (\{1,2\},c,\{0,3\}), \\
&\quad (\{0,3\},b,\{1\}), (\{0,3\},a,\{1,2\})\}, \\
&\ \{0\}, \{\{0,3\}\})
\end{aligned}
$$

2.3　言語理論における正規表現との相違

2.1 において、正規表現とは文字列中のパターンを発見/操作するためのパターンマッチングの機構であると述べた。しかし、本来正規表現とは

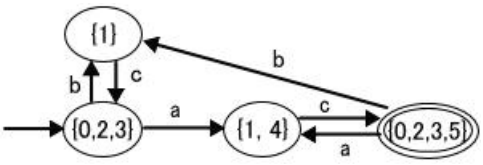

図 2: DFA ([ab]c)*ac

Stephen Kleene らによって考案された正規言語の表現形態の一つである [5] 。以下に正規言語 L と正規表現 r の例を示す。

$$L_a = \{a, b, d\} \quad , \quad r_a = a|b|d$$
$$L_b = \{abc, abd\} \quad , \quad r_b = ab(c|d)$$
$$L_c = \{\epsilon, ab, abab, ababab, \ldots\} \quad , \quad r_c = (ab)*$$

言語理論における正規表現は、「ある文字列がその正規表現があらわす正規言語に含まれるか」という命題までを扱う概念である。これに対し、ソフトウェアにおける正規表現では、「ある文字列のどの部分がその正規表現があらわす正規言語に含まれるか」という判定を必要としている。さらには、「ある文字列のどの部分が、その正規表現のどの部分にマッチしたか」を要求されることもある。この、正規表現がどの部分にマッチしたかの判定は、従来の言語理論の正規表現の範疇を超えたものであることに注意しなければならない。

3 国際化

国際化 [6] とは、国や言語、文化などによって異なる処理をロジックから切り離し、プログラムを汎用化することを言う。

本章では、国際化に関して POSIX ロケールモデルと照合に関して簡単に述べ、正規表現を国際化するうえでの注意点を述べる。

3.1 POSIX ロケールモデル

POSIX ロケールモデルとは国、言語、習慣などに依存して変化する動作を実行時に切り替える仕組みである。プログラムは POSIX ロケールモデルを利用することで、ロジックからそれらの地域依存の部分を切り離すことができる。

具体的には、プログラムは setlocale() API により、それが参照するロケールデータベースを切り替えることができる。ロケールデータベースには 通貨記号、日時のフォーマット、使用する文字とその照合順序 (後述) などが格納されており、プログラムは国際化 API を用いることで、その動作をロケールに従ったものにでき、国際化が可能となる。

このロケール名には、言語と国または地域の名前を指定する。例えば、da_DK ロケールはデンマーク語/デンマークを表し、de_DE ロケールはドイツ語/ドイツを表す。さらに、そのロケールにおいて使用するエンコーディングを指定することもある。例えば、 ja_JP.EUC–JP ロケールは日本語/日本/EUC–JP を表す。

3.2 国際化における文字列の照合

照合 (Collation) とは、辞書順などで用いる文字列の比較を言う。照合は、文字列を照合要素 (Collating Element) と呼ばれる単位に分解し、2 つの文字列を照合要素ごとに比較することで行う[2]。このとき、照合要素として扱う文字や文字列の定義、照合要素間の順序関係である照合順序 (Collation Order) は、ロケールによって異なる。

このため、国際化アプリケーションは、文字列の照合を行う際に `strcmp()` やコードポイントによる比較ではなく、ロケールデータベースの照合順序を参照する `strcoll()`, `wcscoll()` といった API を使わなければならない。

さらに、アプリケーションによっては文字列内の照合要素を扱わなければならない場合がある。このとき、国際化アプリケーションでは “1 照合要素 =1 文字” という仮定をしてはならない。“1 照合要素” は “1 文字” とは限らず、“1 文字” も “1 照合要素” とは限らないためである。

例えば、デンマーク語やノルウェー語における “`aa`”、スペイン語における “`ch`”, “`ll`” は “1 照合要素” として扱う。これらの複数の文字から成る照合要素を複数文字照合要素 (Multi Character Collating Element) と言う。da_DK ロケールにおいては、“`aa`”, “å”, “ǻ” を比較すると、

$$\text{å} < \text{aa} < \text{ǻ}$$

[2]厳密には、照合要素の Weight ごとに文字列の走査を複数回行うことがあるが、ここでは省略する

という順序になり、“aa” は “å” と “á” の間の 1 照合要素として扱われる。

一方、ドイツ語における ßは、 “2 照合要素” として扱う。de_DE ロケールにおいては、“ssa”, “ßb”, “ssc” を比較すると、

$$\text{ssa} < \text{ßb} < \text{ssc}$$

という順序になり、“ß” は “ss” とほぼ同等に扱われる。

3.3 正規表現における国際化

国際化正規表現では、まずマルチバイトキャラクタへの対応が必要である。つまり、正規表現中の各要素を文字単位で扱わなければならない。例えば、ja_JP.EUC–JP ロケールにおいて、正規表現 “あ*” は、文字 “あ” の 0 回以上の繰り返しであり、これを byte 単位に 「“あ” の 1 byte 目 1 つと 2 byte 目の 0 回以上の繰り返し」と処理してはならない。

また、国際化に必要な作業は、マルチバイトキャラクタへの対応を行えば十分だと考えられがちだが、正規表現の場合には、さらに複数文字照合要素が大きな問題となる。正規表現のうち、“[a-c]”, “[abc]” などの bracket expression は、照合要素 1 つとマッチすることができるため、複数文字照合要素を持つロケールにおいては、1 文字だけではなく、複数の文字にもマッチできる場合があるためである。

例えば、da_DK ロケールにおいて、“[å-á]” という正規表現は、 “aa” 2 文字にマッチすることができる。

4 glibc の正規表現関数の問題点

glibc-2.2.x の持つ正規表現関数の実装には、いくつか問題がある。ここでは、これらの問題点に関して述べる。

32KB の制限: 正規表現は内部で byte code へとコンパイルされるが、byte code 内部で使用するアドレスが 2 byte の符号付き整数としてハードコードされているため、正規表現の長さは最大でも 32 KB を超えることができない。

正規表現中の照合要素: 正規表現中の照合要素を正しく扱えない場合がある。例えば、正規表現 “[=å=]” は da_DK ロケールで有効な正規表現だがコンパイルエラーになってしまう。

入力中の照合要素: 入力中の複数文字照合要素を正しく扱えない。例えば、正規表現 “[å-á]” は、da_DK ロケールで “aa” にマッチするはずだが、現在の実装ではこれらを正しく扱えない。

不十分な POSIX 準拠: POSIX の規格としては正しいが、コンパイルできない正規表現がある。例えば、現在の実装で “[[.a.]-b]” はコンパイルできるが、“[a-[.b.]]” はコンパイルエラーになってしまう。

マルチバイトキャラクタの扱い: 本来マルチバイトキャラクタを考慮した設計になっていないため、マルチバイトキャラクタを使うロケールでパフォーマンスに問題がある。

実装のバグ: 最初に作られてから 10 年近くたっているにもかかわらず、度重なる機能拡張のためにバグも枯れきっていない。

例えば、正規表現 “a(b*)\1+” に入力文字列 “a” を与えると無限ループに陥ってしまう。

5 国際化正規表現ライブラリの実装

我々は、前章で述べた問題から、既存の正規表現ライブラリを修正するより、新しく実装を行う方が効率がよいと判断し、国際化正規表現ライブラリの実装を行った。

本実装では、パフォーマンスの面でアドバンテージを得るために、DFA をベースにした実装を行った。ところが DFA をベースに正規表現ライブラリを実装するには、いくつかの問題がある。以前我々は、DFA でマルチバイトキャラクタを扱う方法を紹介 [7] したが、これ以外にもマッチングの範囲の問題がある。

本章では、この問題点と我々の解法を述べる。

表 1: 正規表現と状態の対応

NFA の状態	0	1	2	3	4	5	6	7
正規表現の要素	(	[ab]	c	)	*	a	b	(終端)

5.1 DFA の制限

従来 DFA はすべての正規表現を扱え、その能力は NFA と等価であるとされている。しかし、これは言語理論における正規表現に関しての話であり、ソフトウェアの世界には当てはまらない。

特に正規表現ライブラリを実装するときには、2.3 で述べたマッチングの範囲が問題となる。さらに POSIX においては、正規表現中の括弧で囲われた部分正規表現すべてに対し、入力文字列のどこにマッチしたかの判定を要求している。

例えば、“`([ab]c)*(ac)*`” に入力 “`acac`” を与えた場合、POSIX の最長最左ルールにより、正規表現のうち “`([ab]c)*`” が入力文字列にマッチする。このとき評価器は、入力文字列の 2 つめの “`a`” が、正規表現の “`([ab]c)*`” に含まれる “`a`” にマッチしたのか、それとも正規表現の後半の “`(ac)*`” に含まれる “`a`” にマッチしたのかを判定しなければならない。しかし、これらの情報は DFA への変換で失われているため、DFA では処理できない。

5.2 解決法

本論文の方法[3]は、DFA が遷移した状態を記録しておき、その状態の列から、対応する NFA を用いた場合にたどる状態の列を得ることで、DFA への変換で失われた必要な情報を取得する。

なお、NFA の構成にはいくつもの方法があり、正規文法の観点からは等価な NFA が複数構成できる。我々の方法は、対応する NFA の状態列から情報を取得するため、どのような NFA を構成するかがポイントとなる。

ここでは、まず我々の採用した NFA/DFA の構築法を述べ、状態列の取得、NFA の状態列の復元方法を述べ、マッチした部分を特定する方法を述べる。次にマルチバイトキャラクタを扱う方法を述べる。

[3]back-reference やコンテキストに関しては本論文では省略する。

NFA/DFA の構築 我々の方法では、まず正規表現の各要素に NFA の状態を 1 つ割り当て、正規表現の要素と NFA の状態を 1 対 1 に対応させる。ここで、正規表現の要素とは、文字 (“`a`” など)、bracket expression(“`[b-c]`” など)、選択・繰り返しの演算子 (“`*`”, “`|`” など)、部分正規表現を表す括弧などである。また、正規表現の終端を表す仮想的な要素を導入し、 NFA の終了状態を割り当てる。

次に以下で、遷移規則を決定する。

- 文字、bracket expression は、それらが表す文字または文字列を受理する遷移先をただ 1 つ持ち、ϵ 遷移はしない。
- 選択の演算子は文字を受理せず、ϵ 遷移先を必ず 2 つ持つ。
- 繰り返しの演算子は文字を受理せず、ϵ 遷移先を 2 つ持ち、それぞれ繰り返しを続行する遷移先と繰り返しを終了する遷移先である。
- 括弧は文字を受理せず、ϵ 遷移先をただ 1 つ持つ。

例えば、図 1 で示した正規表現 “`([ab]c)*ac`” の NFA をここで述べた方法で構築すると図 3 になる。また、このときの正規表現の各要素と NFA の状態の対応関係を表 1 に示す。

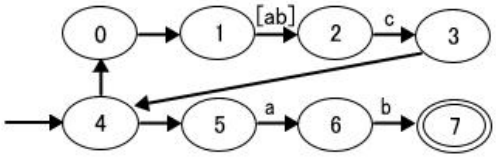

図 3: NFA “`([ab]c)*ac`”

また、DFA の構築は通常の方法で行う。

状態列の取得 まず、入力 $(c_1, c_2, \ldots c_n)$ を DFA に与え評価を行ったときの状態列 $(S_{D0}, \ldots, S_{Dn})$ を考える。このとき途中で停止状態に達しても停

止せず処理を続けることで、停止状態にたどり着く最長の部分文字列 (n はこの長さとする) を取得する。これには、対応する NFA が入力から状態遷移可能なすべての状態の列が含まれており、停止状態まで到達できないものも含まれる。以下の手順は最長の停止状態に到達できない NFA の状態を取り除く。

algorithm 1.

1. 次の手順で S'_{Dn} を求め[4]、$i = n-1$ とする。
   a. $S'_{Dn} = \{s | s \in S_{Dn}, s \in F_N\}$
   b. $^{\forall}s(s \in S_{Dn})$ について、$(s, \phi, s_d) \in t_N \wedge s_d \in S'_{Dn})$ ならば、$S'_{Dn} \cup \{s\}$ を新たな S'_{Dn} とする。
   c. S'_{Dn} の要素が増加しなくなるまで b を繰り返す。
2. 次の手順で S'_{Di} を求める[5]。
   a. $S'_{Di} = \{s | s \in S_{Di} \wedge (s, c_{i+1}, s_d) \in t_N \wedge s_d \in S'_{Di+1}\}$
   b. $^{\forall}s(s \in S_{Di})$ について、$(s, \phi, s_d) \in t_N \wedge s_d \in S'_{Di})$ ならば、$S'_{Di} \cup \{s\}$ を新たな S'_{Di} とする。
   c. S'_{Di} の要素が増加しなくなるまで b を繰り返す。
3. $i > 0$ なら $i = i - 1$ として 2 へ

こうして得られた状態列は、与えられた正規表現にマッチする最長の文字列に対する NFA の状態列だが、一意には定まらない。これは、“*” が複数あった場合どれがより多くマッチするか、などが定まっていないためである。

NFA における状態列の復元　次の手順は、複数ある状態列の候補から、DFA が対応する NFA の評価器から得られる適切な状態列 $s_0, \ldots, s_m$ を得る手順である。

部分正規表現がマッチする候補が複数あったとき、評価器がどれを選択するかは、POSIX において次の文で規定されている。

> Consistent with the whole match being the longest of the leftmost matches, each subpattern, from left to right, shall match the longest possible string.

[4] b, c は入力文字列とは非依存なので、あらかじめ計算が可能。

我々はこの定義に従い、正規表現中でより左にある繰り返し演算子を優先させることにした。

algorithm 2.

1. $s_0 = k_1, i_D = 0, i_N = 1$
2. 以下のように s_{i_N} を求める。
   a. $^{\exists}s'((s_{i_N}, c_i, D) \in t_N \wedge s' \in D \wedge s' \in S'_{Di_D+1})$ ならば、$s_{i_N+1} = s'$, $i_D += 1$
   b. $^{\exists}s'((s_{i_N}, \phi, D) \in t_N \wedge s' \in D \wedge s' \in S'_{Di_D})$ ならば、s' を満たす要素を $s_A, s_B (s_A < s_B)$ として、$s_{i_N+1} = s_A$
3. $i_D \neq n \vee s_{i_N} \notin F_N$ ならば $i_N += 1$ として 2 へ

手順 2.a. において、s' が複数ある場合は、“`a|a`” など全く同じ式が “`|`” で並べられた場合のみであり[5]、どちらを選んでも同じである。

部分正規表現の特定　部分正規表現がマッチした文字列は状態列 s_{i_N} から容易に得られる。例えば、正規表現 “`([ab]c)*ac`” に入力 “`acac`” を与えると、

$$
\begin{aligned}
S_{Di} &: \{0,1,4,5\}, \{2,6\}, \{0,1,3,4,5,7\}, \{2,6\}, \\
&\quad \{0,1,3,4,5,7\} \\
S'_{Di} &: \{0,1,4\}, \{2\}, \{3,4,5\}, \{6\}, \{7\} \\
s_{Ni} &: 4, (\rightarrow)0, (\rightarrow)1, (\xrightarrow{a})2, (\xrightarrow{c})3, (\rightarrow)4, (\rightarrow)5, \\
&\quad (\xrightarrow{a})6, (\xrightarrow{c})7
\end{aligned}
$$

であり、部分正規表現 “`([ab]c)`” に対応する NFA の状態が、0, 1 なので、これらがマッチした入力の先頭の “`ac`” が得られる。

マルチバイトキャラクタへの対応　我々は DFA でマルチバイトキャラクタを扱う方法を紹介 [7] したが、その方法では前述の NFA の状態列を復元する方法と共存できなかったため、新しく次の方法を考案した。この方法は、DFA が matching の判定時に、入力の先読みと動的な状態の生成を行うことで、マルチバイトキャラクタや複数文字照合要素を扱う。

与えられた正規表現から NFA (K_N, T, t_N, k_1, F_N) を構成したとする。た

[5] そのような NFA を採用しているためであり、一般的な話ではない

だし、マルチバイトキャラクタや複数文字照合要素に対応するため、$t_N = \{(k, C, L) | k \in K_N, C \subset (T \cup T^2 \cup ... \cup T^m), L \subset K_N, k$ は $c \in C$ を受理して $^\forall k' \in L$ に状態遷移可能 $\}$ とする[6]。この正規表現に入力: $c_1, c_2, \ldots c_n$ が与えられた場合の処理の手順を以下に示す。

algorithm 3.

1. $K_D = \phi, t_D = \phi, i = 0, S_0 = \{k_1\}, S_j = \phi (j = 1, \ldots, n)$ [7]
2. $S_i \notin K_D$ ならば、$K_D \cup \{S_i\}$ を新たな K_D とし、$^\forall a \in T$ に対して次の手順を実行。
   a. $S_d = \{s_d | (s, C, D) \in t_N, s \in S_i, a \in C, s_d \in D\}$
   b. $t_D \cup \{(S_i, a, S_d)\}$ を新たな t_D とする
3. $(s, C, D) \in t_N, s \in S_i, a \in C, a \notin T$ を満たす (a, D) があれば、そのすべての組に対して次の手順を実行。
   a. a の各 byte を $c'_1, c'_2, ..., c'_l$ とする。
   b. $c'_1 = c_{i+1}, c'_2 = c_{i+2}, ..., c'_l = c_{i+l}$ ならば $S_{i+l} \cup D$ を新たな S_{i+l} とする。
4. $(S_i, c_{i+1}, S_d) \in t_D$ を満たす S_d が存在する場合、$S_{i+1} \cup S_d$ を新たな S_{i+1} とする。存在せず、$S_{i+1} = S_{i+2} = \ldots = S_n = \phi$ ならば、入力を受理せず終了。
5. 入力の終り $(i + 1 = n)$ でなければ $i = i + 1$ として、2 に戻る。$i + 1 = n$ の場合、$^\exists k (k \in S_{i+1}, k \in F_N)$ ならば入力を受理、そうでない場合は受理せず終了。

ここで、任意の正規表現、入力に対して、"algorithm 3 が入力を受理する" ことと "NFA が入力を受理可能" であることは等価である。すなわち、algorithm 3 は正規表現の評価器である。

例えば、正規表現 "`[a-b]|aab`" [8] を考える (図 4)。ただし、"`[a-b]`" は "`a`", "`aa`", "`b`" にマッチするとする。これに入力 "`aab`" を与えた場合、以下の順で処理を進め停止状態で終了する。このとき、最初の "`a`" による遷移は、"`a`" が、"`[a-b]`", "`a`" いずれにもマッチするため $S_1 = \{1, 2\}$、"`aa`" を先読みすることで、$S_2 = \{1\}$ となる。

$$
\begin{aligned}
&(i, S_0, S_1, S_2, S_3) : \\
&\quad (0, \{0\}, \phi, \phi, \phi) \xrightarrow{a} (0, \{0\}, \{1, 2\}, \{1\}, \phi) \\
&\quad \xrightarrow{a} (0, \{0\}, \{1, 2\}, \{1, 3\}, \phi) \\
&\quad \xrightarrow{b} (0, \{0\}, \{1, 2\}, \{1, 3\}, \{4\})
\end{aligned}
$$

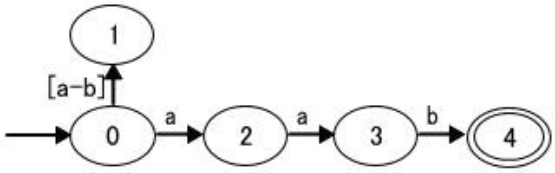

図 4: NFA "`[a-b]|aab`"

以上の方法を組み合わせることで、DFA を用いた国際化正規表現ライブラリを実装できた。

6 評価

ここでは、我々の実装した国際化正規表現ライブラリの特徴を述べる。

国際化: マルチバイトキャラクタ、複数文字照合要素を含む国際化を実現している。

DFA ベースの実装: 上述の通り、我々の実装は DFA をベースとしている。このため、NFA ベースの実装では避けられない back tracking の問題が生じない。

単純な例として、正規表現 "`a+a+a+...`"("`a+`" の数を n とする) と入力文字列 "`aaa...`" のマッチングは、従来の実装だと $O(2^n)$ 時間がかかるが、本実装では、$O(n)$ 時間で評価できる (表 2)。

また、従来の実装では back tracking によりリソースを消費しきらないように、re_max_failures という変数によって、back tracking の回数に恣意的な上限を与えている。このため、ある正規表現と入力がマッチングの候補を大量に持つ場合マッチしない問題があったが、本実装ではこうした上限を持たない。

国際化によるオーバーヘッド: 国際化に伴うオーバーヘッドは現在のマシンパワーでも無視できない。そこで、動的に MB_CUR_MAX の値を判定

[6] m はその locale でのマルチバイトキャラクタや複数文字照合要素のうち最長のものの byte 数

[7] 入力のみを変える場合は、$K_D = \phi, t_D = \phi$ は必要ない

[8] "`a`", "`aa`", "`b`", "`aab`" のいずれかにマッチする。

表 2: “a+a+a+...” の処理時間 (PentiumIII700MHz)

$n =$	1	...	15	16	17	...
従来の実装	0.003[s]	...	0.019[s]	0.036[s]	0.069[s]	...
本実装	0.003[s]	...	0.004[s]	0.004[s]	0.004[s]	...

表 3: テストプログラムの処理時間 (PentiumIII700MHz)

プログラム	runtests	runptests	tst-regex
従来の実装	0.005[s]	0.055[s]	2.127[s]
本実装	0.016[s]	0.038[s]	0.860[s]

することで、マルチバイトキャラクタなどが必要ないロケールでは、国際化に関する処理を省略し、オーバーヘッドを最小にしている。

実験　ここでは、従来の実装とパフォーマンスを比較するために、glibc 内の正規表現ライブラリのテストプログラムで速度を計測した (表 3)。以下に結果と考察を述べる。

- runtest: このテストは単純な正規表現/入力が多く含まれており、既存の実装の方が高速である。これは、正規表現の評価自体の時間がごくわずかであるため、本実装の DFA への変換などのオーバーヘッドの影響が大きかったと考えられる。これらの単純な正規表現に対するオーバーヘッドの削減は今後の課題である。

- runptest: このテストにおいては本実装の方が高速である。このテストには runtest よりも複雑な正規表現が含まれており、平均的なケースにおいては、本実装がパフォーマンスにおいて優れていると考えられる。

- tst-regex: このテストは ISO-8859-1 と UTF-8 の 2 つのコードセットに対するテストであり、この結果から、本実装はマルチバイトキャラクタ (ここでは UTF-8) の処理において特にパフォーマンスに優れていると考えられる。

7　まとめと今後の課題

glibc-2.2.x の正規表現ライブラリの問題点を指摘し、新たな国際化正規表現ライブラリの実装を行った。我々の実装は DFA ベースの実装であり、従来の NFA ベースの実装が持つ back tracking の制限を持たない。また、多くの場合においてパフォーマンスに優れている。

本実装は、GNU C Library の Version 2.3 branch に取り込まれており、CVS から取得することが可能である[9]。

今後の課題としては、単純な正規表現を処理する際のオーバーヘッドの減少がある。また、正規表現ライブラリは、従来の実装が示すようにバグが枯れにくいため、今後もバグフィックスを続ける必要がある。

謝辞

国際化正規表現ライブラリの開発にあたり、助言をしていただいた方々、特に Ulrich Drepper 氏, Arnold Robbins 氏, 杉山 昌治氏, 鷲澤 正英氏, 関場 治朗氏, 知念 充氏, 稗方 和夫氏に深く感謝いたします。

参考文献

[1] *LI18NUX2000 The Linux Globalization Specification*, 2000. http://www.li18nux.org/li18nux2k/.

[9] http://sources.redhat.com/cgi-bin/cvsweb.cgi/libc/?cvsroot=glibc

[2] *GNU C Library.*
http://sources.redhat.com/glibc/.

[3] *The Open Group Base Specifications Issue 6 IEEE Std 1003.1-2001*, 2001.

[4] Jeffrey E.F.Friedl. **詳説正規表現**. オライリー・ジャパン, 1999. 歌代 和正監訳.

[5] V.J.Rayward-Smith. **言語理論入門**. 共立出版, 1986. 井上謙蔵監訳.

[6] 清兼 義弘 末廣 陽一. **国際化プログラミング** – *I18N* **ハンドブック** –. 共立出版, 1998.

[7] 長谷川 勇. マルチバイトキャラクタを扱う決定性有限状態オートマトンの構成法. *Linux Conference 2001*, 2001.