制約オートマトン：実装と課題

新屋 良磨[1,a)]　藤浪 大弥[2,b)]

概要：制約オートマトンとは有限オートマトンに「数を数える能力」を付加した拡張モデルであり，例えば $\{a^n b^n c^n | n \geq 0\}$ のような非正規言語（かつ非文脈自由言語）を認識することができる．数を数える仕組みは Presburger 算術という値の足し算と比較のみを備えた算術体系によって実現される．本講演では制約オートマトンの基礎理論について紹介し，制約オートマトンをバックトラックで実装する際の問題点や最適化技法について議論する．

キーワード：オートマトン，Presburger 算術，grep

1. はじめに

制約オートマトン (*constrained automaton*) とは有限オートマトンに「数を数える能力」を付加した拡張モデルであり，例えば $\{a^n b^n c^n | n \geq 0\}$ のような非正規言語（かつ非文脈自由言語）を認識することができる．数を数える仕組みは Presburger 算術という値の足し算と比較のみを備えた算術体系によって実現される．

より具体的には，制約オートマトンは通常のオートマトンと**制約** *(constraint)* と呼ばれる**半線型集合**（有限的に記述できる自然数ベクトルの集合）が付随した計算モデルとなっている．半線型集合とは正規表現の類似体系で記述できる自然数ベクトルの集合を意味し，例えば３次元の対角線 $D_3 = \{(n, n, n) \mid n \geq 0\}$ は１つの３次元ベクトル $(1, 1, 1)$ を「任意回（０回を含む）足し合わせたもの」として $D_3 = (1, 1, 1)^*$ と Kleene 閉包 * を用いて表現できるため半線型集合である．

1　秋田大学 数理科学コース
2　国立情報学研究所
a)　ryoma@math.akita-u.ac.jp
b)　make.just.on@gmail.com

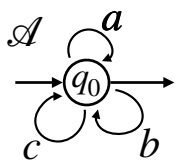

図 1　１状態のオートマトン $\mathcal{A}$

制約オートマトンが語 w を受理するとは，w による初期状態から終了状態への遷移列が存在（ここまでは通常のオートマトンと同様）し，さらにその遷移列中に現れる個々の状態遷移規則の回数をベクトル化したものが制約となる半線型集合に含まれることを言う（すなわち遷移列が制約を満たす必要がある）．具体例を見てみよう．図 1 のオートマトン $\mathcal{A}$ は初期状態かつ受理状態の q_0 という状態のみからなるオートマトンであり，通常のオートマトンの解釈では $\mathcal{A}$ の認識する言語は $\{a, b, c\}^*$ である．しかし，制約として D_3 を考え $(\mathcal{A}, D_3)$ を１つの制約オートマトンとしてみると，$(\mathcal{A}, D_3)$ が認識する言語は $\{w \in \{a, b, c\}^* \mid |w|_a = |w|_b = |w|_c\}$ となる．ここで $|w|_a$ は「w 中に現れる文字 a の個数」を意味する．なぜなら「a, b, c それぞれで定義された遷移規則を通った回数をベクトル化したも

のが D_3 に属す」とは「a, b, c を読む回数が同じ」ということに他ならないからである．

本講演では制約オートマトンの基礎理論について紹介し，制約オートマトンを用いて「数を数えることのできる正規表現エンジン」の実装や課題について議論する．

2. 半線形集合と制約オートマトン

$\mathbb{N}$ で自然数全体の集合（0 を含む）を表すことにする．d 次元の自然数ベクトル全体 $\mathbb{N}^d$ の部分集合 C が**半線形**であるとは，ある有限個のベクトル $\boldsymbol{v_0}, \boldsymbol{v_1}, \cdots, \boldsymbol{v_k} \in \mathbb{N}^d$ が存在して $C = \{v_0 + \sum_{i=1}^{k} x_i \boldsymbol{v_i} \mid x_i \in \mathbb{N}\}$ が成り立つことを言い，$\boldsymbol{v_0}$ を原点ベクトル，$\boldsymbol{v_1}, \cdots, \boldsymbol{v_k}$ を基底ベクトルと呼ぶ．例えば 3 次元の対角線 $D_3 = \{(n, n, n) \mid n \in \mathbb{N}\}$ は原点ベクトルとしてゼロベクトルを，基底ベクトルとして $(1, 1, 1)$ を取ることで上の形で表現できるため半線形集合である．

$C \subseteq \mathbb{N}^d$ が半線形であること，C が *Presburger* **論理式**で定義可能であることは同値であることが知られている．ここで Presburger 論理式とは和 $+$ と等号 $=$ が使える自然数上の一階算術であり，次の構文によって定義される論理式 ϕ である（t は自然数を表す項，x は変数である）．

$$t \coloneqq x \mid n(\in \mathbb{N}) \mid t_1 + t_2$$
$$\phi \coloneqq t_1 = t_2 \mid \exists x \phi(x) | \forall x \phi(x) | \phi_1 \vee \phi_2 | \phi_1 \wedge \phi_2 | \neg \phi$$

k 個の変数 $x_1, \ldots, x_k$ を自由変数に含む Presburger 論理式 $\phi(x_1, \ldots, x_k)$ について，ϕ を真とする変数の割当全体を $S(\phi) = \{(n_1, \ldots, n_k) \in \mathbb{N}^k \mid \phi(n_1, \ldots, n_k)$ が真$\}$ で表す．$C \subseteq \mathbb{N}^k$ が Presburger 論理式で定義可能であるとは，$S(\phi) = C$ となる Presburger 論理式 ϕ が存在することを言う．例えば，3 つの変数 x, y, z を自由変数に含む Presburger 論理式 $\phi(x, y, z) \coloneqq x = y \wedge y = z$ について，ϕ が真となる変数 x, y, z の割当は $x = y = z$ となる自然数の 3 つ組 $(n, n, n)(n \in \mathbb{N})$ であり，それら全体は 3 次元の対角線 D_3 となるため D_3 は Presburger 論理式で定義可能であることがわかる．

制約オートマトン [1] はペア $(\mathcal{A}, C)$ で，$\mathcal{A}$ は非決定性有限オートマトンを，$C \subseteq \mathbb{N}^d$ は半線形集合（を Presburger 論理式や基底ベクトルで表現したもの）をそれぞれ表す．ただし，オートマトン $\mathcal{A} = (Q, \delta \subseteq Q \times A \times Q, q_0, F)$ の状態遷移規則 δ は順序付けられた d 個の規則からなるものとする．オートマトン $\mathcal{A}$ について，$\mathrm{Runs}(p, q)$ で状態 p から状態 q への遷移全体の集合

$$\mathrm{Runs}(p, q) \stackrel{\mathtt{def}}{=} \{((q_0, a_1, q_1), \ldots, (q_{k-1}, a_{k-1}, q_k)) \in \delta^* \mid k \geq 0, q_0 = p, q_k = q\}.$$

を表し，$\mathrm{Runs}(q_0, F)$ で $\mathcal{A}$ の受理遷移列（初期状態から終了状態へと至る遷移列）全体の集合 $\mathrm{Runs}(q_0, F) = \bigcup_{q \in F} \mathrm{Runs}(q_0, p)$ を表すことにする．遷移列 $\rho = ((q_0, a_1, q_1), \ldots, (q_{k-1}, a_{k-1}, q_k)) \in \delta^*$ について，$\pi(\rho)$ で「i 番目の遷移列が現れる回数」を i 次元目の値に持つ d 次元自然数ベクトルを表す：

$$\pi(\rho)_i \stackrel{\mathtt{def}}{=} \text{the number of occurrence of the } i\text{-th transition rule in } \rho \quad (i \in \{1, \ldots, d\})$$

制約オートマトン $\mathcal{C} = (\mathcal{A}, C)$ によって認識される言語 $L(\mathcal{C})$ は，以下のように通常のオートマトンの受理条件に加え，さらに受理遷移列が半線形集合に属するか（遷移規則の個数の制約を満たすか？）どうかを検査することで定義される：

$$L(\mathcal{C}) \stackrel{\mathtt{def}}{=} \{\pi_{A^*}(\rho) \in A^* \mid \rho \in \mathrm{Runs}(q_0, F), \pi(\rho) \in C\}.$$

通常のオートマトンの能力に加え，制約オートマトンでは半線形集合での個数の検査ができ，それによって「個数を数える」様々な言語が認識できるようになっている．例えば，言語 $\mathrm{MIX} \stackrel{\mathtt{def}}{=} \{w \in \{a, b, c\}^* \mid |w|_a = |w|_b = |w|_c\}$（ここで $|w|_a$ は w 中に現れる文字 a の個数を表している）は文脈自由言語ではない例として有名であるが，1 状態の制約オートマトン $\mathcal{C}_1 = ((\{q\}, \delta, q, \{q\}), D_3)$，ここで遷移規則 δ は

$$\delta(q, a) = \delta(q, b) = \delta(q, c) = q$$

を D_3 は 3 次元の対角線をそれぞれ表す，で認識することができる：定義より $L(\mathcal{C}_1) = \mathrm{MIX}$ となる．

より例として，定義は複雑になるものの，うまく制約オートマトンを構成することで非二乗語全体の集合 $A^* \setminus \{ww \mid w \in A^*\}$ も制約オートマトンで認識できることが知られている ([1]).

3. 実装

repa[*1]は筆者の実装した正規表現エンジンである．これは，有限オートマトンを用いる通常の正規表現エンジンとは異なり，制約オートマトンを用いて**数を数えることができる**.

本章では制約オートマトンを用いた「数を数えることのできる正規表現エンジン」である repa の実装と工夫について説明する．実装によりいくつかの考察や課題が生じたので，それらについても説明する.

3.1 制約付き正規表現

制約オートマトンを用いた正規表現エンジンを実装するために，制約付き正規表現を考えた．アルファベット A 上の**制約付き正規表現**は制約付き正規表現の式 $\mathcal{E}$ と変数の集合 V 上の Presburger 算術の制約式 $\mathcal{C}$ の組 $(\mathcal{E}, \mathcal{C})$ で表される．**制約付き正規表現の式**は次の構文で表される.

$$\mathcal{E}, \mathcal{F} \coloneqq \mathcal{E} \cdot \mathcal{F} \mid \mathcal{E} | \mathcal{F} \mid \mathcal{E}^* \mid a \mid \{x\}$$

$\mathcal{E} \cdot \mathcal{F}$，$\mathcal{E}|\mathcal{F}$，$\mathcal{E}^*$ は通常の連接・選択・Kleene 閉包であり，$a \in A$ は文字リテラルである．$x \in V$ について $\{x\}$ は制約付き正規表現に固有の式で，文字列としては空文字列だが，変数 x をインクリメントすることを表す．この変数は制約式 $\mathcal{C}$ で参照する.

次の例は $\{a^n b^n c^n | n \geq 0\}$ を表す制約付き正規表現である.

$$((a\{x\})^*(b\{y\})^*(c\{z\})^*, x = y = z)$$

a, b, c のあとの $\{x\}, \{y\}, \{z\}$ で各文字の出現回数を数えており，Presburger 算術の式でそれらが等しいことを確認する.

[*1] `https://github.com/makenowjust-labs/repa` でソースコードを公開している.

表 1 repa のマッチング VM の基本的な命令
(下線は一般的なマッチング VM と異なる点)

命令	説明
`FORK` ℓ	ラベル ℓ と現在の環境をスタックにプッシュし，次のバックトラック位置とする
`JUMP` ℓ	ラベル ℓ にジャンプする
`LITERAL` a	文字 a にマッチする
`MATCH`	現在の環境が制約式を満たしている場合，マッチングを成功とする.
`INC` x	変数 x をインクリメントする

制約付き正規表現は 1 つの変数を複数の箇所でインクリメントすることができ，状態遷移規則の回数を数える制約オートマトンとはやや扱いが異なる．しかし，制約式の変数を対応する状態遷移規則で適切に置き換えることで，等しい制約オートマトンへと変換できる.

3.2 repa

repa は制約オートマトンを用いた「数を数えることのできる正規表現エンジン」の実装である．VM 型の正規表現エンジンで，正規表現をマッチング VM の命令列に変換し，バックトラックにより非決定性有限状態オートマトンの振る舞いを模倣することで，正規表現マッチングを実現する.

VM 型の正規表現エンジンの基本的な命令は，`FORK` ℓ, `JUMP` ℓ, `LITERAL` a, `MATCH` の 3 種類である (表 1)．`FORK` ℓ は正規表現の選択を表現でき，`JUMP` ℓ は `FORK` ℓ と組み合わせて Kleene 閉包が表現できる．そして，`LITERAL` a により文字 a が表現できるため，正規表現を一通り扱うことができる.

repa ではこの命令に加えて `INC` x を持っている．これは変数 x をインクリメントする命令で，$\{x\}$ が表現できる．`FORK` ℓ でバックトラックのためのスタックに分岐先のラベルだけでなく，現在の環境もスタックにプッシュすることで，変数を含めたバックトラックを正しく行なえるようにしている．`MATCH` では単にマッチングを成功とするのではなく，現在の環境が制約式を満たしているか確認して，満たしている場合のみマッチング成功とする.

これらの変更により，VM 型の正規表現実装を拡張で制約付き正規表現のマッチングを実現できた．

3.3 考察と課題

3.3.1 空ループの扱い

通常の VM 型の正規表現では，空文字列にマッチするループがあった場合，無限ループを防ぐためにマッチング失敗とする．通常の正規表現であれば空文字列の Kleene 閉包は空文字列にマッチするだけなので問題はないが，制約付き正規表現の場合，空文字列の Kleene 閉包が変数のインクリメントを含んでいる可能性がある．例えば $(\{x\}^*, x > 0)$ は $\{x\}$ を任意の回数繰り返すことができるため制約式を満たせるのだが，空ループをマッチング失敗としてしまうと制約式を満たさなくなってしまう．

repa は従来の VM 型の実装に従って，空ループをマッチング失敗としているため，この問題がある．この問題を解決するためには，現在の環境に保持する変数の値を，個々の変数について単一の値で持つのではなく，全体で半線型集合として持つ必要があると考えられる．その場合，空ループを予め分析して，その空ループで生じるインクリメントを半線型集合として計算する必要がある．

3.3.2 DFA 型のマッチング

正規表現マッチングを VM 型で行う場合，正規表現によってはマッチング時間が爆発して，ReDoS 脆弱性を引き起こす可能性がある．repa も同様に，ReDoS 脆弱性を引き起こすようなマッチング時間の爆発が起こる可能性がある．

ReDoS を防ぐためには正規表現マッチングで決定性有限オートマトンの振る舞いを模倣する DFA 型のマッチングが有効である．制約付き正規表現のマッチングを DFA 型のようにする場合，遷移を状態と環境を組の集合に対して行なうことになる．このとき，環境は前述したように半線型集合として表現した方が空ループに対する扱いなどで困らないだろう．

今後の課題として，DFA 型の正規表現エンジンの実装も行い，VM 型との性能評価を行いたい．

4. まとめ

本研究では制約オートマトンを VM 型の正規表現エンジンの拡張で実装し，実装の課題についていくつか考察を行った．ソースコードは github にて公開しており，誰でも利用できる．今後の課題として制約オートマトンを DFA 型の正規表現エンジンでの実装が挙げられるが，制約オートマトンにおいては決定性と非決定性で真に能力が異なることから，理論的にも難しい点が複数あると思われる．

謝辞 本研究は JSPS KAKENHI Grant Number JP19K14582 および JST ACT-X Grant Number JPMJAX210B の助成を受けている．

参考文献

[1] Michaël Cadilhac, Alain Finkel, Pierre McKenzie: *Unambiguous Constrained Automata*. LNCS Vol. 7410, pp.239–250, Springer, 2012.