ビット並列手法を用いた

正規表現照合アルゴリズムの実証実験

Implementation and Evaluation of a Bit-parallel Algorithm

for Regular Expression Matching

光吉　健汰

Kenta Mitsuyoshi

令和**2**年**2**月

北海道大学工学部 情報エレクトロニクス学科
情報理工学コース
情報知識ネットワーク研究室

要 旨

本論文では，正規表現照合問題に取り組む．本研究では，2010 年に Kanera らが提案したビット並列手法を用いたアルゴリズムに対し，CPU が提供している SIMD 命令を導入する．また，SIMD 命令を導入した実装とそうでない実装で比較実験を行う．比較実験の結果，既存アルゴリズムと提案アルゴリズムでは実行時間に差はみられなかった．しかしながら，バックトラック法を用いた python ライブラリでは実行時間が指数的に増加するパターンにおいても，Kaneta らのアルゴリズムでは実行時間の増加が線形に抑えられたことを確認した．

目次

第1章

はじめに

1.1 背景と目的

文字列集合を表現する方法の一つとして正規表現がある．これは，文字と演算子から表現され，正規表現に再帰的に演算を作用することで，柔軟に文字列集合を表現している．正規表現の歴史は古く，1951 年に Kleene[1] が考案し，今日まで利用されている．

正規表現照合問題とは，正規表現で与えられた文字列集合の要素が文字列中に出現するかを判定する問題である．既存のアルゴリズムとしては，バックトラック法や正規表現と同じ文字列集合を受理する，等価なオートマトンを模倣する手法がある．オートマトンを模倣するアルゴリズムを高速化する手法の一つに，遷移のテーブル表現を用いた手法やビット並列手法がある．

Kaneta ら [2] は，ビット並列手法を用いた正規表現照合アルゴリズムを提案している．これは，パターンから非決定性オートマトンを構築し，オートマトンの遷移の模倣をビット演算によって高速化している．Kaneta らは，このアルゴリズムを FPGA 上で実装している．

本研究では，Kaneta らのアルゴリズムに対し，CPU 上で動作する，SIMD 命令を導入する．近年の CPU では長いレジスタ長を持ち，このレジスタに対応した SIMD 命令を持つ．この SIMD 命令を用いることによって，ビット演算の回数が少なくなり高速化が期待できる．また，SIMD 命令を導入した実装と導入していない実装の性能を比較する．比較には，正規表現とその部分クラスである拡張文字列パターン，ネットワーク表現，拡張ネットワーク表現のパターンを用いる．また，バックトラック法を用いた Python の正

規表現照合ライブラリとの比較を行う．

1.2 主結果

SIMD 命令を導入した実装とそうでない実装で，実行時間，前処理時間の差はみられなかった．これらの実装は，入力テキスト長に対して理論通り線形時間で動作していることが確認できた．NFA の状態数を変化させた場合でも大きな差はみられなかった．また，バックトラック法を用いたライブラリでは実行時間が指数的に増加するパターンにおいても，Kaneta らのアルゴリズムでは実行時間が線形的な増加に抑えられたことを確認した．

1.3 関連研究

Glushkov[3] は，ϵ 遷移を用いない非決定性オートマトンの構築法を提案し，正規表現照合アルゴリズムに導入している．Glushkov の構築法は，正規表現のパターン長 m に対して，NFA の状態数が $m+1$，遷移の数が $\mathcal{O}\left(m^2\right)$ である NFA の構築法を与えている．また，Thompson[4] は，ϵ 遷移を導入した非決定性オートマトンを用いた正規表現照合アルゴリズムを提案している．Thompson の構築法は，正規表現のパターン長 m に対して，NFA の状態数が高々 $2m$，遷移の数が高々 $4m$ である NFA の構築法を与えている．この二つのアルゴリズムはどちらも NFA の構築法を提案している．

Baeza-Yates ら [5] が，パターン照合問題にビット並列手法を導入した研究として，Shift-AND 法を提案している．これは，NFA の状態をビット列で表し，遷移の模倣をビット演算によって行うことでパターン照合を高速に行うアルゴリズムである．また，Navarro[6] は，Shift-AND 法を拡張文字列パターンに適用している．Kaneta ら [2] は，Shift-AND 法に加え，ϵ 遷移をビット並列手法を用いて計算することで，拡張ネットワーク表現の効率的な照合アルゴリズムを提案している．

ビット並列手法を用いた他の正規表現照合アルゴリズムとしては，NFA の遷移をテーブルで持つアルゴリズムがある．このアルゴリズムは，テーブルへのアクセスに，状態集合を表すビットマスクを添字として用いる．テーブルには，添字のビットマスクに対

応した遷移後のビットマスクを格納している．これにより，遷移の計算が定数時間で可能にしている．Wu ら [7] は，Thompson の構築法によって作られた NFA に対してテーブルを作成したアルゴリズムを提案している．Navarro ら [8, 9, 10] は，Glushkov の構築法によって作られた NFA に対してテーブルを作成したアルゴリズムを提案している．

1.4 本論文の構成

2 章で本論文で用いる，用語と記法，問題の定義をする．3 章では，本研究で用いる Kaneta ら [2] のアルゴリズムを説明する．4 章では，3 章のアルゴリズムに対し，SIMD 命令を導入する手法を説明する．5 章では，実装したアルゴリズムを実装し，実行時間を比較する実験について述べる．6 章では，本研究のまとめを述べる．

第2章

準備

次の 2.1 節では，本論文で用いる記号と用語の説明を行う．2.2 節では，正規表現の定義と，その部分クラスについて述べる．2.3 節では，非決定性オートマトンの定義を行う．2.4 節では，正規表現照合問題の定義を行う．2.5 節では，SIMD 命令について述べる．

2.1 記号，用語の定義

2.1.1 文字列

文字を要素とする空でない有限集合 Σ をアルファベットと呼ぶ．アルファベット Σ について，Σ の要素数を $|\Sigma|$ で表す．アルファベット Σ の要素を 0 個以上並べたものを Σ 上の文字列と呼ぶ．Σ 上の文字列全体の集合を Σ^* で表す．n 個の文字 $a_1, a_2, \ldots, a_n \in \Sigma$ を並べた文字列を $a_1 a_2 \ldots a_n$ と表す．また，文字列の文字の個数を文字列の長さという．長さが 0 の文字列を空文字列と呼び，Σ に含まれない特別な記号 ϵ を用いて表記する．長さ n の文字列 $A = a_1 a_2 \ldots a_n$ に対して，$a_i \ldots a_j \, (1 \leq i \leq j \leq n)$ を A の部分文字列といい，$A[i, j]$ と表す．文字列 A，B に対して，A の末尾に B を連結してできる文字列を $A \cdot B$，または AB で表し，これを A と B の連接という．文字列 A を n 個並べてできる文字列を A^n と表す．例として，アルファベット $\Sigma = \{a, b, c\}$ 上の文字列 $A = abb$，$B = c$ について，$A^2 B$ は，$A^2 B = AAB = abbabbc$ であり，部分文字列 $A[1, 2] = ab$ である．

Σ^* の部分集合 L を言語という．言語 L_1，L_2 の連接を $L_1 \cdot L_2$，または $L_1 L_2$ で表し，

$$L_1 \cdot L_2 = \{AB \mid A \in L_1 \text{かつ}, B \in L_2\} \tag{2.1}$$

と定義する．例として，アルファベット $\Sigma = \{a, b, c\}$ 上の言語 $L_1 = \{a, ab\}$，$L_2 = \{b, ca\}$

について，L_1 と L_2 の連接 L_1L_2 は，$L_1L_2 = \{ab, abb, abca, aca\}$ である．また，言語 L 中の要素を繰り返して得られる文字列集合として，

$$L^0 = \{\epsilon\}, \tag{2.2}$$

$$L^{n+1} = L^n L \, (n \geq 0), \tag{2.3}$$

$$L^* = \bigcup_{n=1}^{\infty} L^n, \qquad (L \text{ の Kleene 閉包}) \tag{2.4}$$

をそれぞれ定義する．

2.1.2 ビットマスク

アルファベット $\Sigma = \{0, 1\}$ 上の長さ n の文字列 $D \in \Sigma^n$ を長さ n のビットマスクとする．長さ n のビットマスク D は簡単のため，添字を逆順にした $D = d_{n-1} \ldots d_0$ のように表される．長さが等しいビットマスクに対して，以下の演算を定義する．

- $(x\&y)$ x と y のビット毎の論理積（AND）をとる．
- $(x \mid y)$ x と y のビット毎の論理和（OR）をとる．
- $(x \oplus y)$ x と y のビット毎の排他的論理和（XOR）をとる．
- $(\sim x)$ x のビット毎の否定（NOT）をとる．
- $(x \gg y)$ x の左 y ビット論理シフトをとる．
- $(x \ll y)$ x の右 y ビット論理シフトをとる．
- $(x + y)$ x と y の論理加算をとる．
- $(x - y)$ x と y の論理減算をとる．

例として，長さ 8 のビットマスク $D_1 = 00011101$，$D_2 = 11000011$ について，$D_1\&D_2 = 00000001$，$\sim D_1 = 00111100$，$D_1 - D_2 = 01011010$ である．

2.1.3 ワードマシンモデル

ワードマシンモデル RAM とは，W ビット長のビット演算 $(|, \&, \oplus, \sim, \ll, \gg)$ を定数時間で実行できる計算機モデルである．これらの演算に加えて，W ビット長の加減算 $(+, -)$ が定数時間で実行可能である計算機モデルを $RAM(+)$ とする．本研究では，$RAM(+)$ 上でのアルゴリズムについて考察する．

2.2 正規表現

本節では，正規表現と，その部分クラスである拡張文字列パターン [11]，ネットワーク表現 [12]，拡張ネットワーク表現 [2] を定義する．

2.2.1 定義

アルファベット Σ 上の正規表現（Regular Expression）R とは，文字集合 $\Sigma \cup \{\epsilon, |, \cdot, *, (,)\}$ 上の文字列であり，以下のように再帰的に定義される．

空文字　ϵ は正規表現である．

文字　$\alpha \in \Sigma$ のとき，α は正規表現である．

連接　R_1, R_2 が正規表現であるとき，$(R_1 \cdot R_2)$ は正規表現である．

選言　R_1, R_2 が正規表現であるとき，$(R_1 \mid R_2)$ は正規表現である．

Kleene 閉包　R が正規表現であるとき，$(R)*$ は正規表現である．

$(R_1 \cdot R_2)$，$(R_1 \mid R_2)$ は結合則 $((R_1 \cdot R_2) \cdot R_3) = (R_1 \cdot (R_2 \cdot R_3))$，$((R_1 \mid R_2) \mid R_3) = (R_1 \mid (R_2 \mid R_3))$ が成り立つことから，括弧を省略し，それぞれ，$(R_1 \cdot R_2 \cdot R_3)$，$(R_1 \mid R_2 \mid R_3)$ と表す．また，演算子の結合の強さを “$*$” $>$ “$\cdot$” $>$ “$|$” とすることから，不要な括弧を適宜省略できる．正規表現 R の長さ $|R|$ を，R に出現する文字，空文字，演算子 “$\cdot$”，“$|$”，“$*$” の数とする．

正規表現の言語 $L(R) \subset \Sigma^*$ は以下のように定義される．

1. $L(\epsilon) = \varnothing$.

2. $\alpha \in \Sigma$ について，$L(\alpha) = \{\alpha\}$.

3. 正規表現 R_1，R_2 について，$L(R_1 \cdot R_2) = L(R_1) \cdot L(R_2)$.

4. 正規表現 R_1，R_2 について，$L(R_1 \mid R_2) = L(R_1) \cup L(R_2)$.

5. 正規表現 R について，$L(R*) = L(R)^*$.

例として，$\Sigma = \{a, b, c\}$ 上の正規表現 $R = (ab|ac)ca*$ としたとき，$L(R) = \{abc, acc\} \cdot \{a\}^*$ となる.

2.2.2 拡張文字列パターン

Σ 上の正規表現 $R = r_1 r_2 \ldots r_m \, (m \geq 0)$ が**拡張文字列パターン（Extended String Pattern）**[11] であるとは，$r_i \, (1 \leq i \leq m)$ が以下のいずれかであることをいう.

文字	$r_i = \alpha \, (\alpha \in \Sigma)$ である.
ワイルドカード	$r_i = .$ である.
オプショナル	$r_i = \alpha? \, (\alpha \in \Sigma \cup \{.\})$ である.
制限付き繰り返し	$r_i = \alpha\{x, y\} \, (\alpha \in \Sigma \cup \{.\}, x, y \in \mathbb{N}, x < y)$ である. このとき，$L(\alpha\{x, y\}) = L\left((\alpha?)^{y-x} \alpha^x\right)$ である.
繰り返し	$r_i = \alpha*$，または，$\alpha+$ である．このとき，$L(\alpha+) = L(\alpha\alpha*)$ である.

拡張文字列パターンは，正規表現と異なり，演算子 “$|$”，“$*$” が用いられない（上の定義で使用されている演算子 “$*$” は，文字の繰り返しを意味しており，Kleene 閉包を表現できない）．例として，$\Sigma = \{a, b, c\}$ としたとき，$ab?a\{2, 4\}c+$ は拡張文字列パターンであり，$L(ab?a\{2, 3\}c+) = \{aaac, aaaac, abaac, abaaac\} \cdot \{c\}^*$ である.

2.2.3 ネットワーク表現

ネットワーク表現（Network Expression）[12] は正規表現であり，以下の様に再帰的に定義される.

1. 空文字列 ϵ はネットワーク表現である．

2. $\alpha \in \Sigma$ のとき，α はネットワーク表現である．

3. R_1，R_2 がネットワーク表現であるとき，R_1R_2 はネットワーク表現である．

4. R_1，R_2 がネットワーク表現であるとき，$R_1 \mid R_2$ はネットワーク表現である．

ネットワーク表現が表す言語は以下のように定義される．

1. $L(\epsilon) = \{\epsilon\}$.

2. $\alpha \in \Sigma$ であるとき，$L(a) = \{\alpha\}$.

3. R_1，R_2 がネットワーク表現であるとき，$L(R_1 \cdot R_2) = L(R_1) \cdot L(R_2)$.

4. R_1，R_2 がネットワーク表現であるとき，$L(R_1 \mid R_2) = L(R_1) \cup L(R_2)$.

ネットワーク表現は正規表現のうち，Kleene 閉包を除いた表現といえる．例として，正規表現 $ab(ac|bc)c$ はネットワーク表現であるが，正規表現 $ab(ac|bc)*c$ は Kleene 閉包が含まれているため，ネットワーク表現でない．

2.2.4 拡張ネットワーク表現

拡張ネットワーク表現（Extended Network Expression）[2] は正規表現であり，ネットワーク表現に拡張文字列パターンを導入した表現であり，以下のように定義される．

1. R が拡張文字列パターンであるとき，R は拡張ネットワーク表現である．

2. R がネットワーク表現であるとき，R は拡張ネットワーク表現である．

3. R_1, R_2 が拡張ネットワーク表現であるとき，$R_1 \cdot R_2$ は拡張ネットワーク表現である．

4. R_1, R_2 が拡張ネットワーク表現であるとき，$R_1 \mid R_2$ は拡張ネットワーク表現である．

例として，$ab(ac?|cc*)d?$ は拡張ネットワーク表現であるが，$ab(ac?|cc*)*d?$ は Kleene 閉包を含んでいるため，拡張ネットワーク表現でない．

2.2.5 各表現の階層

正規表現，拡張文字列パターン，ネットワーク表現，拡張ネットワーク表現の言語全体の集合をそれぞれ，正規表現クラス $\mathcal{REG}$，拡張文字列パターンクラス $\mathcal{EXT}$，ネットワーク表現クラス $\mathcal{NET}$，拡張ネットワーク表現クラス $\mathcal{EXNET}$ とする．これらのクラスは包含関係として図 2.1 のようになっている．

2.3 非決定性有限オートマトン

非決定性有限オートマトン（NFA）N とは 5 項組 $N = (Q, \Sigma, \sigma, q_0, F)$ で定義され，それぞれの項は以下のように定義される．

Q: 空でない有限集合

Σ: アルファベット

σ: $Q \times (\Sigma \cup \{\epsilon\})$ から状態の冪集合 $P(Q)$ への遷移関数

q_0: Q の要素

F: Q の部分集合

Q の要素を状態，q_0 を開始状態，F の要素を受理状態という．
遷移関数 σ を拡張した関数 σ^* を，状態と文字列集合の組 $(Q \times \Sigma^*)$ から状態の冪集合 2^Q への遷移関数し，(2.5) で定義する．

$$\sigma^*(q, \epsilon) = \{q\} \qquad (q \in Q) \tag{2.5}$$

$$\sigma^*(q, aA) = \bigcup_{s \in \sigma(q,a)} \sigma^*(s, A) \qquad (q \in Q, A \in \Sigma^*) \tag{2.6}$$

また，$\phi \in \sigma(\theta, \alpha)\,(\theta \in Q, \alpha \in \Sigma \cup \{\epsilon\})$ となる状態 ϕ が存在するとき，(θ, α, ϕ) と表す．
NFA N が文字列 A を受理するとは，$\sigma^*(q_0, A) \cap T \neq \varnothing$ であることをいう．また，NFA N が言語 L の任意の要素 a を受理するとき，N は言語 L を受理するという．例として，アルファベット $\Sigma = \{a, b, c\}$，遷移関数を表 2.1 とした NFA（図 2.2）は，言語 $\{a\} \cdot \{a, b\}^* \cdot \{b\}$ を受理する．

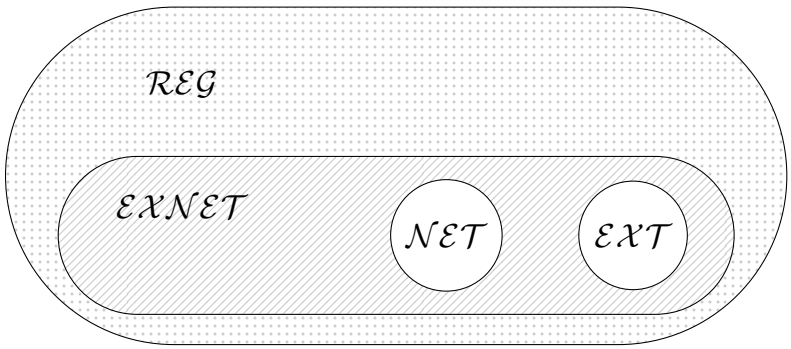


図 2.1: 各表現クラスの包含関係を表した図．各部分クラスが階層構造になっていることが確認できる．

表 2.1: NFA の遷移関数の例．Q と文字から集合の各要素へ遷移する．

Q	A	B	C
1	$\{2\}$	$\varnothing$	$\varnothing$
2	$\{2\}$	$\{2,3\}$	$\varnothing$
3	$\varnothing$	$\varnothing$	$\varnothing$

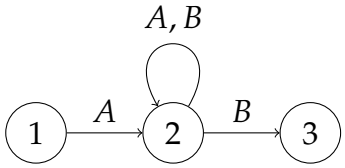


図 2.2: 表 2.1 を遷移関数とした NFA の状態遷移図．状態 2 から文字 B による遷移が複数あることが確認できる．

σ のうち，文字を受け取って遷移するものを α 遷移とし，空文字を受け取って遷移するものを ϵ 遷移とする．

2.4 正規表現照合問題

本研究で扱う問題である，正規表現照合問題を定義する．長さ $n \geq 0$ のテキストを，文字列 $T = t_1 \ldots t_n$ とし，パターン R を正規表現とする．パターン R がテキスト T 中の位置 q で出現するとは，$1 \leq p \leq q$ について，$T[p \ldots q] \in L(R)$ が成り立つことをいう．正規表現照合問題とは，T と R を受け取り，R が T 中に出現する全ての位置 q を出力する問題である．

正規表現照合問題は，NFA が受理する T 中の部分文字列の位置を全て出力する問題に置き換えることができる．これは，正規表現が表す言語を受理する NFA が存在するためである．

2.5 SIMD 命令

SIMD（Single Instruction Multiple Data）とは，複数のデータに対し，単一の命令を適用することで処理の並列化を行う手法である．SIMD 命令とは，ハードウェア上で SIMD 処理を実行する命令である．SIMD 命令では，ワードサイズを超えるレジスタに複数のデータをロードし，単一の命令を同時に演算する．例として，4 個の 64 ビット整数型について，それぞれ加算を行う__mm256_add_epi64(__m256i a, __m256i b) の動作を図 2.3 に示す．SIMD 命令を用いない場合は，それぞれのデータを個別に演算する必要があるため，SIMD 命令を用いることで全体としての演算回数が少なくなる．本研究では，CPU 上での実装のため，CPU 拡張命令セットである，Intel AVX2 命令セット[1]を用いる．これは，256 ビットレジスタでの整数演算命令をサポートしている．

[1]Intel® Architecture Instruction Set Extensions and Future Features Programming Reference
https://software.intel.com/sites/default/\\files/managed/c5/15/
architecture-instruction-set-extensions-programming-reference.pdf

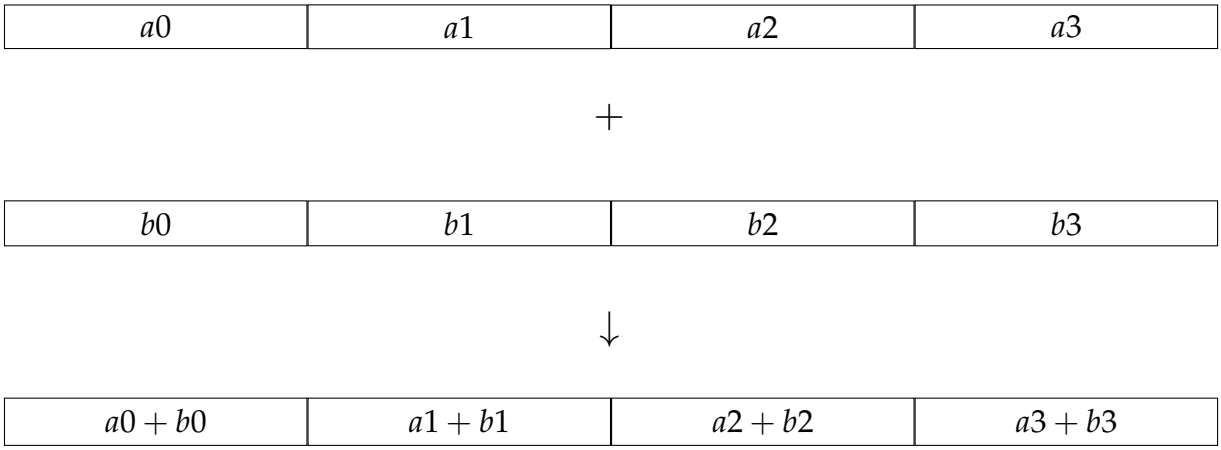


図 2.3: SIMD 命令 __mm256_add_epi64(__m256i a, __m256i b) の動作例．64 ビットの整数型のデータに対して，それぞれ独立に加算が行われる．

第3章

Kanetaらの正規表現正規表現照合アルゴリズム

本章では，Kanetaら [2] が提案した，ビット並列手法による正規表現照合アルゴリズムの説明をする．はじめに，前処理で構築するトンプソンオートマトンの説明を行い，次に，正規表現照合のアルゴリズムを説明する．

3.1 トンプソンオートマトン

3.1.1 正規表現からの構築手順

トンプソンオートマトン（TNFA）$TNFA(R)$ を [4] に基づき定義する．与えられた正規表現 R から以下の手順で構築される NFA を，トンプソンオートマトン $TNFA(R)$ という．正規表現 $R_i\,(1 \leq i \leq n)$ について，$TNFA(R_i) = (Q_i, \Sigma_i, \sigma_i, \theta_i, \{\phi_i\})\,(0 \leq i \leq n)$ が構築されているとして，

1. $R = \epsilon$ ならば，$TNFA(R) = (\{\theta, \phi\}, \Sigma, \{(\theta, \epsilon, \phi)\}, \theta, \{\phi\})$.
2. $R = \alpha\,(\alpha \in \Sigma)$ ならば，$TNFA(R) = (\{\theta, \phi\}, \Sigma, \{(\theta, \epsilon, \phi)\}, \theta, \{\phi\})$.
3. $R = (R_1 \cdot \cdots \cdot R_n)$ ならば，$\phi_1 = \theta_2, \ldots, \phi_{n-1} = \theta_n$ となるように $N(R_i)$ を連結した TNFA $N(R)$ を得る．
4. $R = (R_1 \mid \cdots \mid R_n)$ ならば，状態集合に開始状態 θ，受理状態 ϕ を追加し，遷移として $\{(\theta, \epsilon, \theta_1), \ldots, (\theta_n, \epsilon, \phi), (\phi_1, \epsilon, \phi), \ldots, (\phi_n, \epsilon, \phi)\}$ を追加した TNFA $N(R)$ を得る．

5. $R = (R_1)^*$ ならば，状態集合に開始状態 θ，受理状態 ϕ を追加し，遷移として $\{(\theta,\epsilon,\theta_1),(\phi_1,\epsilon,\phi),(\theta,\epsilon,\phi),(\phi_1,\epsilon,\theta_1)\}$ を追加した TNFA $N(R)$ を得る．

$N(R_1 \cdot R_2)$ について，Thompson は手順 3. を用いず，遷移 $(\phi_1,\epsilon,\theta_2)$ を追加することで，TNFA を構築していた．本研究では状態数を小さくするために，手順 3. の構築を行う．また，選言の構築時に追加される遷移 $(\theta,\epsilon,\theta_i)$ を分配遷移（Scatter Transition），(ϕ_i,ϵ,ϕ) を集約遷移（Gather Transition），Kleene 閉包の構築時に追加される遷移 $(\phi_1,\epsilon,\theta_1)$ を逆伝搬遷移（Backedge Transition），ϵ 遷移のうち逆伝搬遷移でない遷移を伝搬遷移（Propagate Transition）という．それぞれの構築手順によって得られる TNFA を図 3.1 に示す．

3.1.2 拡張文字列パターンからの構築手順

拡張文字列パターン $R = r_1 r_2 \ldots r_m\ (m \geq 0)$ を受け取ったとき，以下の手順でパターンの要素 r_i を構築し，前述した連接の構築手順によって R から $TNFA$ を構築する．

1. $r_i = \alpha\ (\alpha \in \Sigma)$ ならば，正規表現での構築手順と同様の手順で構築をする．
2. $r_i = .$ ならば，$TNFA(R) = (\{\theta,\phi\},\Sigma,\{(\theta,\Sigma,\phi)\},\theta,\phi)$ を構築する．ただし，遷移 (θ,Σ,ϕ) は，アルファベット Σ の任意の文字を受け取ったとき，θ から ϕ への遷移を表す．
3. $r_i = \alpha?$ ならば，$TNFA(R) = (\{\theta,\phi\},\Sigma,\{(\theta,\alpha,\phi),(\theta,\epsilon,\phi)\},\theta,\phi)$ を構築する．
4. $r_i = \alpha\{x,y\}\ (\alpha \in \Sigma\{.\}, x,y \in \mathbb{N})$ ならば，$\alpha?$ を $y-x$ 個，α を x 個並べた $(\alpha?)^{y-x}\alpha^x$ を上の手順で構築する．
5. $r_i = \alpha*$ ならば，$TNFA(R) = (\{\theta,\phi\},\Sigma,\{(\theta,\alpha,\phi),(\theta,\epsilon,\phi),(\phi,\alpha,\phi)\},\theta,\{\phi\})$ を構築する．
6. $r_i = \alpha+$ ならば，$\alpha\alpha*$ とみなし，上の手順で構築する．

それぞれの構築手順によって得られる TNFA を図 3.2 に示す．

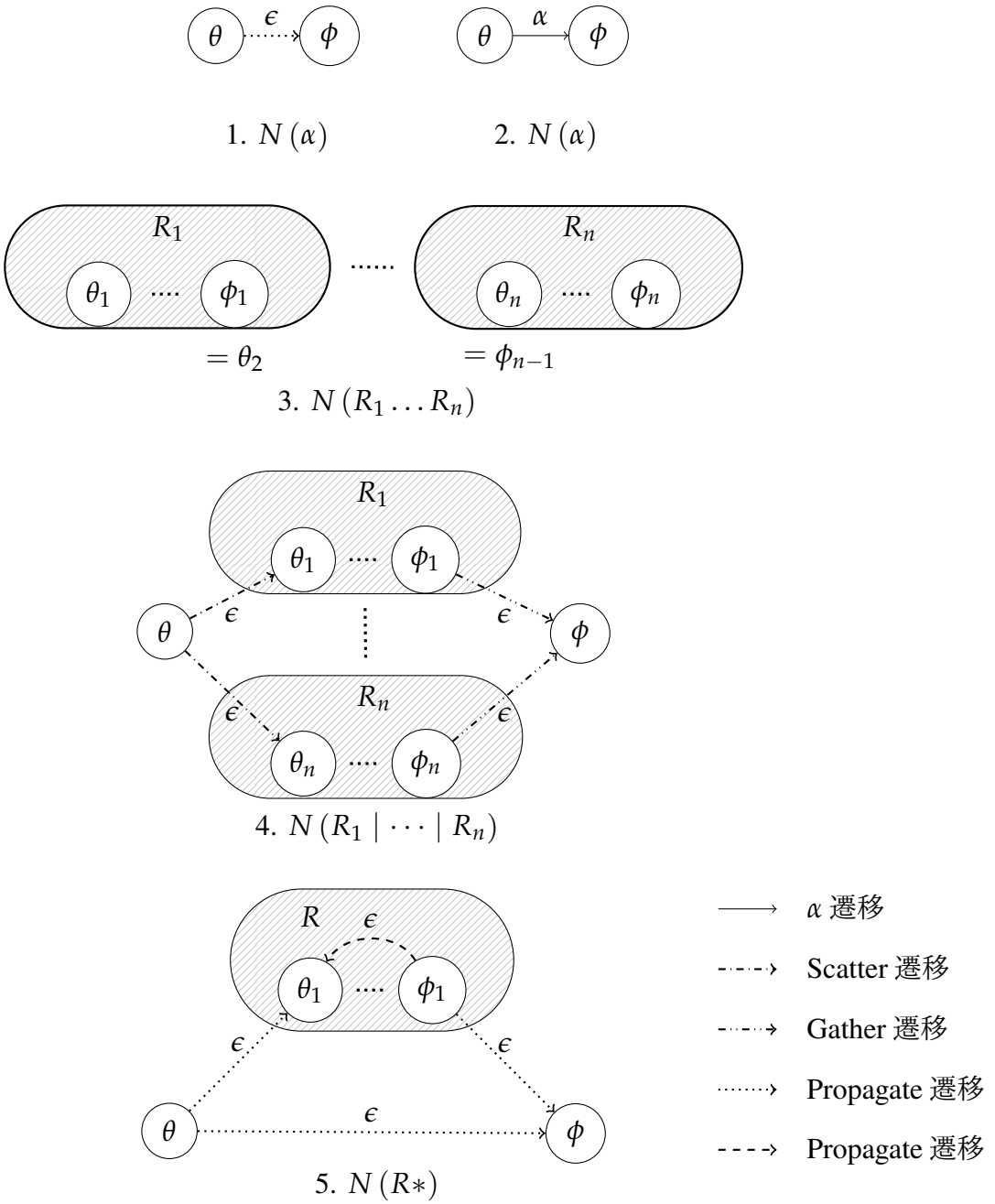
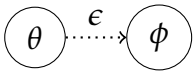
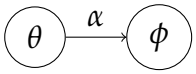


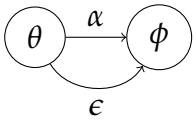
図 3.1: 正規表現に対する TNFA の構築手順．円が状態，矢印が遷移を表す．3.4.5. の斜線部は，正規表現からすでに構築された TNFA であり，θ_i が開始状態，ϕ_i が受理状態を表す．



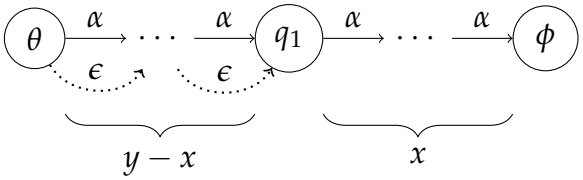
1. $N(\alpha)$



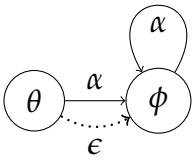
2. $N(.)$



3. $N(\alpha?)$



4. $N(\alpha\{x,y\})$



5. $N(\alpha*)$

図 3.2: 拡張文字列パターンに対する TNFA の構築手順．4. は α 遷移と ϵ 遷移が $y-x$ 個，α 遷移が x 個繋がっていることを表す．ϵ 遷移は，全て伝搬遷移である．

ネットワーク表現は正規表現のうち，Kleene 閉包の演算を除いて得られる表現なので，正規表現の手順に従って TNFA を構築する．

拡張ネットワーク表現は拡張文字列パターンとネットワーク表現の構築手順に従って TNFA を構築する．

3.2 Kaneta らのビット並列手法

本節では，前処理で構築したビットマスクを用いた，正規表現照合アルゴリズムを説明する．

3.2.1 ビット並列手法で用いる記号・記法の定義

パターンから得られた TNFA より，各状態の深さを定義する．TNFA の状態 q の深さ $d(q)$ とは，開始状態 θ から q へのパス中に存在する分配遷移の数 $s(q)$ と集約遷移の数 $g(q)$ の差で定義され，(3.1) で求められる．

$$d(q) = s(q) - g(q) \tag{3.1}$$

また，遷移の深さ $d(\theta, \epsilon, \phi)$ を (3.2) のように定義する．

$$d(\theta, \epsilon, \phi) = \begin{cases} d(\theta) & (\text{遷移が分配遷移}) \\ d(\phi) & (\text{遷移が集約遷移}) \\ d(\theta) & (otherwise) \end{cases} \tag{3.2}$$

次に，深さ d の ϵ ブロック $B(d)$ を (3.3) で定義する．深さ d の ϵ ブロックは 0 個以上存在する．

$$B(d) = \{q \in Q \mid \epsilon \text{ 遷移のみで構成された連結成分のうち，深さが } d \text{ のもの}\} \tag{3.3}$$

3.2.2 本アルゴリズムの概要

本アルゴリズムでは，前処理として，パターン P から TNFA を構築し，得られた TNFA からビットマスクを構築する．実行時には，入力テキストを一字ずつ読み取り，TNFA の

遷移を模倣する．このアルゴリズムを Algorithm 1 に示す．$\mathrm{Move}(D, \alpha)$ では，α 遷移の模倣を行う．$\mathrm{EpsClo}_N(D)$ では，ϵ 遷移の模倣を行う．これを Algorithm 2 に示す．ϵ 遷移の模倣は，分配遷移，集約遷移，伝搬遷移，逆伝搬遷移に分けて，深さごとに行う．

Algorithm 1 実行時の TNFA の模倣を行うアルゴリズム

1: **function** RunTNFA(T, N)

Input:

T: 入力テキスト

$N(R)$: パターン R を受理する TNFA

2: $Ans \leftarrow \varnothing$ ▷ パターンの出現位置を格納する.

3: $D \leftarrow 0^{m-1}1$ ▷ ビットマスクに初期状態をセットする.

4: $D \leftarrow \mathrm{EpsClo}_N(D)$

5: **for** $i \leftarrow 1, \ldots, n$ **do**

6: **if** D & $10^{m-1} \neq 0$ **then**

7: $Ans \leftarrow Ans \cup \{i-1\}$

8: **end if**

9: $D \leftarrow \mathrm{Move}_N(D, T_i)$

10: $D \leftarrow \mathrm{EpsClo}_N(D)$

11: **end for**

12: **return** Ans

13: **end function**

Algorithm 2 ϵ 遷移の模倣を行うアルゴリズム

1: **function** $EpsClo_N(D)$

Input:

D: TNFA $N(R)$ の現在の状態集合を表すビットマスク

2: **for** $i \leftarrow d, \ldots, 1$ **do** ▷ d は $N(R)$ の深さである.

3: $D \leftarrow \text{Propagate}_N(D, i)$

4: $D \leftarrow \text{Backward}_N(D, i)$

5: $D \leftarrow \text{Gather}_N(D, i-1)$

6: **end for**

7: $D \leftarrow \text{Propagate}_N(D, 0)$

8: $D \leftarrow \text{Backward}_N(D, 0)$

9: **for** $i \leftarrow 1, \ldots, n$ **do**

10: $D \leftarrow \text{Scatter}_N(D, i-1)$

11: $D \leftarrow \text{Propagate}_N(D, i)$

12: $D \leftarrow \text{Backward}_N(D, i)$

13: **end for**

14: **return** D

15: **end function**

3.2.3 α 遷移の模倣

α 遷移では，[9] に基づく拡張 SHIFT-AND 法を用いる．拡張 SHIFT-AND 法は，二種類のビットマスク，$CHR[\alpha]$，$REP[\alpha]$ を用いる．$CHR[\alpha]$，$REP[\alpha]$ は以下のようなビットマスクであり，前処理で構築される．

- $CHR[\alpha]$ の i ビット目が 1 であることは，遷移 (j, α, i) $(j \neq i)$ が存在することを表す．
- $REP[\alpha]$ の i ビット目が 1 であることは，遷移 (i, α, i) が存在することを表す．

$\mathrm{Move}_N(D, \alpha)$ では，これらのビットマスクを用いて (3.4) の更新によって α 遷移の模倣を行う．

$$D \leftarrow (D \ll 1) \& CHR[\alpha] \mid D \& REP[\alpha] \mid 1 \tag{3.4}$$

3.2.4 分配遷移の模倣

分配遷移では，三種類のビットマスク，$BLK_S[d]$，$SRC_S[d]$，$DST_S[d]$ を用いる．これらは以下のように定義され，前処理で構築される．

- $BLK_S[d]$ の i ビット目が 1 であることは，遷移先が状態 i である，深さ d の集約遷移が存在することを表す．
- $SRC_S[d]$ の i ビット目が 1 であることは，遷移元が状態 i である，深さ d の分配遷移が存在することを表す．
- $DST_S[d]$ の i ビット目が 1 であることは，遷移先が状態 i である，深さ d の分配遷移が存在することを表す．

$\mathrm{Scatter}_N(D, d)$ では，これらのビットマスクを用いて (3.5) の更新を行い，分配遷移の模倣を実現する．$(D \& SRC_S[d])$ で分配遷移の遷移元の状態を取り出し，$BLK_S[d]$ に減算を行なっている．これは，減算の繰り下げによって，分配遷移と集約遷移に挟まれている状態を 1 にしている．この値から分配遷移の遷移先の状態を取り出すことで分配遷移の模倣を実現している．

$$D \leftarrow D \mid ((BLK_S[d] - (D \& SRC_S[d])) \& DST_S[d]) \tag{3.5}$$

3.2.5 集約遷移の模倣

集約遷移では，三種類のビットマスク，$BLK_G[d]$，$SRC_G[d]$，$DST_G[d]$ を用いて遷移の模倣を行う．これらは以下のように定義され，前処理で構築される．

- $BLK_G[d]$ の i ビット目が 1 であることは，状態 i の深さ $d(i)$ が d 未満であることを表す．
- $SRC_G[d]$ の i ビット目が 1 であることは，遷移元が状態 i である，深さ d の集約遷移が存在することを表す．
- $DST_G[d]$ の i ビット目が 1 であることは，遷移先が状態 i である，深さ d の集約遷移が存在することを表す．

$\mathrm{Scatter}_N(D, d)$ では，これらのビットマスクを用いて (3.6) の更新を行い，集約遷移の模倣を実現する．$(D\&SRC_G[d])$ で集約遷移の遷移元の状態を取り出し，$BLK_G[d]$ に加算を行なっている．これは，加算の繰り上げによって，集約遷移の遷移先の状態を 1 にしている．ただし，遷移元が 1 でなかった場合，遷移先以外の状態が 1 である場合があるので，$DST_G[d]$ とビット AND をとることで遷移先の状態を取り出し，集約遷移の模倣を実現している．

$$D \leftarrow D \mid ((D + (D\&SRC_G[d])) \,\&DST_G[d]) \tag{3.6}$$

3.2.6 伝搬遷移の模倣

伝搬遷移は，ϵ ブロックに含まれている状態のうち，現在アクティブになっている状態から ϵ 遷移のみのパスが三種類のビットマスク，$BLK_P[d]$，$SRC_P[d]$，$DST_P[d]$ を用いて遷移の模倣を行う．これらは以下のように定義され，前処理で構築される．

- $BLK_P[d]$ の i ビット目が 1 であることは，状態 i を含む，深さ $d(i)$ の ϵ ブロックが存在することを表す．
- $SRC_P[d]$ の i ビット目が 1 であることは，状態 i が始端となる，深さ $d(i)$ の ϵ ブロックが存在することを表す．

- $DST_P[d]$ の i ビット目が 1 であることは，状態 i が終端となる，深さ $d\,(i)$ の ϵ ブロックが存在することを表す．

伝搬$_N\,(D,d)$ では，これらのビットマスクを用いて (3.7)，(3.8) の更新を行い，伝搬遷移の模倣を実現する．A は，D のうち，ϵ ブロックに含まれている状態と，各 ϵ ブロックの終端が 1 となったビットマスクとなっている．まず，$(A - SRC_P[d])$ は，各 ϵ ブロックの始端からアクティブな状態の直前までと，アクティブな状態のうち，最も始端に近いものを除いた状態を 1 にする．次に，このビットマスクをビット NOT で反転し，A とビット XOR をとる．これによって，アクティブな状態から ϵ ブロックの終端までの状態が 1 となるので，$BLK_P[d]$ によって ϵ ブロックの状態を取り出して，伝搬遷移の模倣を実現している．

$$A \leftarrow (D\&BLK_P[d]) \mid DST_P[d] \tag{3.7}$$

$$D \leftarrow D \mid (BLK_P[d]\&\,((\sim (A - SRC_P[d])) \oplus A)) \tag{3.8}$$

3.2.7 逆伝搬遷移の模倣

逆伝搬遷移は，他の遷移と異なり，定数回の演算によって遷移を模倣することは困難である．そこで，Kaneta ら [2] はバレルシフター（barrel shifter）と呼ばれるテクニックを用いて，$\mathcal{O}(\log m)$ 回の操作で複数の逆伝搬遷移の模倣を行っている．逆伝搬遷移では，それぞれの遷移元から遷移先へビットシフトによって遷移の模倣をする．このとき，各逆伝搬遷移はビット数が異なるビットシフトが必要となる．そこで，それぞれのビットシフトの距離を二進展開し，$2^k (k \geq 0)$ ビットごとに一度にビットシフトを行う．k は高々 $\mathcal{O}(\log m)$ となるので，ビットシフトは，$\mathcal{O}(\log m)$ 回となる．

逆伝搬遷移では，三種類のビットマスク，$BLK_B[d][k]$，$SRC_B[d]$，$DST_B[d]$ を用いて遷移の模倣を行う．これらのビットマスクは以下のように定義され，前処理で構築される．

- $BLK_B[d][k]$ の i ビット目が 1 であることは，状態 i に対し，深さ d の，$\phi \leq i \leq \theta$ かつ，$(\theta - \phi)$ をビットマスクで表したときの k ビット目が 1 であるような逆伝搬遷移 (θ, ϵ, ϕ) が存在することを表す．

- $SRC_B[d]$ の i ビット目が 1 であることは，遷移元が状態 i となる，深さ d の逆伝搬遷移が存在することを表す．
- $DST_B[d]$ の i ビット目が 1 であることは，遷移先が状態 i となる，深さ d の逆伝搬遷移が存在することを表す．

Backward$_N(D, d)$ では，これらのビットマスクを用いて Algorithm 3 によって逆伝搬遷移の模倣を行う．$(D' \& \sim BLK_B[d][i])$ は，逆伝搬遷移のうち，左 $jump$ ビットシフトを行わないものを取り出す操作となる．$(D' \& BLK_B[d][i])$ は左 $jump$ ビットシフトを行うものを取り出し，これを左 $jump$ ビットシフトする．ビットシフト後に逆伝搬遷移の遷移元よりも前のビット列にビットシフトしたものを消すために $BLK_B[d][i]$ とビット AND をとる．これを，$(D' \& \sim BLK_B[d][i])$ とビット OR をとり，D' に代入する．これを，$\mathcal{O}(\log m)$ 回繰り返し，$(D' \& DST_B[d])$ を D に代入する．これにより，逆伝搬遷移の模倣を実現する．

3.2.8 計算量

Kaneta らのアルゴリズムでは，各クラスでの前処理，正規表現照合の計算量は表 3.1 の通りである．ただし，入力されたパターン P の長さを m，P 中の最大深さを d，入力されたテキスト長を n，アルファベットサイズを $|\Sigma|$，ワードサイズを w とする．

前処理では，$\mathcal{O}(m)$ ビットのビットマスクを構築する．これは RAM 上では $\mathcal{O}\lceil m/w \rceil$ ワードのビットマスクとなる．α 遷移は，$|\Sigma|$ 個のビットマスクを構築する．ϵ 遷移のビットマスクはクラスごとに構築するビットマスクの個数が異なる．$\mathcal{EXT}$ クラスでは，分配遷移と集約遷移が存在しないため，$d = 1$ となる．$\mathcal{NET}$，$\mathcal{EXNET}$ では，$\mathcal{O}(d)$ 個のビットマスクを構築する．また，$\mathcal{REG}$ クラスでは，逆伝搬遷移のために，$\mathcal{O}(d \log m)$ 個のビットマスクを構築する．

領域は，前処理の過程において，構築されるビットマスクが領域として使用される．

実行時間は，テキストの文字一つにつき，遷移の模倣を行う．$\mathcal{EXT}$ では，定数個のビットマスクによる遷移の模倣を行うので，実行時間は $\mathcal{O}n\lceil m/w \rceil$ となる．$\mathcal{NET}$ と $\mathcal{EXNET}$ では，$\mathcal{EXT}$ での処理を深さごとに行うので，実行時間は $\mathcal{O}(nd\lceil m/w \rceil)$ となる．$\mathcal{REG}$ では，$\mathcal{NET}$ での処理に加えて，逆伝搬遷移の模倣を行う．逆伝搬遷移は，深さごとに $\mathcal{O}\lceil m/w \rceil \log m$ 回の処理を行う．このことから，実行時間は $\mathcal{O}nd\lceil m/w \rceil \log m$ となる．

Algorithm 3 深さ d の逆伝搬遷移の模倣を行うアルゴリズム

1: **function** $Backward_N(D, d)$

Input:

D: TNFA N (R) の現在の状態集合

d: 模倣を行う逆伝搬遷移の深さ

2: $jump \leftarrow 1$

3: $D' \leftarrow D$

4: **for** $i \leftarrow 0, \ldots, \lceil \log m \rceil$ **do**

5: $D' \leftarrow (D' \& \sim BLK_B[d][i]) \mid (((D' \& BLK_B[d][i]) \gg jump) \,\&BLK_B[d][i])$

6: $jump \leftarrow jump \ll 1$

7: **end for**

8: **return** $D \mid D' \& DST_B[d]$

9: **end function**

表 3.1: 各クラスでの計算量

クラス	前処理時間	実行時間	領域
$\mathcal{EXT}$	$\mathcal{O}(m + \lvert\Sigma\rvert m)$	$\mathcal{O}\left(n\lceil\frac{m}{w}\rceil\right)$	$\mathcal{O}\left(\lceil\frac{m}{w}\rceil + \lvert\Sigma\rvert\lceil\frac{m}{w}\rceil\right)$
$\mathcal{NET}$	$\mathcal{O}(dm + \lvert\Sigma\rvert m)$	$\mathcal{O}\left(nd\lceil\frac{m}{w}\rceil\right)$	$\mathcal{O}\left(d\lceil\frac{m}{w}\rceil + \lvert\Sigma\rvert\lceil\frac{m}{w}\rceil\right)$
$\mathcal{EXNET}$	$\mathcal{O}(dm + \lvert\Sigma\rvert m)$	$\mathcal{O}\left(nd\lceil\frac{m}{w}\rceil\right)$	$\mathcal{O}\left(d\lceil\frac{m}{w}\rceil + \lvert\Sigma\rvert\lceil\frac{m}{w}\rceil\right)$
$\mathcal{REG}$	$\mathcal{O}(dm \log m + \lvert\Sigma\rvert m)$	$\mathcal{O}\left(nd\lceil\frac{m}{w}\rceil \log m\right)$	$\mathcal{O}\left(d\lceil\frac{m}{w}\rceil \log m + \lvert\Sigma\rvert\lceil\frac{m}{w}\rceil\right)$

第4章

SIMD命令の正規表現照合アルゴリズムへの導入

本章では，3章で説明した正規表現照合アルゴリズムにSIMD命令を導入する．

4.1 SIMD命令の目的

3章のアルゴリズムでは，パターンRから構築したTNFAの状態数mがワードサイズwを超えた場合，状態を表すビット列をwビット毎に分割し，分割したビット列に対しビット演算を行う．例として，$m > w$であるときのビットANDのアルゴリズムをAlgorithm 4に示す．Algorithm 4のような処理は，SIMD命令を用いることで分割数が少なくなり，全体として演算回数が少なくなることが期待できる．

本研究で用いた，Intel AVX2命令セットでは，256ビットレジスタ上での演算を提供している．一方で，広く普及している計算機は，ワードサイズが64であることから，SIMD命令を用いることで4倍のビット数の演算を一度に実行することができる．

4.2 ビット演算へのSIMD命令の適用

ビット演算$\{|, \&, \oplus\}$は，これらの演算を行うSIMD命令が存在するので，以下の命令を用いた．__mm256iは8，16，32，64ビットの整数型を格納する256ビットレジスタの内容を指すデータ型である．

- “|” __mm256i _or_si256(__mm256i a, __mm256i b)
- “&” __mm256i _and_si256(__mm256i a, __mm256i b)

- “$\oplus$” __mm256i _xor_si256(__mm256i a, __mm256i b)

ビット NOT は AVX2 命令セットで提供されていないため，複数の命令を用いて実現する（Algorithm 5）．__mm256_cmpeq_epi8(D, D) は，2 個の引数の対応するビットが同じであるとき，そのビットが 1 となる．引数に同じデータを渡すことで，レジスタ全体のビットを 1 にしている．このレジスタと XOR をとることで，ビットを反転する．

4.3 SIMD 命令を用いないビット演算

4.2 節で SIMD 命令を適用しなかった命令として，加減算 $\{+, -\}$，ビットシフト $\{\ll, \gg\}$ がある．加減算については，レジスタ全体で，繰上げを含めた加減算の命令を提供されていない．このため，加減算は，SIMD 命令を用いない実装と同様に，64 ビットのワード毎に計算を行い，繰り上げを次のワードに加算することを繰り返す．ビットシフトは，レジスタ全体での演算は 8 ビット毎のシフトのみ提供されているが，8 ビット以下のシフトの操作を複数行う必要がある．このため，SIMD 命令を用いない実装と同様に，64 ビットのワード毎にビットシフトを行う．

Algorithm 4 長さが w を超えたビットマスクのビット AND

1: **function** BITAND(D_1, D_2)

Input:

D_1, D_2: 演算を行うビットマスク（w ビットの要素の配列とみなす．長さが等しいことが保証される．）

2: $d \leftarrow D_1$ の長さ

3: **for** $i \leftarrow 1, \ldots, \lceil \frac{d}{w} \rceil$ **do**

4: $D_1[i] \leftarrow D_1[i] \& D_2[i]$

5: **end for**

6: **return** D_1

7: **end function**

Algorithm 5 ビット NOT 演算.

1: **function** BITNOT(D)

2: $a \leftarrow$ __mm256_cmpeq_epi8(D, D)

3: **return** __mm256_xor_si256(D, a)

4: **end function**

第5章

実験

本章では3章で説明したKanetaらのアルゴリズムについて，4章で説明したSIMD命令を導入したアルゴリズムと導入していないアルゴリズムをC++上で実装し，前処理時間，実行時間を比較する．また，バックトラック法を用いているPythonライブラリとKanetaらのアルゴリズムでの実行時間の比較も行う．

5.1 実験環境

入力データとして用いたテキストは，Pizza&Chili Corpus[1]で公開されている，雑誌や学会誌の情報を提供するXMLデータを用いた．テキストを先頭から1MB，10MB，50MB，100MBに分割したデータを作成した．パターンは，正規表現，拡張文字列パターン，ネットワーク表現，拡張ネットワーク表現それぞれの，人工的に作成したパターンを用いた．パターンは，TNFAに構築したときの状態数が32，64，128，256，512，1024となるデータと，URLのフォーマットを照合するパターンを作成した．

5.2 実験方法

また，テキスト長による変化の比較として，1MB，10MB，50MB，100MBに分割したテキストデータと，URLのフォーマットを照合するパターンを正規表現照合し，実行時間を計測する．状態数による変化の比較として，10MBのテキストデータと，各クラスによるパターンを正規表現照合し，前処理時間，実行時間を計測する．このパターンは，

[1] http://pizzachili.dcc.uchile.cl/texts/xml/

構築した TNFA の状態数が，32 から 1024 まで 32 刻みで作成したものを用いる．Python ライブラリとの比較では，1MB のテキストデータと，$\mathcal{EXT}$ クラスのパターンを TNFA の状態数が 6 から 39 まで 1 刻みで作成したものを用いる．

5.3 実験結果

図 5.1〜 5.4 に各表現クラスでの，テキスト長毎の正規表現照合の実行時間を示す．図 5.9 〜 5.12 に各表現クラスでの，状態数毎の正規表現照合の前処理時間を示す．図 **??**〜 5.8 に各表現クラスでの，状態数毎の正規表現照合の実行時間を示す．図 **??**に Python ライブラリと Kaneta らのアルゴリズムの，状態数毎の正規表現照合の実行時間を示す．

5.3.1 テキスト長毎の変化

テキスト長毎の実行時間は，どのクラスもテキスト長に対して線形に実行時間が増加したことを確認した．また，SIMD を用いた照合アルゴリズムと既存アルゴリズムで実行時間の差はみられなかった．これは，データをレジスタにロードする，もしくは，レジスタからメモリにストアする処理が頻繁に起こっていることで，SIMD 命令を用いないアルゴリズムとの差が生じなかったと考えられる．クラス毎の処理時間を比べると，拡張文字列パターンの照合の実行時間が他のクラスに比べて小さかった．これは，拡張文字列パターンでは，Scatter 遷移と Gather 遷移が現れないため，パターンの深さが常に 1 であることが原因と考えられる．

5.3.2 状態数毎の変化

状態数毎の実行時間は，状態数が大きくなるにつれ，実行時間が増加していることが確認できた．一方で，ある状態数で実行時間が大きく増加していることを確認した．前処理時間も同様に，増加していた．SIMD を用いた照合アルゴリズムと既存アルゴリズムで，実行時間と前処理時間の差はみられなかった．

5.3.3 バックトラック法との比較

Python ライブラリでは，実行時間が状態数に対して指数的に増加していることが確認できた．一方で，Kaneta らのアルゴリズムは実行時間が状態数に対して線形の増加に抑えられていた．Kaneta らのアルゴリズムは，実行時間の計算量が，状態数に対して線形であることから理論通りの動作であることが確認できた．

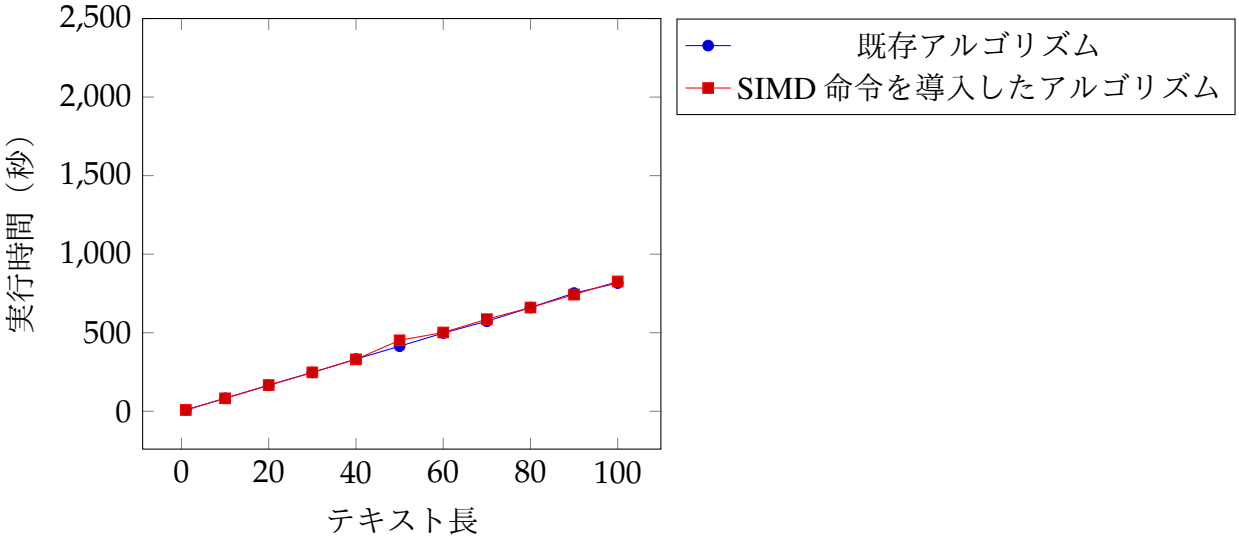


図 5.1: 拡張文字列パターンのテキスト長毎の実行時間の変化

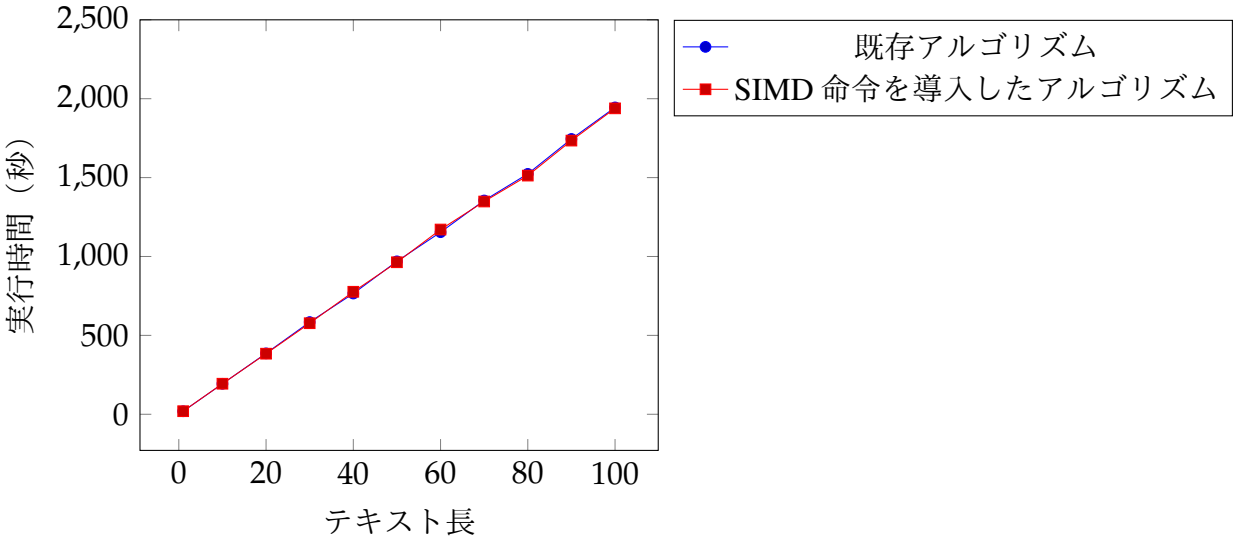


図 5.2: ネットワーク表現のテキスト長毎の実行時間の変化

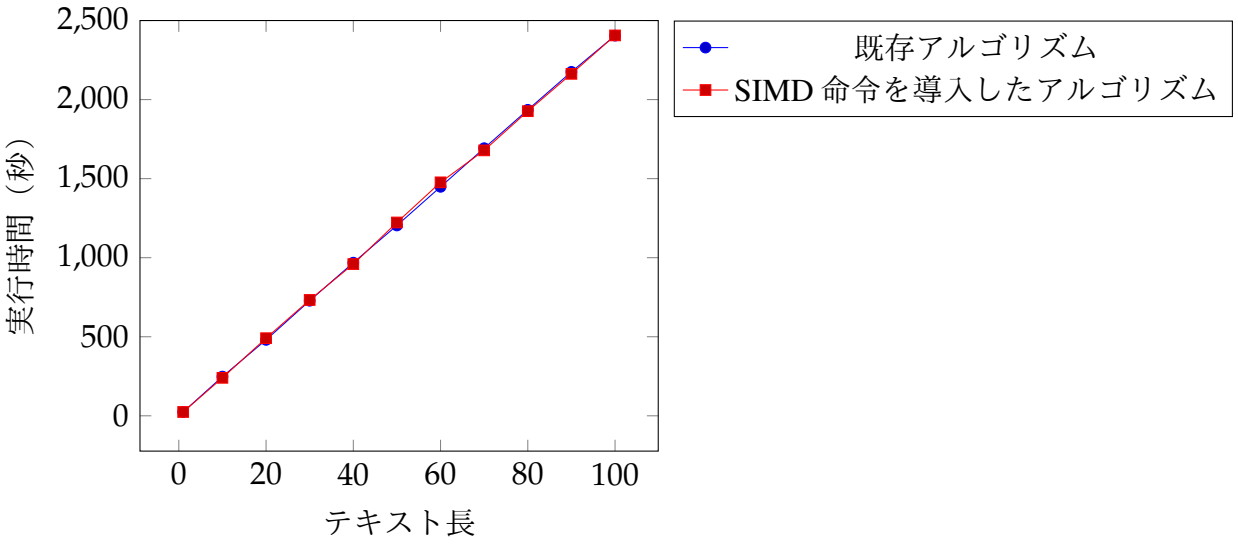


図 5.3: 拡張ネットワーク表現のテキスト長毎の実行時間の変化

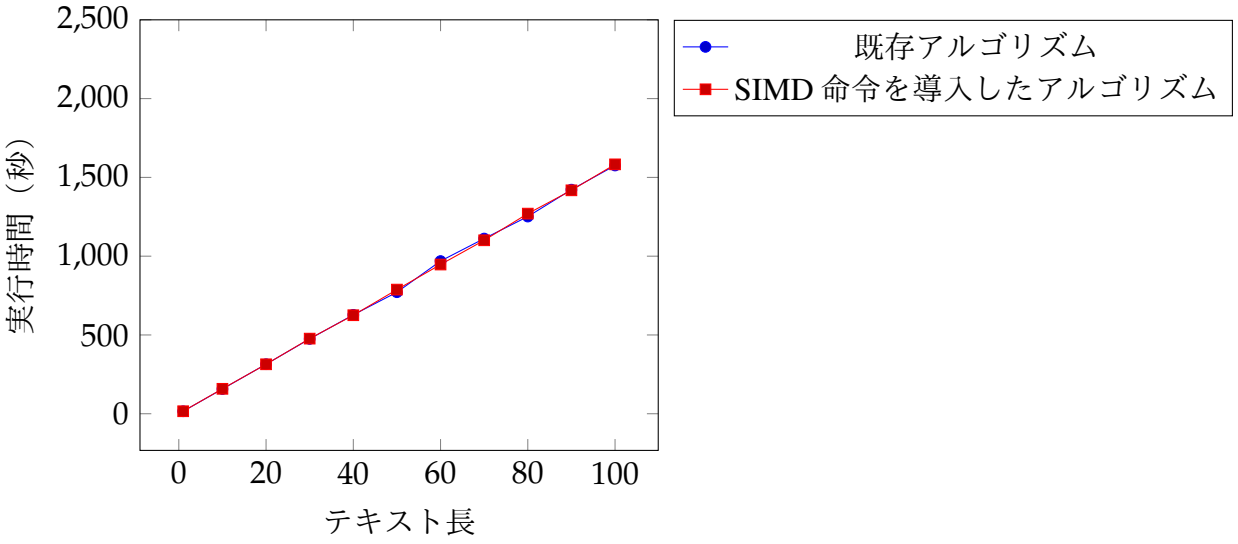


図 5.4: 正規表現のテキスト長毎の実行時間の変化

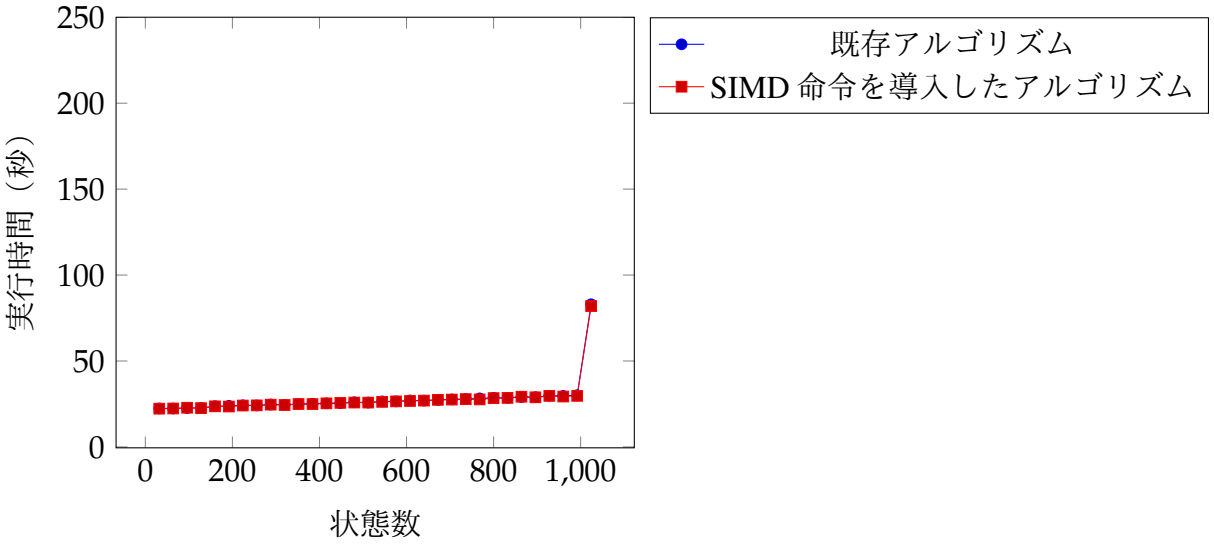


図 5.5: 拡張文字列パターンの状態数毎の実行時間の変化

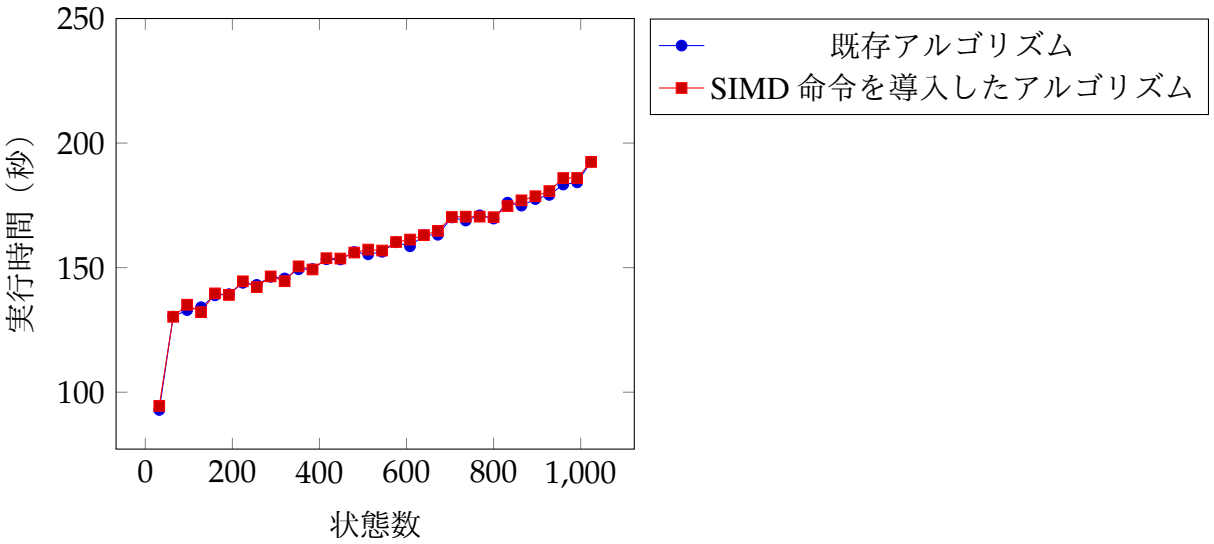


図 5.6: ネットワーク表現の状態数毎の実行時間の変化

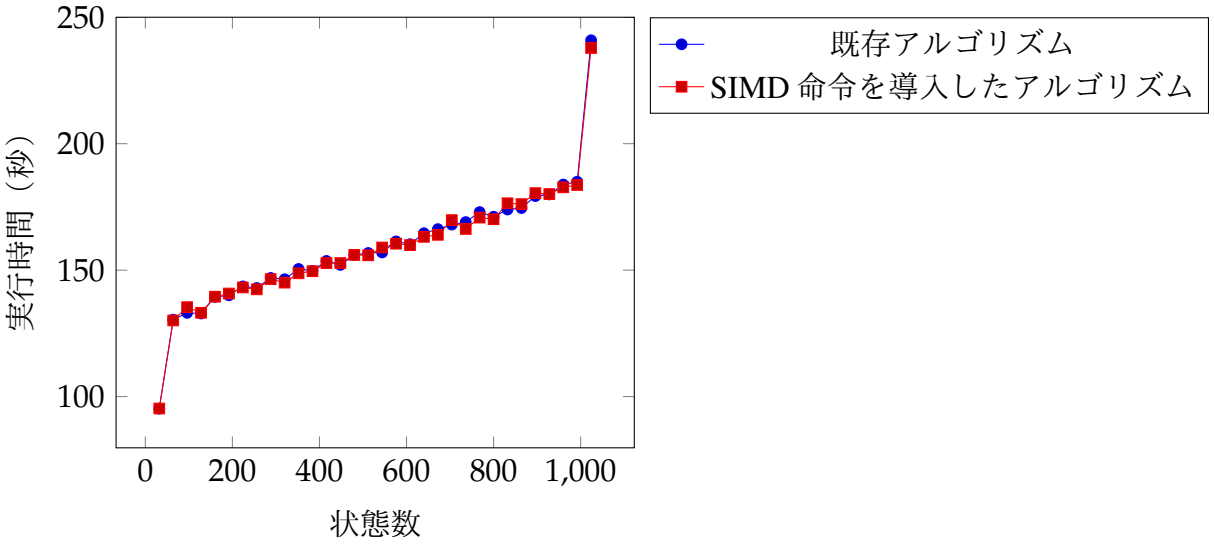


図 5.7: 拡張ネットワーク表現の状態数毎の実行時間の変化

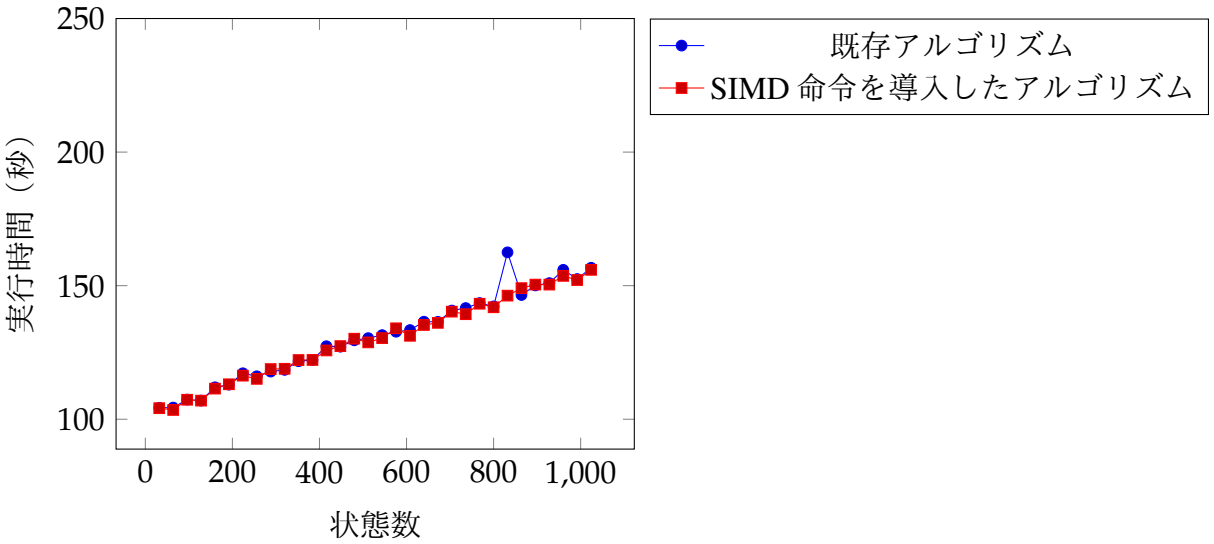


図 5.8: 正規表現の状態数毎の実行時間の変化

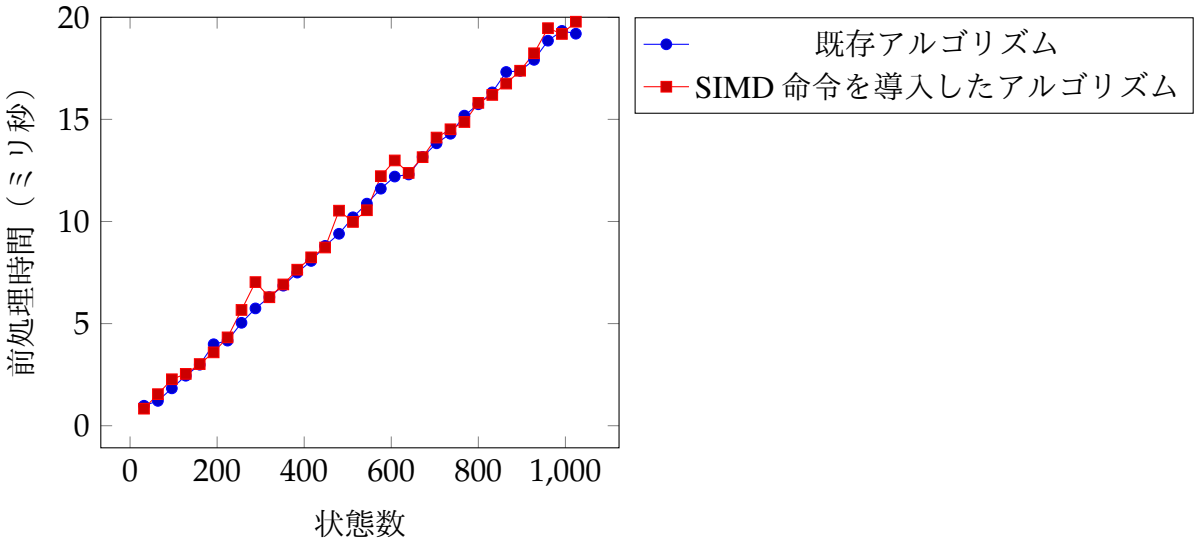


図 5.9: 拡張文字列パターンの状態数毎の前処理時間の変化

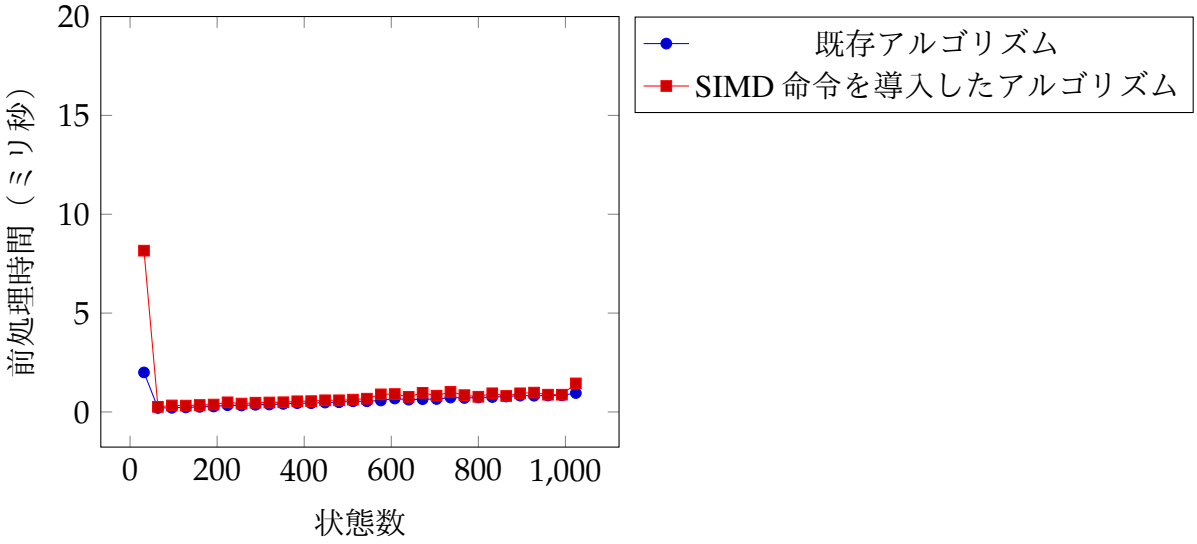


図 5.10: ネットワーク表現の状態数毎の前処理時間の変化

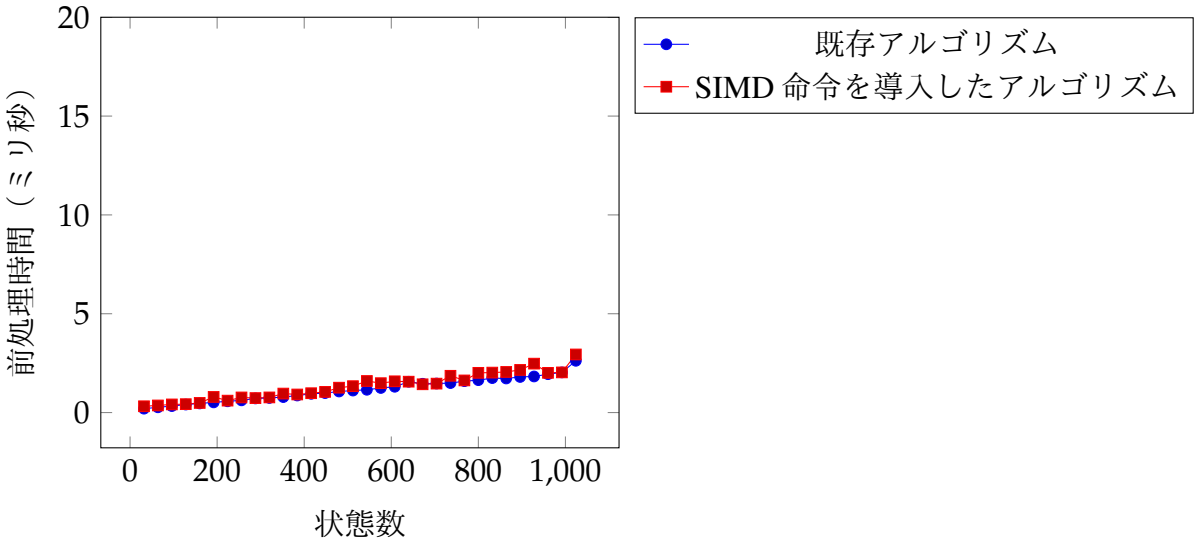


図 5.11: 拡張ネットワーク表現の状態数毎の前処理時間の変化

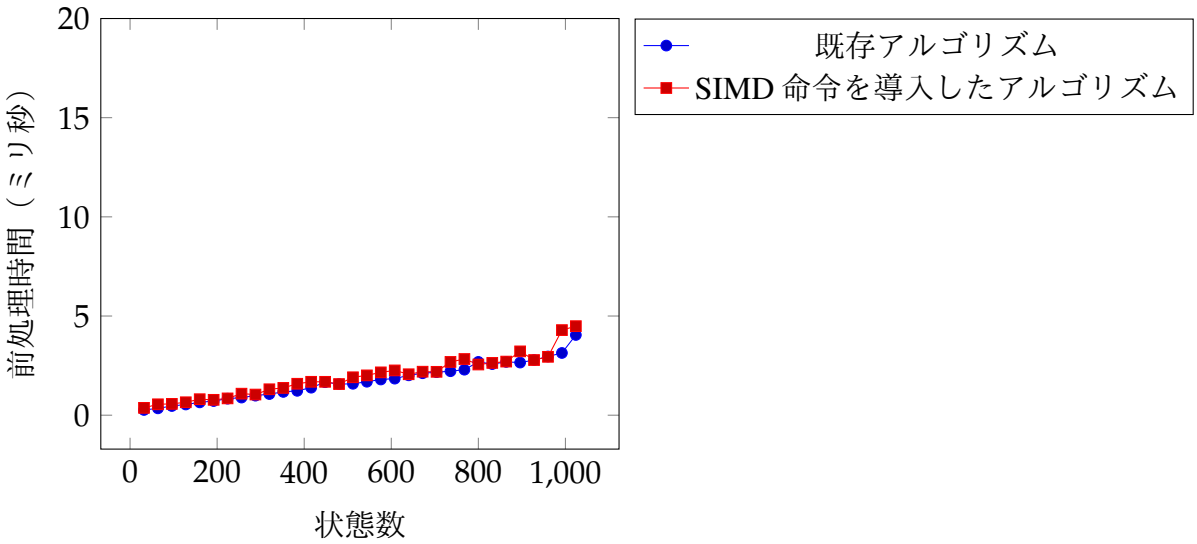


図 5.12: 正規表現の状態数毎の前処理時間の変化

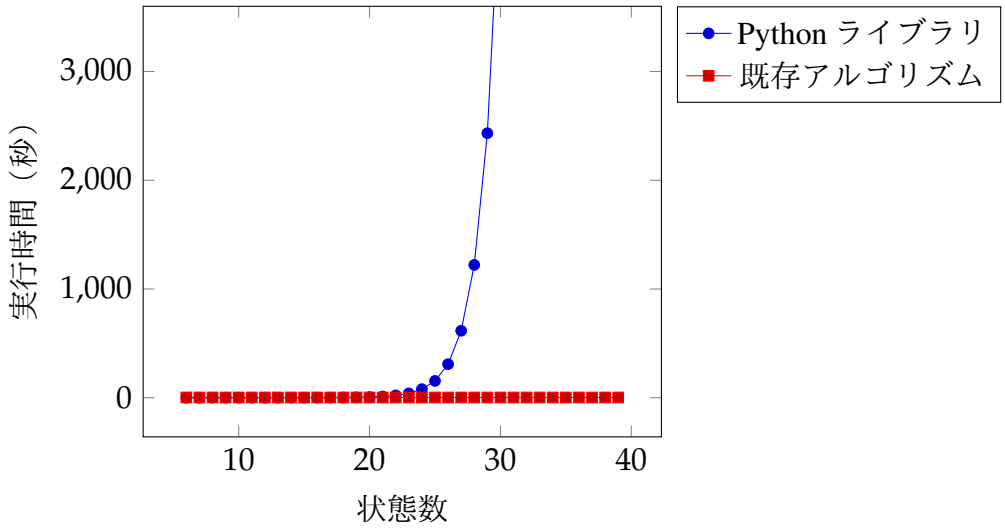


図 5.13: Python ライブラリと，Kaneta らのアルゴリズムの実行時間の変化

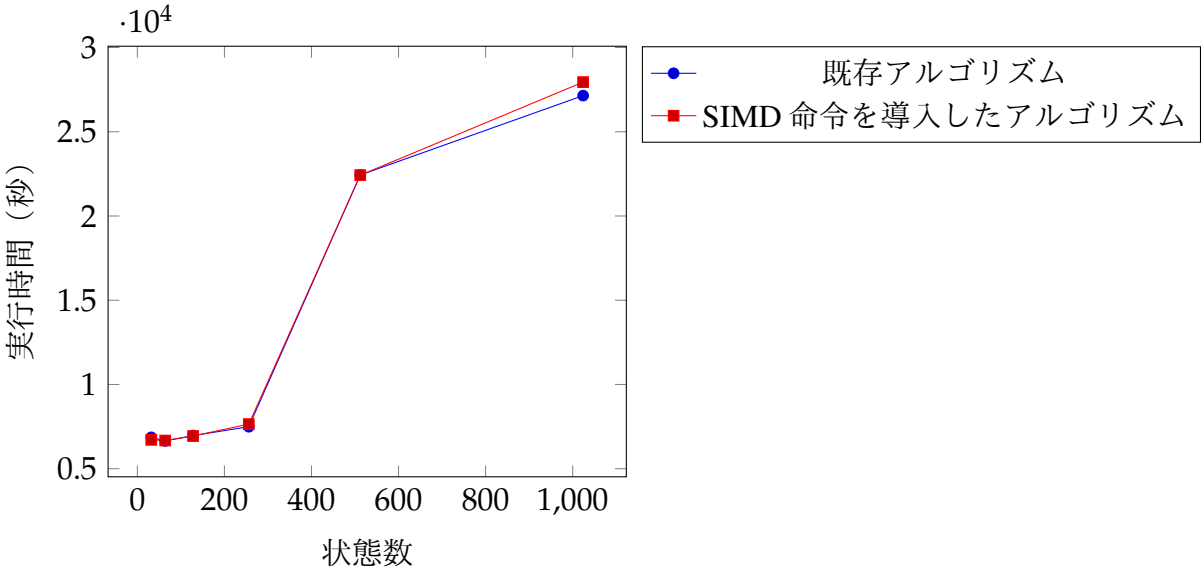


図 5.14: パターン長毎の実行時間の変化

第6章

おわりに

本論文では，Kaneta らのアルゴリズムに SIMD 命令を導入し，それぞれを実装して，いくつかの表現クラスに対し，実行時間の比較を行った．実験の結果，SIMD 命令を導入した実装と導入していない実装では実行時間の差はみられなかった．拡張文字列パターンの実行時間は他の表現クラスに比べて小さかったことを確認した．また，バックトラックを用いている Python の正規表現照合アルゴリズムでは実行時間が指数的に増加するようなパターンを用いても，本アルゴリズムは実行時間の増加が線形に抑えられることを確認した．

今後の課題として，遷移の処理を SIMD 命令によって完結させることが挙げられる．これは，主記憶へのデータの移動を最小限にすることでアルゴリズムの高速化が期待できる．また，状態数ごとの実験において，実行時間が大きく変化する箇所があった．この原因についての考察も今後の課題とする．

謝辞

本研究でご指導いただきました北海道大学大学院有村博紀教授，喜田拓也准教授は深く感謝いたします．喜田准教授には数多くの助言をいただき，論文指導を密に行なっていただきました．情報知識ネットワーク研究室の秘書真鍋由布さんには，書類，備品の管理など様々な面でご支援いただきました．研究室の先輩方には，論文のチェックや発表の指導など，ご指導いただきました．大変感謝しております．最後に，生活の支援をしてくださった家族に感謝を申し上げます．

参考文献

[1] Stephen Cole Kleene. Representation of events in nerve nets and finite automata. Technical report, RAND PROJECT AIR FORCE SANTA MONICA CA, 1951.

[2] Yusaku Kaneta, Shin-Ichi Minato, and Hiroki Arimura. Fast bit-parallel matching for network and regular expressions. In *International Symposium on String Processing and Information Retrieval*, pp. 372–384. Springer, 2010.

[3] Victor Mikhaylovich Glushkov. The abstract theory of automata. *Russian Mathematical Surveys*, Vol. 16, No. 5, p. 1, 1961.

[4] Ken Thompson. Programming techniques: Regular expression search algorithm. *Communications of the ACM*, Vol. 11, No. 6, pp. 419–422, 1968.

[5] Ricardo Baeza-Yates and Gaston H Gonnet. A new approach to text searching. *Communications of the ACM*, Vol. 35, No. 10, pp. 74–82, 1992.

[6] Gonzalo Navarro. Nr-grep: a fast and flexible pattern-matching tool. *Software: Practice and Experience*, Vol. 31, No. 13, pp. 1265–1312, 2001.

[7] Sun Wu and Udi Manber. Fast text searching: allowing errors. *Communications of the ACM*, Vol. 35, No. 10, pp. 83–91, 1992.

[8] Gonzalo Navarro and Mathieu Raffinot. Fast regular expression search. In *International Workshop on Algorithm Engineering*, pp. 198–212. Springer, 1999.

[9] Gonzalo Navarro and Mathieu Raffinot. *Flexible pattern matching in strings: practical on-line search algorithms for texts and biological sequences*. Cambridge university press, 2002.

[10] Gonzalo Navarro and Mathieu Raffinot. Compact dfa representation for fast regular expression search. In *International Workshop on Algorithm Engineering*, pp. 1–13. Springer, 2001.

[11] Gonzalo Navarro and Mathieu Raffinot. Fast and simple character classes and bounded gaps pattern matching, with applications to protein searching. *Journal of Computational Biology*, Vol. 10, No. 6, pp. 903–923, 2003.

[12] Eugene W Myers. Approximate matching of network expressions with spacers. *Journal of Computational Biology*, Vol. 3, No. 1, pp. 33–51, 1996.