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 [1]

February 5, 2020

[1] 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 4 年
情報知識ネットワーク研究室

背景・目的

背景

文字列の集合を表す方法の一つに正規表現がある．
文字列中の，正規表現が表す文字列集合の要素を列挙したい．

近年の CPU は，長いレジスタ長を持ち，このレジスタを用いた SIMD 命令を提供している．

目的

SIMD 命令を用いて正規表現照合アルゴリズムを高速化する．

研究内容

研究内容

- 既存手法である Kaneta らのアルゴリズム [1] に対し，SIMD 命令を用いた実装を行う．
- それぞれの実装に対し，比較実験を行う．
  - 正規表現の部分クラスでの性能の比較（割愛）．
  - テキスト長に対する，実行時間の比較．
  - パターン長に対する，前処理時間と実行時間の比較．
  - バックトラック法では実行時間が大きいパターンでの，実行時間の調査．

主結果

- 各部分クラスにおいて，SIMD 命令を用いた実装と
  そうでない実装で，性能の差はみられなかった.
- それぞれの実装で，
  実行時間，前処理時間の差はみられなかった.
- どちらの実装も，テキスト長に対し，
  理論通り線形時間で動作することを確認した.
- バックトラック法を用いた python のライブラリで
  実行時間が指数的に増加するパターンにおいて，
  どちらの実装も高速に動作することを確認した.

関連研究

Thompson のアルゴリズム [2]

- $\mathcal{O}(nm)$ 時間，$\mathcal{O}(m)$ 領域
- 正規表現照合アルゴリズム
- Thompson オートマトンの構築

SHIFT-AND 法 [3]

- $\mathcal{O}(nm/w)$ 時間，$\mathcal{O}(|\Sigma|m/w)$ 領域
- 文字列照合アルゴリズム
- ビットシフトとビットアンドを用いたビット並列アルゴリズム

Kaneta らのアルゴリズム [1]

- $\mathcal{O}(nmdlogm/w)$ 時間，$\mathcal{O}(dm/w + |\Sigma|m/w)$ 領域
- 正規表現照合に対してビット並列手法を導入.
- 本研究で考察.

準備

- アルファベット Σ：文字を要素とする空でない有限集合.
- 文字列：文字を 0 個以上並べた列.
  - Σ^*：Σ 上の文字列全体の集合.
  - 空文字列 ϵ：長さが 0 の文字列.
- ワードマシンモデル（RAM）：
  W ビットのビット演算を，定数時間で計算可能な計算機モデル.
  本研究では，加減算についても定数時間で実行可能な
  モデル $RAM(+)$ 上でのアルゴリズムについて考察する.

正規表現照合問題

正規表現照合問題

テキスト T 中に出現する正規表現が表す
文字列集合の要素をすべて見つける問題.

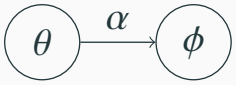
正規表現と正規表現が表す言語

- 定数文字：$\alpha \in \Sigma \cup \{\epsilon\}$ ならば，$L(\alpha) = \{\alpha\}$.
- 連接：$R_1 \cdot R_2$ について，$L(R_1 \cdot R_2) = L(R_1) \cdot L(R_2)$.
- 選言：$R_1 \mid R_2$ について，$L(R_1 \mid R_2) = L(R_1) \mid L(R_2)$.
- クリーネ閉包：$R*$ について，$L(R*) = L(R)^*$.

トンプソンオートマトン（TNFA）

正規表現に対して，再帰的に構築を行う．

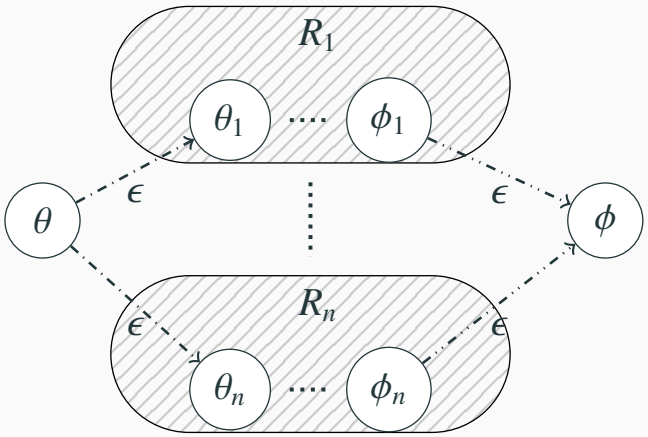
1. $N(\epsilon)$



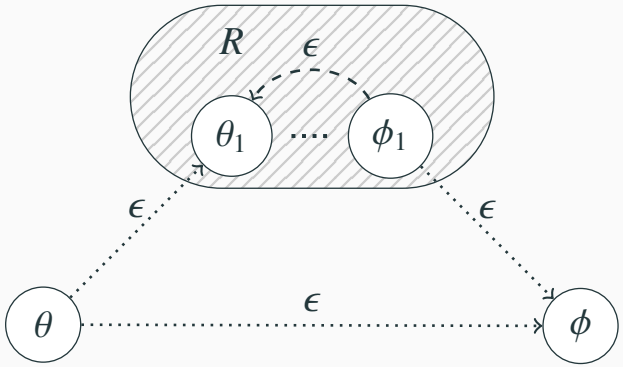
2. $N(\alpha)$

$= \theta_2$ $= \phi_{n-1}$

3. $N(R_1 \ldots R_n)$



4. $N(R_1 \mid \cdots \mid R_n)$

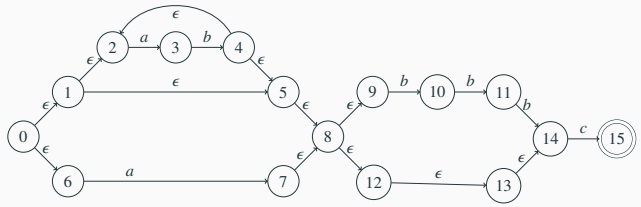


5. $N(R*)$

Kaneta らのアルゴリズム

- 前処理
  1. 正規表現から構文木を構築.
  2. 構文木から TNFA を構築.
  3. TNFA からビットマスクを構築.
     - このビットマスクの各ビットが，状態に対応.
- 実行時
  - ビットマスクを用いて遷移を模倣

$$a((ab)*|a)(bb|\epsilon)c \xrightarrow{1,\ 2}$$



3

$CHR[a]$	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0
$BLK_G[1]$	0	0	0	1	1	1	1	1	0	1	1	1	1	1	1	1	0
$SRC_G[1]$	0	0	0	1	0	1	0	0	0	1	0	1	0	0	0	0	0
$DST_G[1]$	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0

遷移の模倣

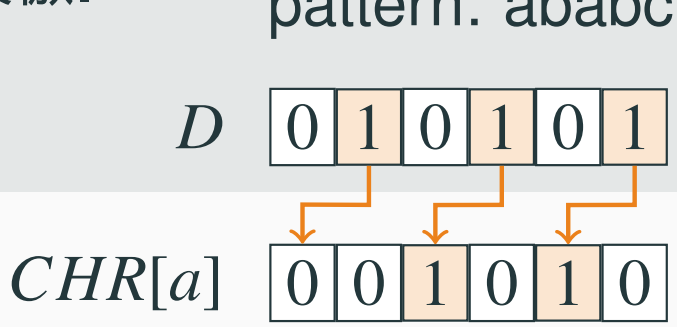
状態集合をビット列で表し，演算によって遷移の模倣を行う．
文字の遷移（α 遷移）と空文字列の遷移（ϵ 遷移）に分ける．

- α 遷移：SHIFT-AND 法によって模倣．
- ϵ 遷移：４つの遷移に分け，深さごとに遷移の模倣を行う．

SHIFT-AND 法 [3]

ビットシフトとビット AND による遷移の模倣．

- $D \leftarrow (D \ll 1) \& CHR[\alpha]$



遷移の模倣：ϵ 遷移

Scatter 遷移

選言によって分岐する遷移
引き算の繰り下げによって模倣をする.

$$D \leftarrow D \mid ((BLK_s[k] - (D\&SRC_s[k]))\&DST_s[k])$$

Gather 遷移

選言の分岐が合流する遷移
足し算の繰上げによって模倣をする.

$$D \leftarrow D \mid ((BLK_g[k] + (D\&SRC_g[k]))\&DST_g[k])$$

Propagate 遷移

受理状態の方向への遷移
引き算と XOR によって模倣をする.

$$A \leftarrow (D\&BLK_p[k]) \mid DST_p[k]$$

$$D \leftarrow D \mid (BLK_p[k]\&((\sim (A - SRC_p[k])) \oplus A))$$

Back 遷移

開始状態の方向への遷移
バレルシフタ技法によって模倣をする. $D \leftarrow (D\& \sim BLK_b[k][i]) \mid$

$$(((D\&BLK_b[k][i]) \gg jump)\&BLK_b[k][i])$$

SIMD 命令の導入（1/2）

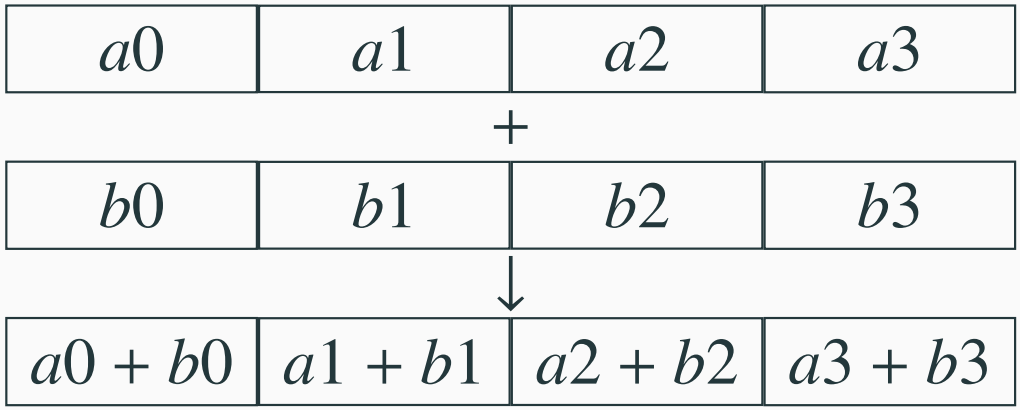
Kaneta らのアルゴリズムに SIMD 命令を導入する．

単一命令多重データ（Single Instruction Multiple Data，SIMD）

複数のデータを一つの命令によって処理を行う手法．
GPU や CPU 拡張命令などで用いられている．
本研究では，CPU 拡張命令である AVX2 を用いる．

AVX2

Intel が提供している，SIMD 拡張命令セット．
256 ビットレジスタでの，整数演算をサポートしている．



SIMD 命令の導入

- ビット AND，OR，XOR：SIMD 命令が提供されている．
  - ビット AND の場合：
    __mm256i_and_si256(__m256i a, __m256i b)
- ビット NOT：2 つの SIMD 命令によって演算可能．

  $a \leftarrow$__mm256_cmpeq_epi8(D, D)
  return _mm256_xor_si256i(D, a)
- 加減算，ビットシフト：レジスタ全体での SIMD 命令が提供されていないため，SIMD 命令を用いずに行う．

例．Scatter 遷移の場合

$$D \leftarrow D \mid ((BLK_s[k] - (D \& SRC_s[k])) \& DST_s[k])$$

SIMD 命令を用いる

実験

実装

C++上で SIMD を用いた実装と用いない実装を行なった.

SIMD 命令は，x86intrin.h ヘッダの組み込み関数を用いた.

実験

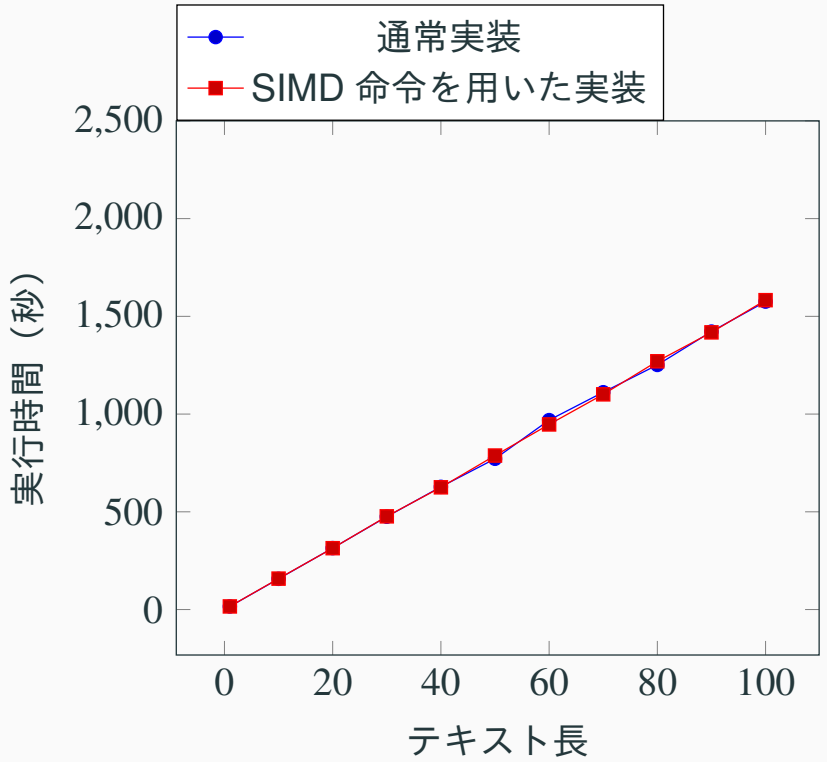
テキスト長，パターン長を変えて比較を行った.

- 実験環境：
  - CPU：Intel Core i5 2.3GHz × 2
  - メモリ：8GB
  - OS：macOS Catalina 10.15.2
  - コンパイラ：g++ 9.2.0
- テキスト：Pizza & Chili Corpus[1] XML データ.
- パターン：人工的に作成したパターン.

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

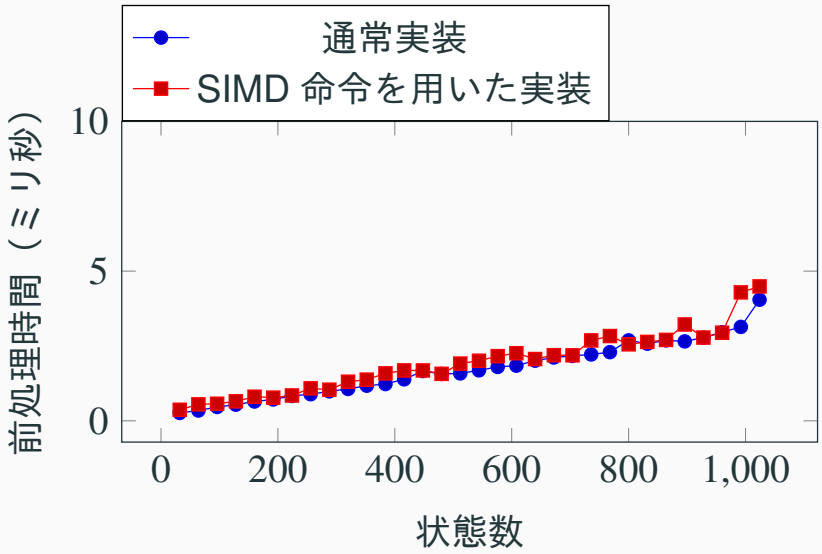
実験結果（テキスト長に対する実行時間の変化）

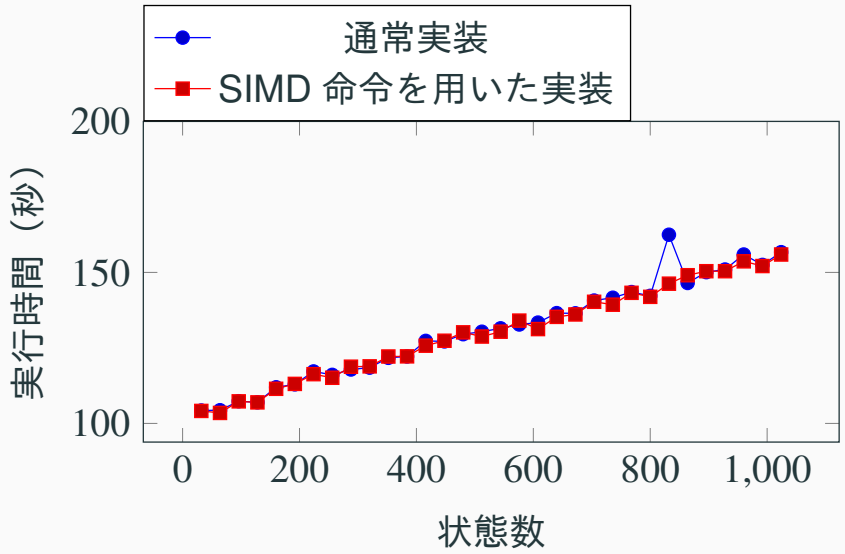
- それぞれの実装で実行時間の変化はみられなかった.
  - メモリとレジスタのやりとりが頻繁に行われる.
- どちらの実装もテキスト長に対して，線形時間の実行時間となった.



実験結果（パターン長に対する前処理，実行時間の変化）

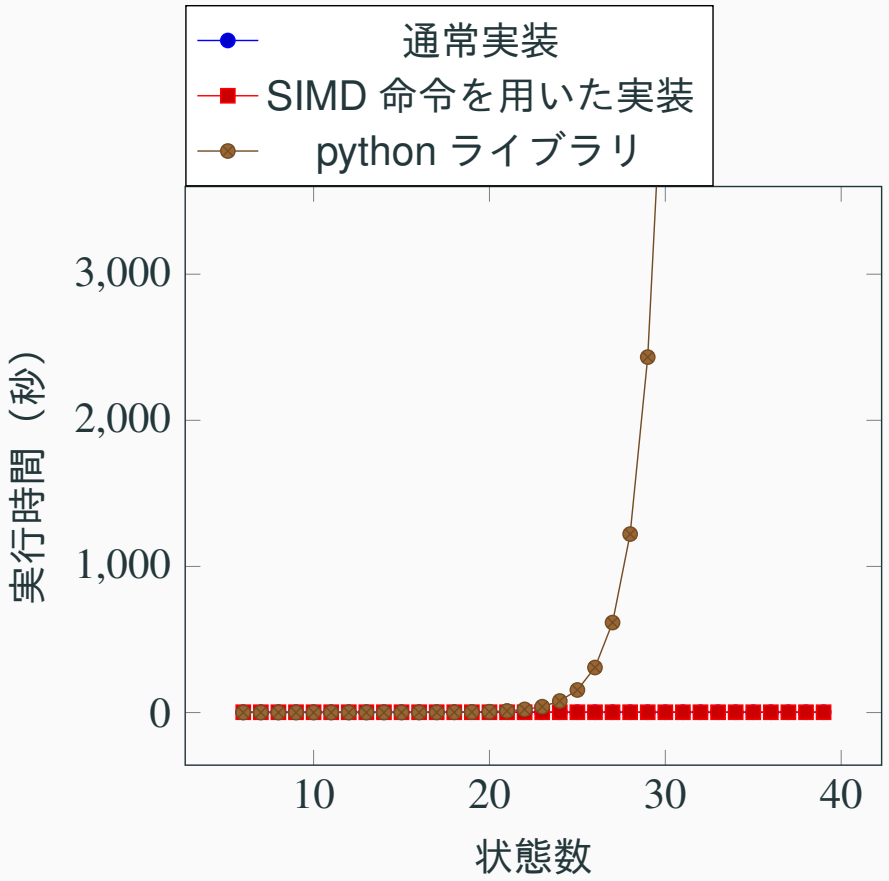
- ふたつの実装で前処理時間，実行時間の変化はみられなかった.
- どちらの実装もパターン長に対して，線形時間の前処理時間，実行時間となった.





実験結果（バックトラック法との比較）

- バックトラック法では
  実行時間が指数的に
  増加するパターンでも，
  Kaneta らのアルゴリズムは，
  高速に動作した．



まとめ

- Kaneta らのアルゴリズムを実装した.
- SIMD 命令を用いてアルゴリズムの高速化を図った.
- SIMD 命令を用いることによる
  実行時間の改善はみられなかった.
- 今後の課題
  - 実験結果のより深い考察.
    - 実行時の CPU の状態の確認.
  - SIMD 命令での高速化.
    - 加算やビットシフトを SIMD 命令を組み合わせて演算.
    - 各遷移の模倣を SIMD 命令で完結.
  - Back 遷移を定数回の演算で実現できるか検討.

参考文献

[1] 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.

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

[3] 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.