北海道大学
Hokkaido University

情報知識ネットワーク特論
「情報検索とパターン照合」

情報科学研究科 コンピュータサイエンス専攻
情報知識ネットワーク研究室
喜田拓也

第2回
Prefix型アルゴリズム※

Naïve アルゴリズム
KMP アルゴリズム
Aho-Corasick アルゴリズム
Shift-And/Or アルゴリズム

※ 参考文献：Felxible Pattern Matching in Strings, Gonzalo Navarro and Mathieu Raffinot

パターン照合問題(Pattern Matching Problem)とは？

- テキストT中に含まれるパターンPの出現を求める問題
  有名なアルゴリズム:
  - KMP法 (Knuth&Morris&Pratt[1974]) O(n+m) 時間
  - BM法 (Boyer&Moore[1977])
  - Karp-Rabin法 (Karp&Rabin[1987])

パターン P: compress

テキスト T: We introduce a general framework which is suitable to capture an essence of compressed pattern matching according to various dictionary based compressions. The goal is to find all occurrences of a pattern in a text without decompression, which is one of the most active topics in string matching. Our framework includes such compression methods as Lempel-Ziv family, (LZ77, LZSS, LZ78, LZW), byte-pair encoding, and the static dictionary based method. Technically, our pattern matching algorithm extremely extends that for LZW compressed text presented by Amir, Benson and Farach [Amir94].

Existence problem と All-occurrences problem

Existence problem

テキスト T: テクマクマヤコンテクマクマヤコン

パターン P: クマクマ

Yes!

All-occurrences problem

テキスト T: テクマクマヤコンテクマクマヤコン

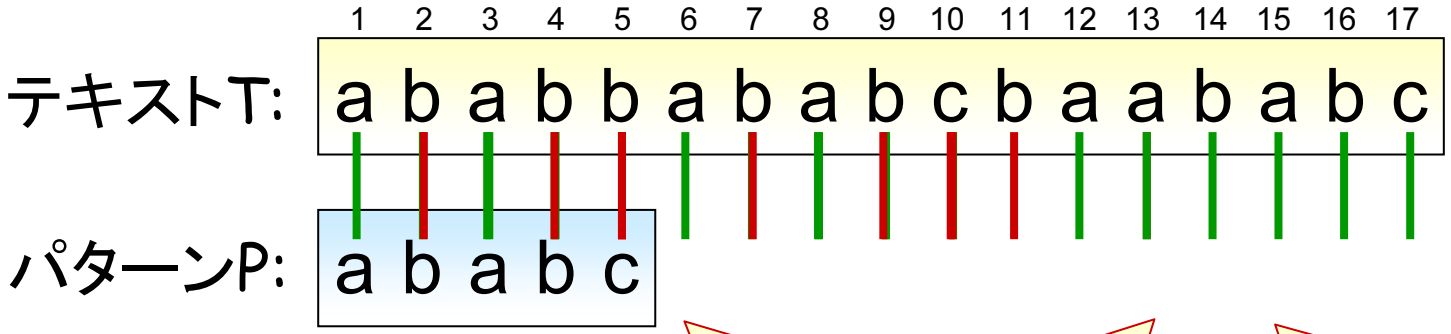
パターン P: クマクマ

2

10

全文検索で文書を単位として検索する場合は、Existence problemで充分
しかし、一般的にはAll-Occurrences problemを指すことが多い

Naïve アルゴリズム



パ
at po

パターン出現！
at position 13 of T

一文字づつずらして
マッチングしていく

テキスト上のポインタ
（比較する文字の現在
位置）が前後する！

```
Naïve-String-Matching (T, P)
1   n ← length[T].
2   m ← length[P].
3   for s ← 0 to n – m
4      do if P[1..m] = T[s+1…s+m]
5           then report an occurrence at s.
```

もっと大雑把
にいえば、
O(nm) 時間

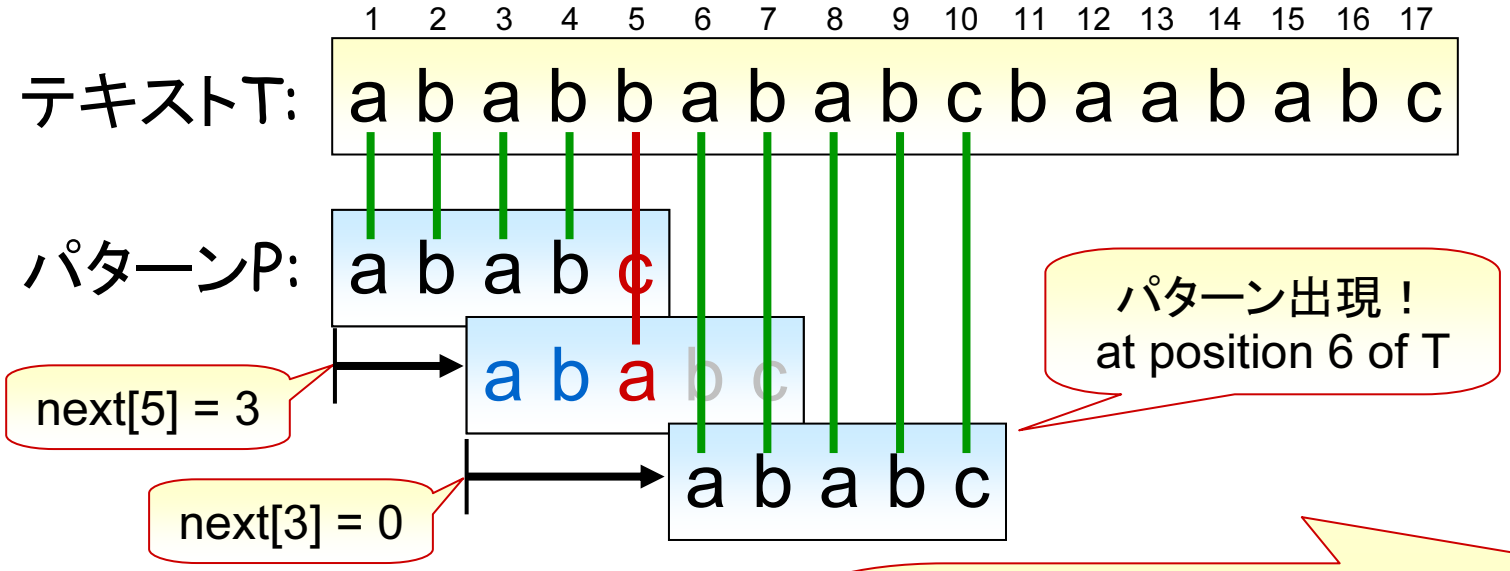
最悪の場合 $O((n-m+1)m)$ 時間かかる。

※演習：$T=a^8$, $P=a^4b$ の場合を文字比較の回数は何回か？

P=aaaab の意味

Knuth-Morris-Pratt アルゴリズム

D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt. Fast pattern matching in strings.
SIAM Journal on Computing, 6(1):323-350, 1977.



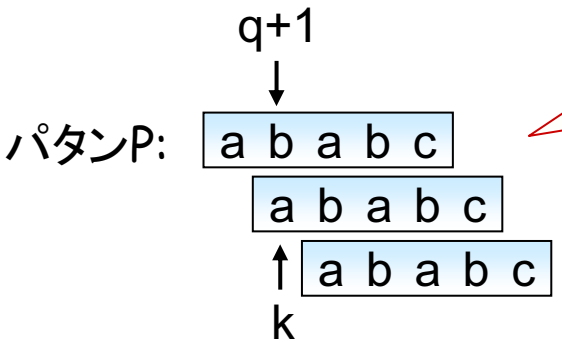

```
KMP-String-Matching (T, P)
1    n ← length[T].
2    m ← length[P].
3    q ← 1.
4    next ← ComputeNext(P).
5    for i ← 1 to n do
6       while q>0 かつ P[q]≠T[i] do q ← next[q];
7       if q=m then report an occurrence at i-m;
8       q ← q+1.
```

next関数によって次にPの何文字目とテキストを比較するかがわかる。(シフト量は q−next[q])
値が0のときは、テキストの次の文字と比較する。
テキストの各文字との比較はO(1)回ずつである。

最悪の場合でも O(n+m) 時間（nextはあらかじめ配列として計算）

next関数の計算

- next[q] = k の条件
  - P[1:k−1] が P[1:q−1] の接頭辞かつP[k]≠P[q]を満たす最長のもの



パターンをずらしながら比較し、
next[q]を計算する

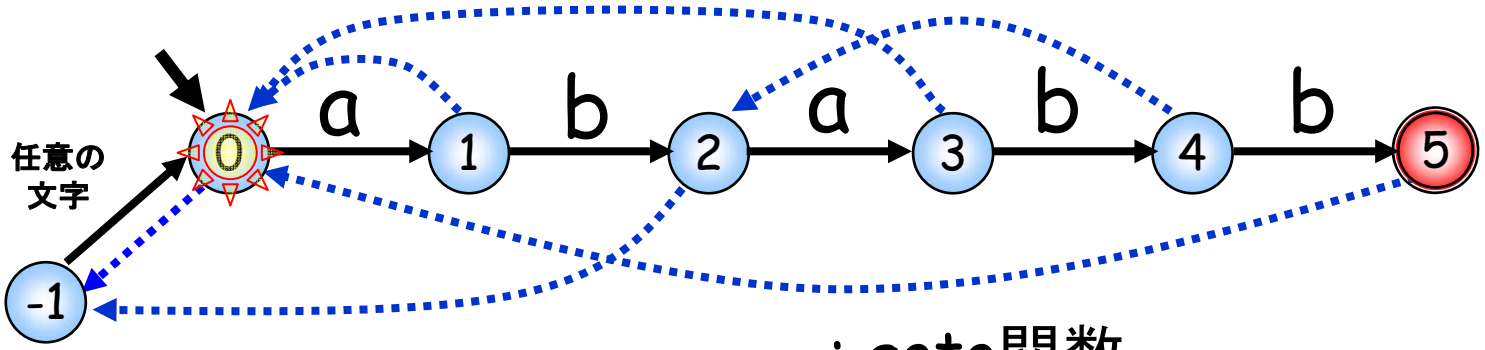
q	1	2	3	4	5	6
P[q]	a	b	a	b	c	
next(q)	0	1	0	1	3	1

O(m) 時間・領域

```
ComputeNext (P)
1    m ← length[P]. next[1] ← 0. k ← 0.
2    for q ← 1 to m do
3        while k>0 かつ P[q]≠P[k] do k ← next[k];
4        k ← k+1; q ← q+1;
5        if P[q]=P[k] then
6            next[q] ← next[k]
7        else
8            next[q] ← k;
```

決定性オートマトンによる照合

パターン P = ababb を検知する（ε 遷移付き）決定性有限オートマトン
（KMPオートマトン）



→ : goto関数

⇢ : failure関数

Next関数に対応

テキスト: a b a b a b b a

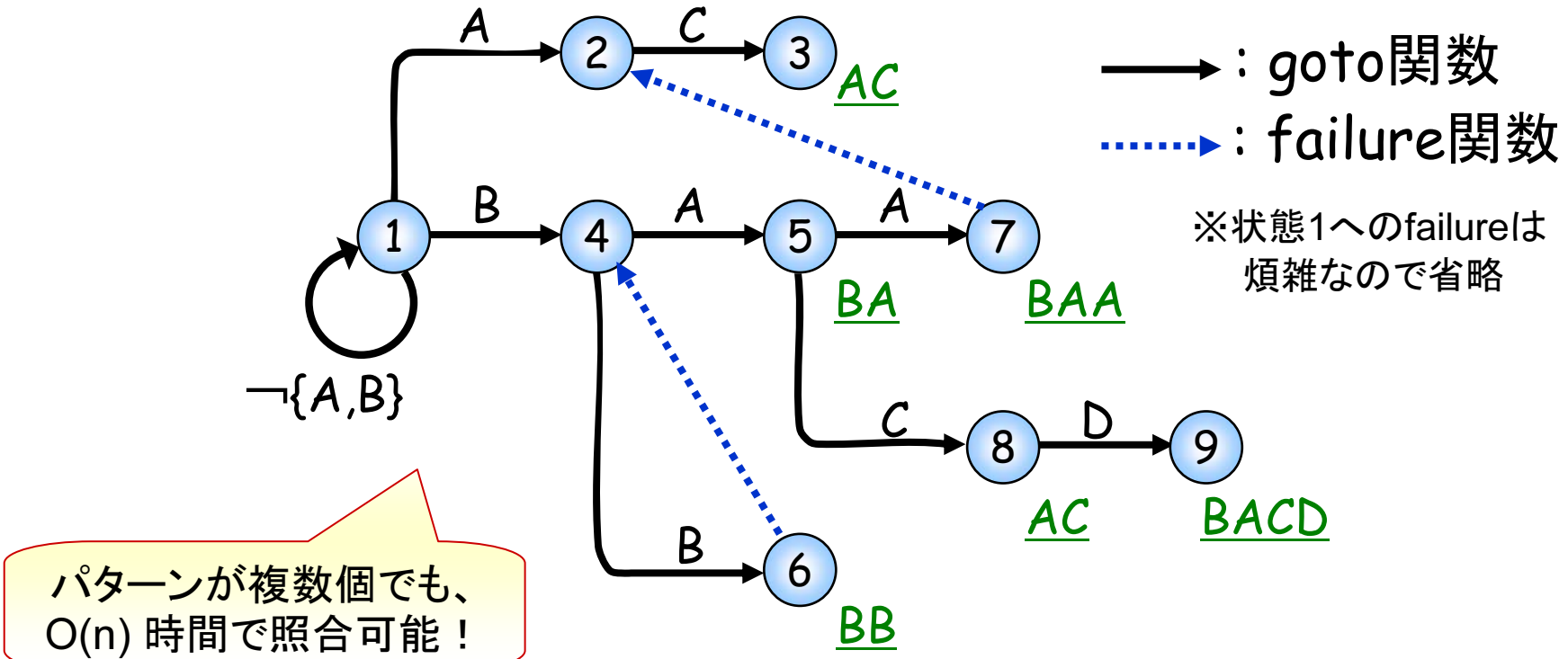
状態: 0 → 1 → 2 → 3 → 4 ⇢ 2 → 3 → 4 → 5 ⇢ 0 → 1

前処理は O(|Σ|m) 時間・領域。照合の時間はKMPと同じ O(n) 時間

Aho-Corasick(AC) アルゴリズム

A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search.
Communications of the ACM, 18(6):333-340, 1975.

パターン集合 ∏={AC,BA,BB,BAA,BACD} を検知する(ε遷移付き)順序機械
(AC照合機械)



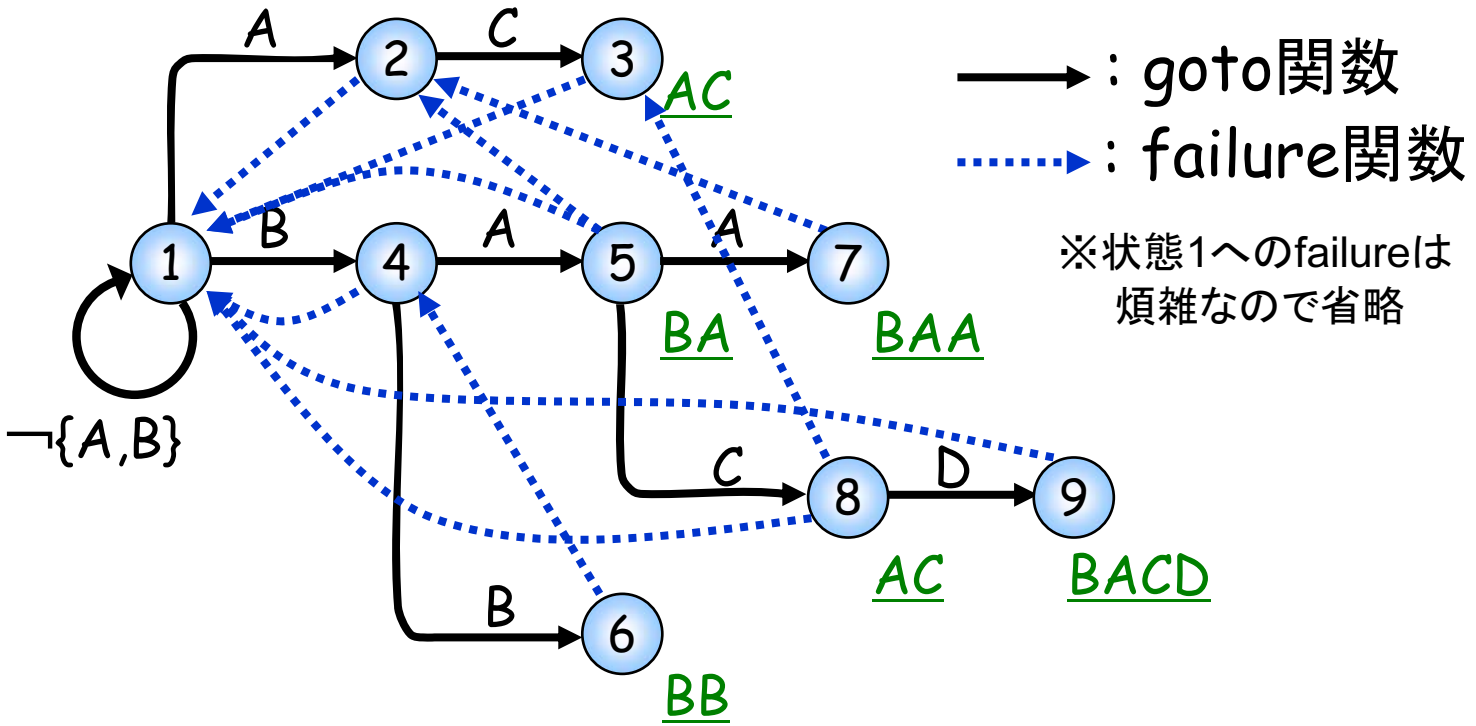
KMPオートマトンは、パターンが一つの場合のAC照合機械に等しい

※ 初期状態のとりかたが違うように見えるが、原理的には同じように構成できる

構成アルゴリズム

A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search.
Communications of the ACM, 18(6):333-340, 1975.

パターン集合 ∏={AC,BA,BB,BAA,BACD} を検知する（ε遷移付き）順序機械
（AC照合機械）



Phase 1. パタンを処理しながらtrie(gotoグラフ)を作る
Phase 2. 幅優先探索しfailure関数を求めつつ、出力関数を補完する
Phase 3. 最適化する

※ 詳細は配布資料を参照

AC照合機械構成の擬似コード

```
Build-ACmachine (P={p1,p2,…,pr)
1    AC_trie ← Trie(P).
2    // 配列g,f,o をそれぞれgoto関数、failure関数、output関数とする
3    // AC_trie上の出力があるノードを terminal とよぶ
4    initial_state ← root of AC_trie.
5    f[initial_state] ← nil.
6    for current in transversal order do
7        parent ← parent of current in AC_trie.
8        σ ← label of the transition from parent to current.
9        down ← f[parent].
10       while down ≠ nil and g[down, σ ] = nil do down ← f[down].
11       if down ≠ nil then
12           f[current] ← g[down, σ ].
13           if f[current] is terminal then
14                Mark current as terminal.
15                o[current] ← o[current] ∪ o[ f[current] ].
16           end_if
17       else
18           f[current] ← initial_state.
19       end_if
20   end_for
```

Phase 1

以下 Phase 2 の処理

※演習：P={cacao, ocaca} に対するAC照合機械を作図せよ

SIGMA方式のデータベース

- 例）九州大学大学院 農学研究院 昆虫学教室の「昆虫学データベース(KONCHU)」(http://konchudb.agr.agr.kyushu-u.ac.jp/index-j.html)
  - データの各レコードは、タグ付けされたデータ項目の並び
    (FTAX)科名(亜科名等を含むレコードもある)
    (GTAX)　属名(亜属名を含むレコードもある)
    (STAX)　種名または亜種名
    (JTAX)和名
    (DST) 分布(国内;国外)

タグ　データ

このレコードが「デリミタ」で仕切られて並べられ、一つのDBを構成する

検索は、検索語・タグ・デリミタをパターンとしたAC照合機械によって行われる

  - レコードの実例

    (FTAX)315180550000. Pieridae シロチョウ科(A)
    (GTAX)　Pieris (Artogeia)
    (STAX)　rapae crucivora Boisduval,1836
    (JTAX)モンシロチョウ
    (DST) HOKKAIDO, Rebun Is., Rishiri Is., HONSHU, Sado Is., Izu Isls., Ogasawara Isls., Awajishima Is., Oki Is., SHIKOKU, Shodoshima Is., KYUSHU, Tsushima Is., Iki Is., Goto Is., Tanegashima Is., Yakushima Is., Tokara Isls., AmamiOshima Is., Tokunoshima Is., OkinawaHonto Is., Miyako Is., Ishigaki Is., Iriomote Is., Yonaguni Is.; Taiwan, Far East Continent

富士通 Interstage Shunsaku Data Manager

- 特徴：
  - XML形式のテキストをDBにみたてて、逐次検索（パターン照合）による高速なデータアクセスを実現する
    - インデックス不要
    - データ様式の構成を柔軟に変更可能。
  - コアの部分は、九州大学の有川節夫教授（現、同大学副学長）と研究グループが開発した検索システム「SIGMA」をベースにしている

理論的な中核は
Aho-Corasick アルゴリズム

- 導入事例：
  - 国立遺伝学研究所 生命情報・DDBJセンター
    - 三大国際DNAデータバンクの一つ、DDBJ（日本DNAデータバンク）のARSA (All-round Retrieval of Sequence and Annotation) システム
  - 富士通社内
    - 生産管理システム、新電子電話帳システム

Shunsaku の優れた特徴（山手線方式）

http://software.fujitsu.com/jp/shunsaku/

- データの集まりを一つのXML文書として表現
- AC照合機械をフルに活用して検索　（1CPUで100MB/s）
- 分散処理・フォールトトレラント機構 etc.
- 複数のクエリをまとめて処理することで
  高速なレスポンスを確保（山手線方式）

数万件の同時アクセス
でも大丈夫！

ちょっと、ひといき・・・

- ここまでのまとめ
  - naïveな照合は大きなテキストやテキスト・ストリームには不向き
  - KMPアルゴリズムはO(m)時間・領域の前処理の後、O(n) 時間で照合
  - ACアルゴリズムはKMPオートマトンの考えを複数パターンに拡張したものである
    - O(|Σ|m)時間・領域の前処理の後、O(n) 時間で照合（ここで、mはパターン長の総和）

ペンギン隊の行進（旭山動物園にて '05.3.13）

～トリビア～

二つの整数xとyの内容を余計な変数を用いずに交換できるか？

ビットパラレルにおけるビット列の基本操作

- 論理演算

ビット単位の	記号	集合	C	
– 否定(not)		～x	~x	補集合
– 積(and)		x & y	x & y	積集合
– 和(or)		x | y	x | y	和集合
– 排他的論理和(xor)		x ⊕ y	x ^ y	異なり集合（仮称）
– (0充填)左シフト		x << y	x << y	全要素の加算
– (0充填)右シフト		x >> y	x >> y	全要素の減算
– 代入		x ← y	x = y	
– 比較演算子		x>y, x<y	x>y, x<y	

論理シフトとも言う

ここでは、すべて unsigned

- 算術演算
  - 算術マイナス −x
  - 加算 x+y
  - 減算 x−y
  - 乗算・除算・剰余はほとんど使われない

注）1の補数と2の補数では異なる

参考：「ハッカーのたのしみ -本物のプログラマはいかにして問題を解くか」ヘンリー・S・ウォーレン, Jr.: “Hacker’s Delight”訳本

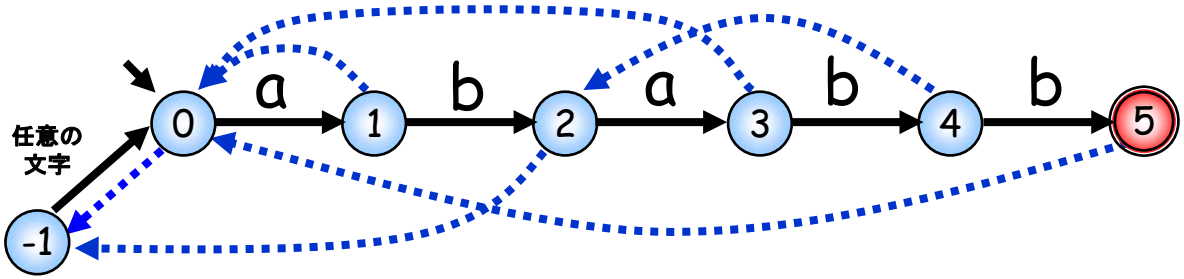
Shift-And アルゴリズム

R. A. Baeza-Yates and G. H. Gonnet. A new approach to text searching. Proceedings of the 12th International Conference on Research and Development in Information Retrieval, 168-175. ACM Press, 1989.

S. Wu and U. Manber. Fast text searching allowing errors. Communications of the ACM, 35(10):83-91, 1992.

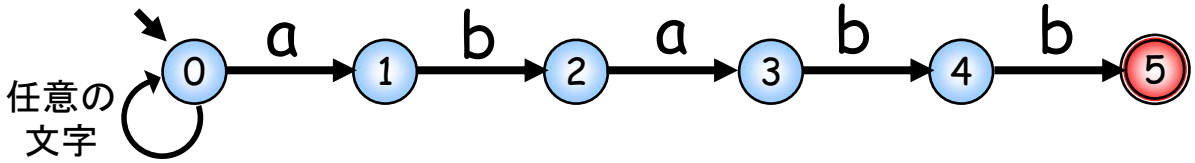
- レジスタ長のビット演算が並列に計算されることを利用
- パタン長mがワード長wよりも短い場合には、O(n)時間で非常に高速に動作する
  - 一般には O(n･⌈m/w⌉)時間、前処理はO(m+|Σ|)

パタン P = ababb を受理する決定性有限オートマトン



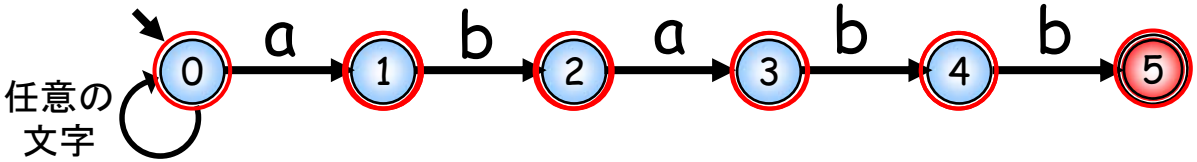
パタン P = ababb を受理する非決定性有限オートマトン

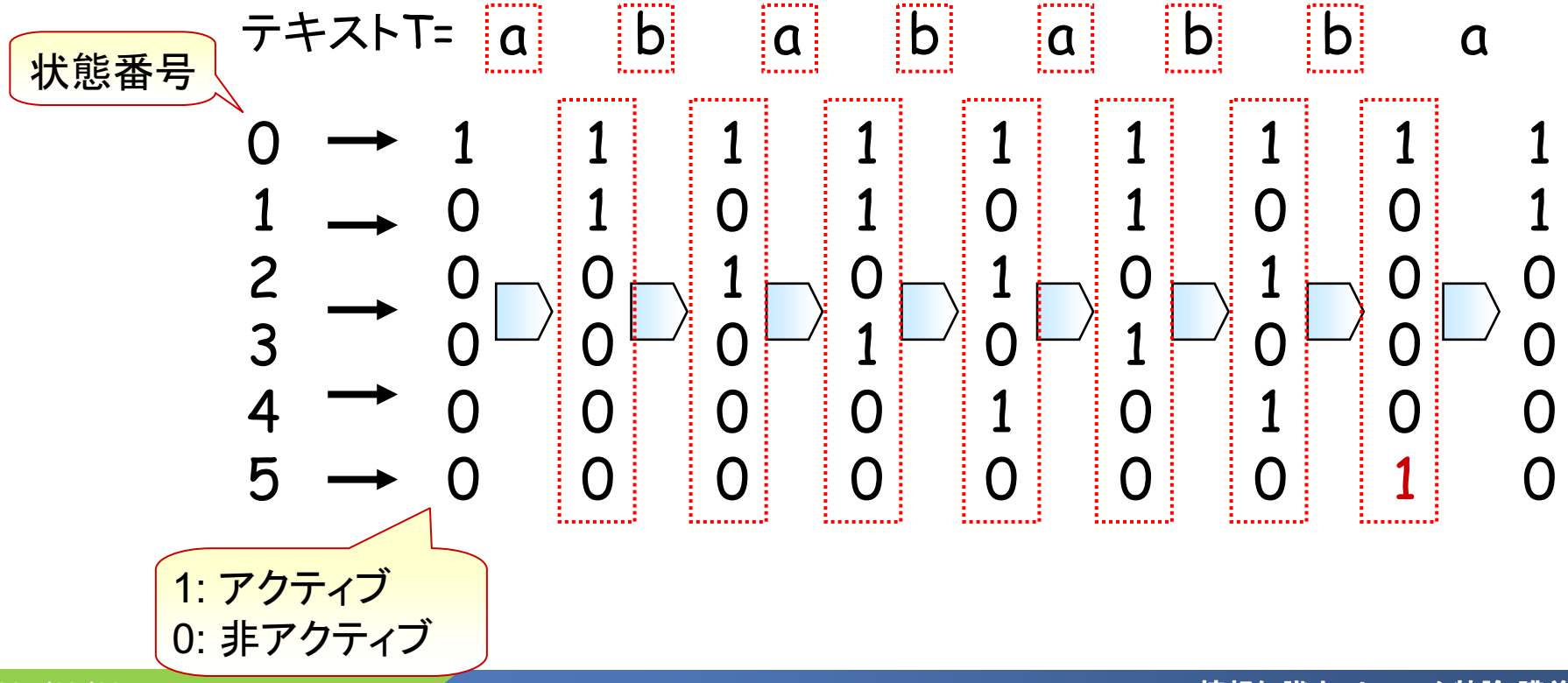
このNFAを
シミュレートする



NFAの動き（状態遷移の様子）

パタン P = ababb を受理する非決定性有限オートマトン





ビットパラレル手法のアイデア

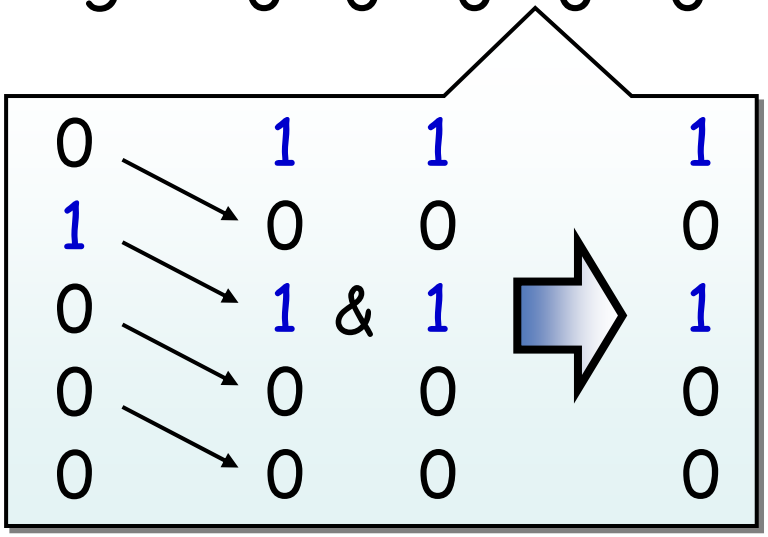
パタン P: a b a b b

テキスト T: a b a b a b b a

1 →	0	1	0	1	0	1	0	0	1
2 →	0	0	1	0	1	0	1	0	0
3 →	0	0	0	1	0	1	0	0	0
4 →	0	0	0	0	1	0	1	0	0
5 →	0	0	0	0	0	0	0	1	0

Mask table M

	a	b
a	1	0
b	0	1
a	1	0
b	0	1
b	0	1



$$R_i = (R_{i-1} << 1 \mid 1) \,\&\, M(T[i])$$

O(1)時間で計算可能

※つまり、マスクビット列Mと「&」をとることで、正しい遷移だけを残している！

擬似コード

Shift-And (P, T)
1 m ← length[P].
2 n ← length[T].
3 Preprocessing:
4 **for** c ∈ Σ **do** M[c] ← 0^m .
5 **for** j ← 1 to m **do** M[P[j]] ← M[P[j]] | $0^{m-j}10^{j-1}$.
6 Searching:
7 R ← 0^m.
8 **for** s ← 1 **to** n **do**
9 R ← ((R << 1) | $0^{m-1}1$) & M[T[s]];
10 **If** R & 10^{m-1} ≠ 0^m **then** report an occurrence at s.

パターンが出現するかどうかの判定も
ビット演算で高速に処理できる！

パターン長がマシン語長（マシンのレジスタ長）
より短い場合には、O(m+|Σ|) 時間・領域の前
処理の後、O(n) 時間で照合できる。

文字クラスへの拡張

パターン P: a b [a b] b b

テキスト T: a b a b b b b a

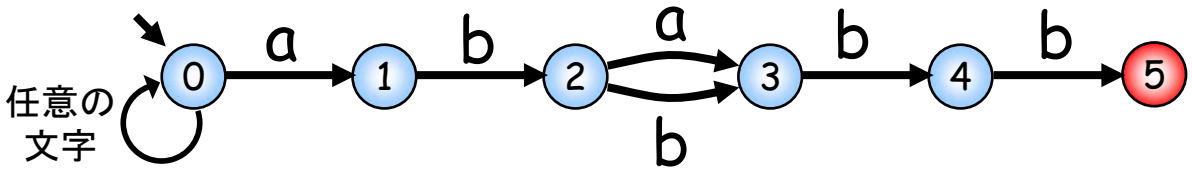
1 →	0	1	0	1	0	0	0	0	1
2 →	0	0	1	0	1	0	0	0	0
3 →	0	0	0	1	0	1	0	0	0
4 →	0	0	0	0	1	0	1	0	0
5 →	0	0	0	0	0	1	0	1	0

Mask table M

	a	b
a	1	0
b	0	1
[ab]	1	1
b	0	1
b	0	1

これだけ！

パターン P = ab[ab]bb を受理する非決定性有限オートマトン



ここは同じ

$$R_i = (R_{i-1} << 1 \mid 1) \ \& \ M(T[i])$$

※ 文字種[a..z,0..9]やその否定[^]、任意文字"!"が扱える

Shift-Or アルゴリズム

- Shift-Andにおけるビット列を反転させたもの
  - 利点： Shift-Andよりも演算が少なくなる

パタン P: a b a b b

テキスト T: a b a b a b b a

1 →	1	0	1	0	1	0	1	1	0
2 →	1	1	0	1	0	1	0	1	1
3 →	1	1	1	0	1	0	1	1	1
4 →	1	1	1	1	0	1	0	1	1
5 →	1	1	1	1	1	1	1	0	1

Mask table M

	a	b
a	0	1
b	1	0
a	0	1
b	1	0
b	1	0

$$R_i = (R_{i-1} << 1) \mid M(T[i])$$

※ チャレンジ！：この場合の擬似コードはどのようになるか？

第2回 まとめ

- Prefix型アルゴリズム
  - Naïveアルゴリズム
    - O(mn)時間で照合、大きなテキストやテキスト・ストリームには不向き
  - KMPアルゴリズム
    - O(m)時間・領域の前処理の後、O(n) 時間で照合
  - ACアルゴリズム
    - KMPオートマトンの考えを複数パターンに拡張したもの
    - O(|Σ|m)時間・領域の前処理の後、O(n) 時間で照合（ここで、mはパターン長の総和）
  - Shift-And/Orアルゴリズム（ビットパラレル手法）
    - 非決定性のKMPオートマトンを基にした考え方
    - 文字クラスへの拡張が容易
    - パターンが短いときにはO(m+|Σ|)時間・領域の前処理の後、O(n)時間で照合
- 次回のテーマ
  - Suffix型アルゴリズム：より効率のよいパターン照合のための工夫

ビットパラレルとは何か？ 何ができるのか？

- （レジスタ長の）ビット列に対する演算の並列性を利用して計算を高速化する手法

例：

	0011	0101	1100	1001	= <3, 5, 12, 9>
&	1000	1000	1000	1000	= <8, 8, 8, 8>
	0000	0000	1000	1000	= <0, 0, 8, 8>

「1ビットどうしの論理積を16回分」
「ビット幅4の整数に対するマスク処理を4回分」

定数時間で

- 画像や音声などのマルチメディアデータを高速処理
- 特徴空間のベクトルをコンパクトに表現
- 文字列処理（近似文字列照合、正規表現照合、他）の巧妙な実装による高速化

※ このアイデアは、IntelのMMX・SSEテクノロジーやAthlonの3D Now!テクノロジーにも見られる

ビットパラレル手法の真髄

- 整数の集まりをコンパクトに表現し、一括処理する！
  - 順番に意味のある数字の列
    → n-ビット幅の整数を一つにパック
    一つのレジスタにパック可能な個数は、⌈w/n⌉ 個
    例： 〈3, 5, 12, 9〉 = 〈0011 0101 1100 1001〉
    → MMX・SSEと同じアイデア

w はレジスタ長
（つまり32とか64）

  - 重複がない整数の集合
    → 連続する w 個の整数の集合を一つにパック
    例： { 3, 5, 9, 12 } = 〈0000 1001 0001 0100〉
    → 集合の演算 ⇔ ビット列の論理演算
    例： { 3, 5, 9, 12 } ∪ { 2, 5, 8, 9, 15 } = { 2, 3, 5, 8, 9, 12, 15 }

ビット幅が16の場合、
[1,16]でも[0,15]でも可

$$\begin{array}{rr} & \langle 0000\ 1001\ 0001\ 0100\rangle \\ \Leftrightarrow & \text{or } \langle 0100\ 0001\ 1001\ 0010\rangle \\ \hline & \langle 0100\ 1001\ 1001\ 0110\rangle \end{array}$$

各整数は、ビットが
立つ位置と対応する

長いビット列を扱う場合

- 2倍長の加算と減算
  - $(z_1, z_0) \leftarrow (x_1, x_0) + (y_1, y_0)$

$z_0 \leftarrow x_0 + y_0$
$c \leftarrow (z_0 < x_0)$
$z_1 \leftarrow x_1 + y_1 + c$

  - $(z_1, z_0) \leftarrow (x_1, x_0) - (y_1, y_0)$

$z_0 \leftarrow x_0 - y_0$
$b \leftarrow (x_0 < y_0)$
$z_1 \leftarrow x_1 - y_1 - b$

ここでも、すべて unsigned

- 2倍長のシフト（$0 \leqq n \leqq w$）
  - $(y_1, y_0) \leftarrow (x_1, x_0) \ll n$

$y_1 \leftarrow x_1 \ll n \mid x_0 \gg (w - n)$
$y_0 \leftarrow x_0 \ll n$

  - $(y_1, y_0) \leftarrow (x_1, x_0) \gg n$

$y_0 \leftarrow x_0 \gg n \mid x_1 \ll (w - n)$
$y_1 \leftarrow x_1 \gg n$

参考：「ハッカーのたのしみ -本物のプログラマはいかにして問題を解くか」ヘンリー・S・ウォーレン, Jr.: “Hacker’s Delight”訳本