配列解析アルゴリズム特論
文字列探索・比較（1）

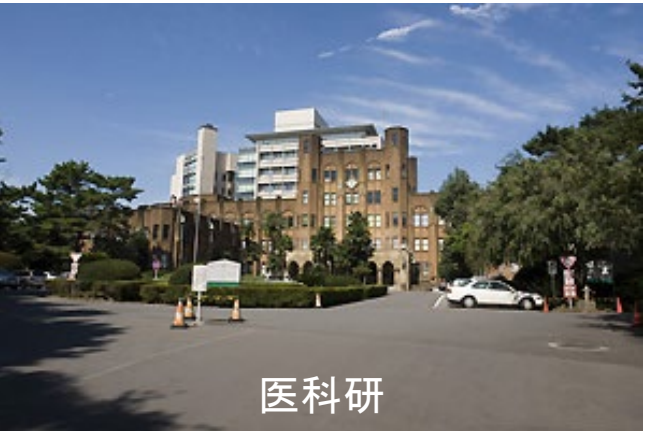
渋谷

東京大学医科学研究所ヒトゲノム解析センター
（兼）情報理工学系研究科コンピュータ科学専攻
http://www.hgc.jp/~tshibuya

自己紹介

- 東京大学医科学研究所ヒトゲノム解析センター（HGC）
  - 白金台キャンパス、地下鉄南北線・三田線白金台駅すぐ
- 専門
  - バイオインフォマティクス・アルゴリズム
  - 検索アルゴリズム
  - プライバシー保護アルゴリズム
  - 学習アルゴリズム
  - それらの医学・生物学への応用

HGC
スパコン
「SHIROKANE」



医科研

HGC

研究室からの風景

この授業について

- 文字列アルゴリズムを中心に様々な話をしていく予定
  - 前半：　基本的な文字列アルゴリズム
    - 文字列比較・探索アルゴリズム
    - 文字列索引・検索アルゴリズム
  - 後半：それらの応用・関連アルゴリズム
    - 圧縮アルゴリズム
    - 学習アルゴリズム
      - クラスタリング
      - 系統樹
      - 教師あり機械学習
    - 実際の生物配列解析のアルゴリズム
- 日程等
  - 授業の進行状況にもよりますが、最後の7/12 は予備としてレポート作成日とするかも
  - 評価はレポート（予定）

教科書 (1/3)

文字列関連アルゴリズム

- 岡野原大輔、高速文字列解析の世界、岩波書店、2012.
- D. Adjeroh, et al., The Burrows-Wheeler Transform, Springer, 2010.
  - この2冊は、BW変換等の最近の話題が詳しい
- W. Sung, Algorithms in Bioinformatics, CRC Press, 2009.
  - バイオインフォマティクスのアルゴリズムに焦点を当てた本の中ではあるが、新しい内容も含み、かなりよい本。
- D. Gusfield, Algorithms on Strings, Trees and Sequences, Cambridge Press, 1997.
  - この分野の**昔の**バイブル**だった**本。載っているアルゴリズムの多くは完全に時代遅れになってしまったが、問題設定等は今でも参考になる。
- V. Mäkinen et al., Genome-Scale Algorithm Design, Cambridge, 2015.
  - 次世代シークエンサー解析がらみのアルゴリズムの本
- D. Salomon, G. Motta, Handbook of Data Compression, Springer, 2010.
  - 圧縮アルゴリズムを概観できる。分厚い。

教科書 (2/3)

- バイオインフォマティクス全般
  - 後藤（訳）、A. Polanski and M. Kimmel（著）、バイオインフォマティクス、シュプリンガー・ジャパン、2010.
    - 様々なバイオインフォマティクスの問題を網羅している。最新の情報も多く含む。
  - H-J. Bockenhauer, D. Bongartz, Algorithmic Aspects of Bioinformatics, Springer, 2010.
    - どんな問題があるのかを知ることができる本。
  - バイオインフォマティクス事典、共立出版、2006.
    - 少し古いのは否めないが、この分野を概観するのにうってつけの事典。
  - 渋谷、坂内（訳）、N. C.Jones, P. Pevzner（著）、バイオインフォマティクスのためのアルゴリズム入門、共立出版、2007.
    - バイオインフォマティクスにおけるアルゴリズム的な問題を知ることができる本。
  - Eidhammer, et al. Protein Bioinformatics: An algorithmic approach to sequence and structure analysis, Wiley, 2004.
    - タンパク質の配列・構造に対するアルゴリズムに焦点をあてた本

教科書 (3/3)

- 分子生物学教科書（例えば）
  - A. Johnson et al., Molecular Biology of the Cell, 6th edition, Grand Science, 2014.
    - 生物系学科の定番教科書。分厚い
  - Essential細胞生物学 第3版、南江堂、2011.

本日の話題： まずは基本からおさらい！

- 文字列探索（照合）アルゴリズムのいろいろ
  - Brute-force algorithm
  - Knuth-Morris-Pratt algorithm
  - Colussi algorithm
  - Aho-Corasick algorithm
  - Boyer-Moore algorithm
  - Horspool algorithm
  - Turbo-BM algorithm
  - Rabin-Karp algorithm
  - Shift-Or method
  - etc.

文字列探索(照合)とは

- 問題
  - テキスト文字列の中に与えられた文字列(パタン)があるかどうか、あればどこにあるか？
    - exact matching:　変異・挿入・削除等は考えない
- 照合と検索
  - 照合:　テキストを頭から順に見ていく　(前処理はパタンのみ)
  - 検索:　テキストに前処理を施す(索引の作成)ことによって高速に探す

Text

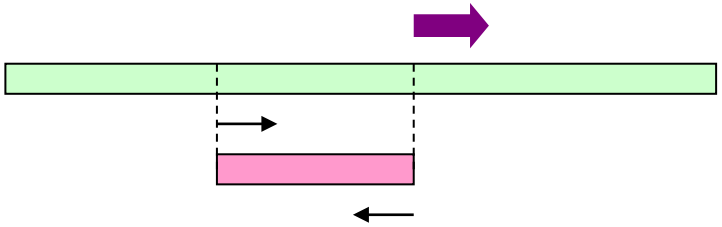
GGTGAGAAGTTATGATACAGGGTAGTTG
TGTCCTTAAGGTGTATAACGATGACATC
ACAGGCAGCTCTAATCTCTTGCTATGAG
TGATGTAAGATTTATAAGTACGCAAATT

Pattern

TATAA

文字列照合の手法の分類

- スキップ系
  - 接頭辞系アルゴリズム　（パタンを前からチェック）
    - Knuth-Morris-Pratt
  - 接尾辞系アルゴリズム　（パタンを後ろからチェック）
    - Boyer-Moore, Horspool, Turbo-BM
- 力任せ系
  - 力任せ(Brute-force)アルゴリズム
  - ハッシュ系アルゴリズム
    - Rabin-Karp
  - ビット計算系アルゴリズム
    - Shift-Or (Shift-And)



Brute-force な文字列探索（1）

ナイーブなアルゴリズム

- すべての位置（ n 箇所）で
- パタンの長さ（ m ）分の比較を行う

$$O(nm)$$

吾輩は猫である。名前はまだ無い。どこで生れたかとんと見当がつかぬ。何でも薄暗

たかとん
　たかとん
　　たかとん
　　たかとん

・・・

たかとん

・・・

Brute-force な文字列探索（2）

- でも、少し考えればランダムテキストの場合には平均で線形時間！
  - ランダムなテキスト（あるいはランダムなパタン）の場合はk文字目をチェックする必要がある確率は $(1 / |\Sigma|)^{k-1}$ にすぎない
  - 実際にテキストがランダムっぽくてプログラムが面倒な時などは、このような方法でもそれほど悪くはない可能性がある、ということ
    - ただし、そうは言っても他の賢い方法と比べるとやはり「かなり」遅い

アルファベットサイズは4

テキスト

GGGACCAAGTTCCGCACATGCCGGATAGAAT

パタン

CCGTATG

この順序でチェック

```
c
 c
  c
   c
    CCg
      ....
          CCGt
            ....
```

平均的にチェックする長さは

$1+1/4+(1/4)^2+\ldots = 4/3$ (constant!)

（ランダムなDNA列の場合）

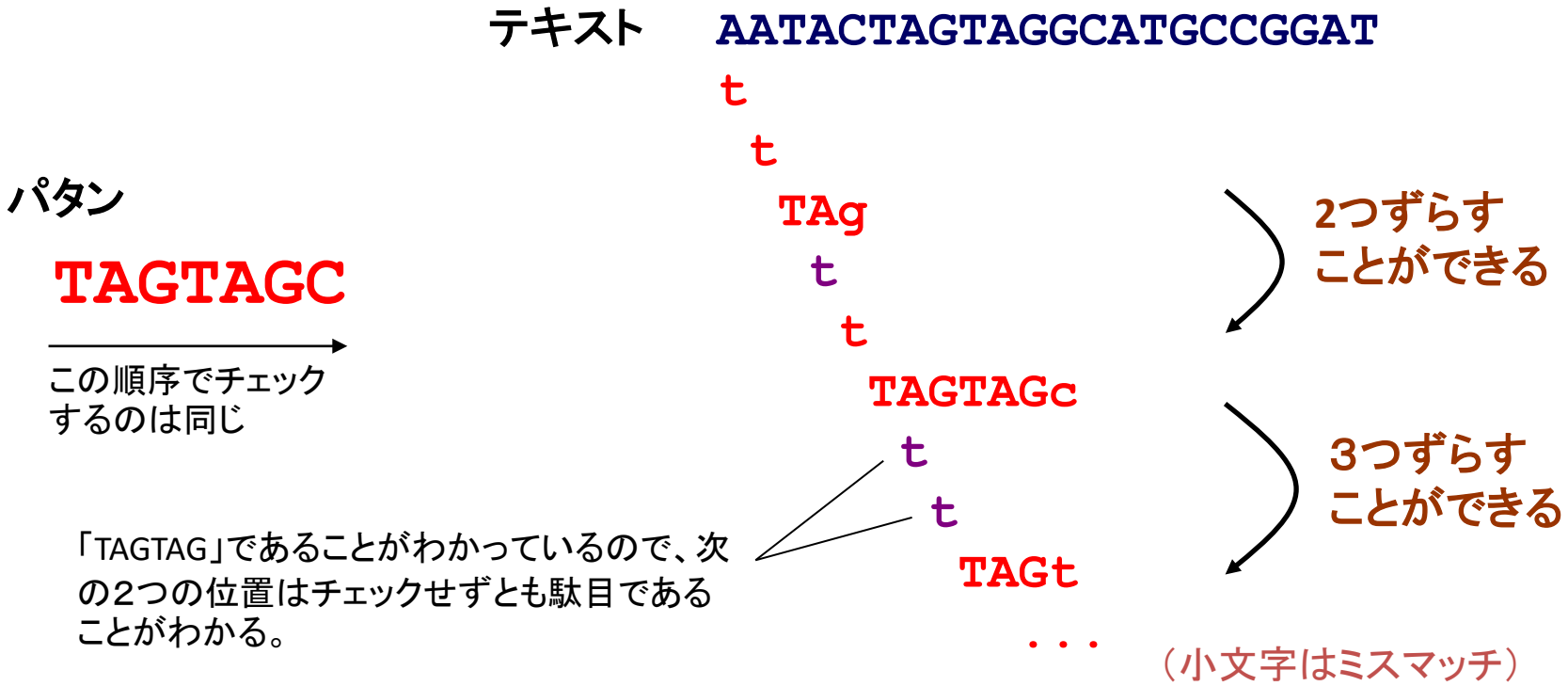
（小文字はミスマッチ）

ということで

- ナイーブでもそれなりに速い
- とはいえ
  - 最悪計算量はあくまで$O(nm)$
    - 「ほとんど」同じ文字列が繰り返しあったりすると当然そのような状況に陥る
- 実は
  - 最悪でも線形時間のアルゴリズムが存在する
    - Knuth-Morris-Pratt
  - しかも、線形時間、というのが最速であるとは限らない
    - 線形時間より速い計算量のアルゴリズムも存在する
      - Boyer-Moore

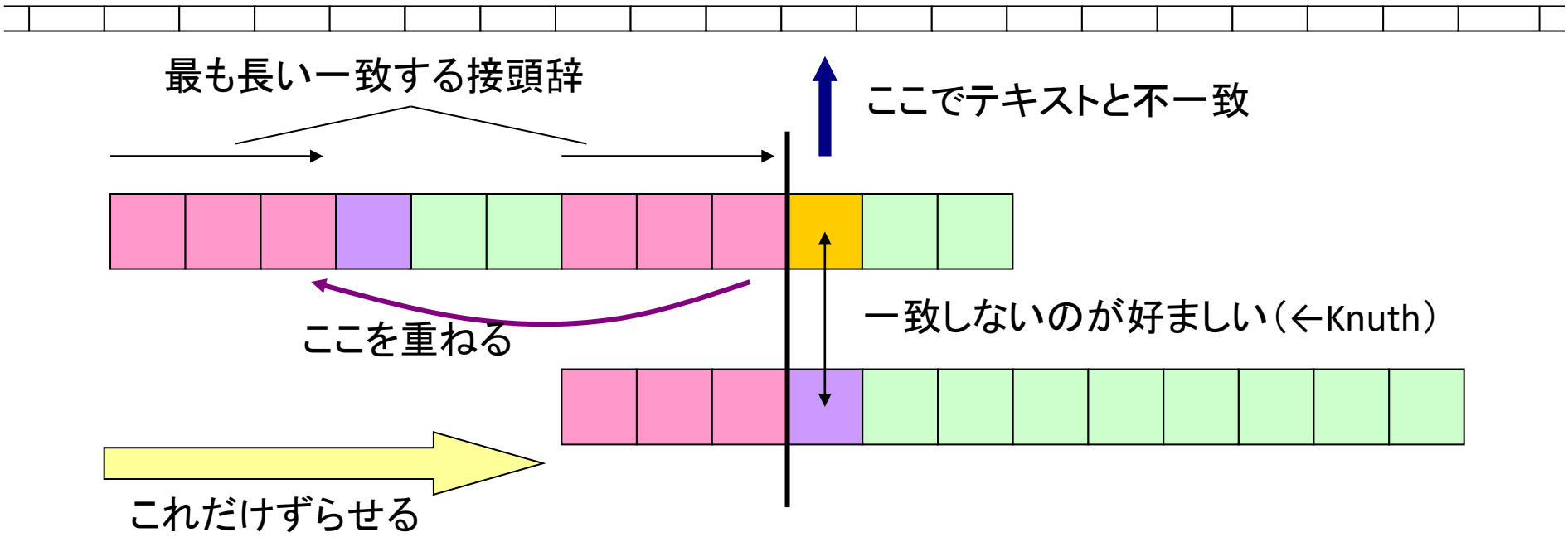
Knuth-Morris-Pratt(1)

- Brute-forceでは、
  - 同じ場所を2度チェックすることも多い
    →無駄！
  - ある場所のチェックの結果次第では、それに続くいくつかの場所のチェックが不要である場合がある！
    → Knuth-Morris-Pratt Algorithm



Knuth-Morris-Pratt (2)

- $P[0..i]$までマッチして、$P[i+1]$でマッチしなかった際にずらせる量
  - i - (max j s.t. $P[0..j] \equiv P[i-j..i]$, $P[j+1] \neq P[i+1]$, $j < i$)
  - そのような j がなければ
    - $P[j+1] \neq P[i+1]$ならば $i+1$
    - $P[j+1] = P[i+1]$ならば $i+2$ずらすことが可能
  - パタンのみで事前に計算可能



Knuth-Morris-Pratt (3)

テキスト

CTACTGATCTGATCGCTAGATGC

パタン

CTGATCTGC

Tで始まるものはないので、2文字まるまるシフト。

CTGATCTGC

一致するものがないので次。

CTGATCTGC

「CTG」を重ねるようにシフト。

CTGATCTGC

CTGATCGC

Morris-Pratt では、「C」を重ねるように5文字シフトする。

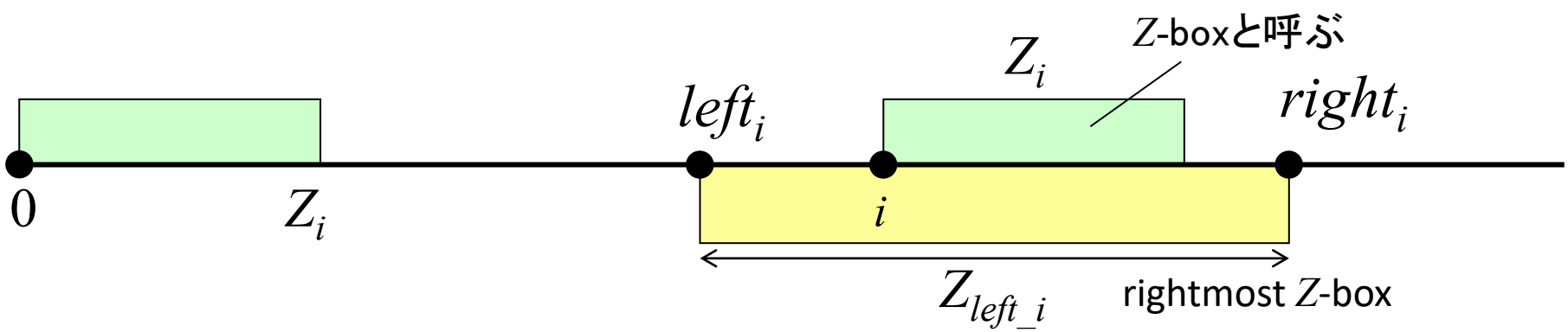
KMPでは、テキスト上でCの次が「Tでない」ことがわかっていることを利用して、6文字まるまるシフトする。

Knuth-Morris-Pratt (4)

- パタンに対する前処理
  - 力任せで計算すると$O(m^3)$
    - 少し頑張ればすぐ2乗のオーダーくらいにはなるが、、、
  - 実は線形時間アルゴリズムが存在
    - KMPそのものを用いる
      - パタン自身をテーブルを作りながら探索する
    - Z アルゴリズム [Gusfield 97]
      - 上のKMPを使う方法の方が速いが、Boyer-Mooreの前処理でも使えるので便利。

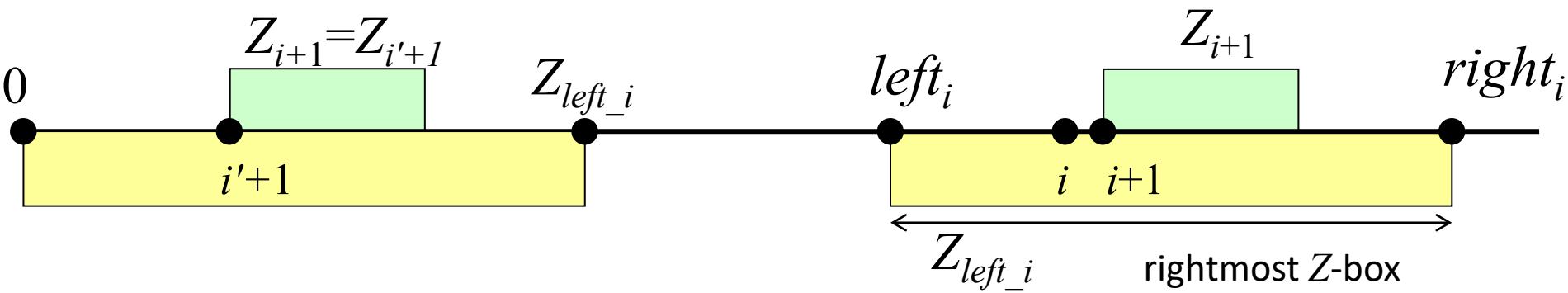
Z Algorithm (1)

- Z_i （すべての i （$i>0$）について順番に計算していく）
  - $S[0..n\text{-}1]$ と $S[i..n\text{-}1]$ の最大共通接頭辞長
- $right_i$
  - $x \le i$ における $x+Z_x\text{-}1$ の最大値（= 右側が最も右側の Z-boxの右側）
- $left_i$
  - $x \le i$ において $x+Z_x\text{-}1$ が最大値をとる x （= 右側が最も右側の Z-boxの左側）
- 初期状態
  - $right_0=left_0=0$としてZ_1から順番に計算していく。
  - $right_i$ と $left_i$ は Z_i の計算後に計算 （Z_i の計算は次のスライド）
    - もちろん定数時間でupdate可能



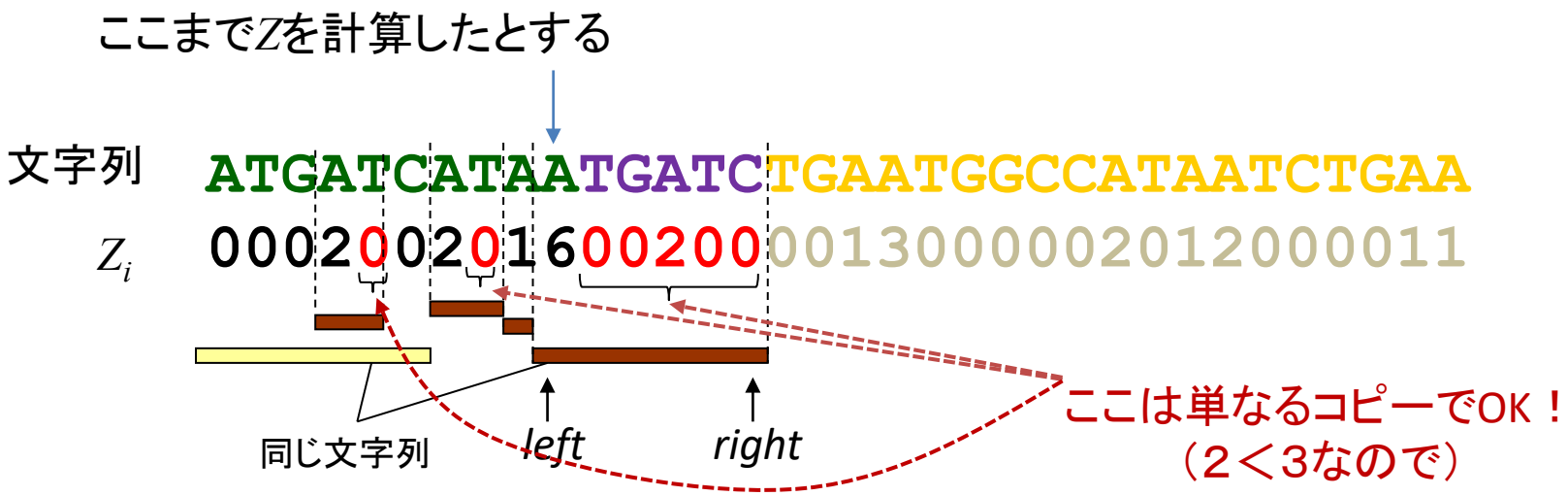
Z Algorithm (2)

- Z_{i+1}の計算
  - $i+1 \le right_i$ の場合
    - $right_i$ まではすでに計算したことがある！
      - Z_i が $right_i - i$ より短い場合は$O(1)$ — この場合は当然ながら全体で線形時間
      - 長い場合は$right_i$ から後ろをナイーブに計算 — ①
  - $i+1 > right_i$ の場合
    - 必要な分だけナイーブに計算 — ②
  - ①と②をあわせて、全体で線形時間！
    - ナイーブ計算は$right_i$ より右側で行われ、しかも$right_i$ は必ずナイーブ計算した後、その終わった場所にリセットされる（単調増加）
- 実はこれ自体も線形時間探索アルゴリズム
  - パタン P とテキスト T を連結した $P\$T$ に対して Z_i を計算すればよい
    - $ は P や T に出てこない新しい文字



Z Algorithm (3)

例

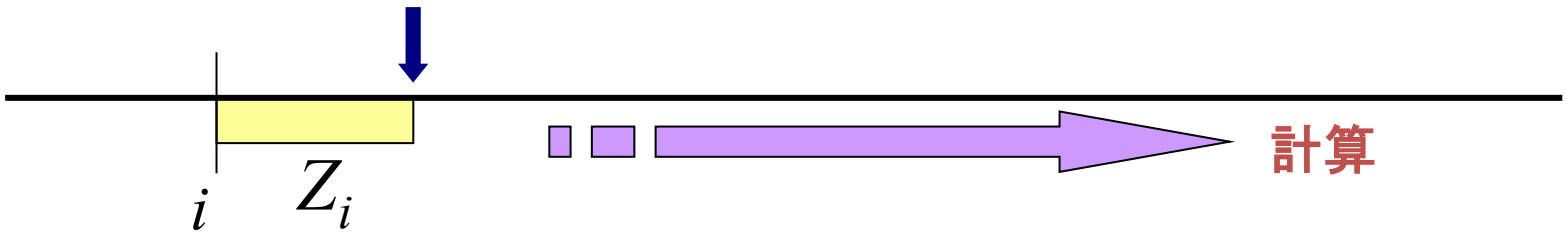


Z_i から、「重ねる」量を求める

- 重ねる長さ: その場で終わる最大のZ-boxのサイズなので、Z_iをスキャンするだけで計算可能

初期状態として 0 をセット後、左から順に、

if ($Overlap[i+Z_i] = 0$) $Overlap[i+Z_i] = Z_i$



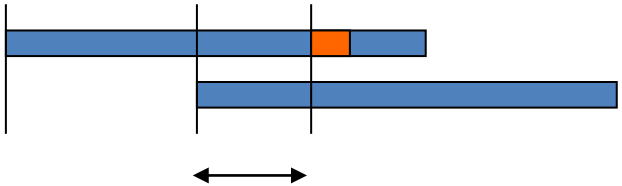
パタン文字列	GTAGGCATGTAGCGTAA
i	0123456789.....
Z_i	00011000400103000
unmatch時に重ねる量	00001100000040003
	GTAGG...

$Overlap[12] = 4$　なので、パタン上の位置12でマッチングに失敗すると、4つ重なる場所に動かす、すなわちテキスト上で12-4=8つ進める

後処理（Knuth's Trick）

- unmatch時に重ねる長さが0の場合、その字が最初の字と同じであれば、さらに1文字動かすことが可能
  - その場所はもう比べても無駄であることがわかっているため

KMPで次の位置に移動する際に重ねる長さ



↓では $Overlap[3] = 0$　だが、パタン上の位置3でマッチングに失敗したとき、Pattern[3]=Pattern[0]ならば、テキスト上で、3-0=3ではなく、3-0**+1**=4つ進めてよい。

パタン文字列	GTAGGCATGTAGCGTAA
i	0123456789.....
Z_i	00011000400103000
unmatch時に重ねる長さ	00001100000040003

→ GTAGGCATGTAGCGT..

→ GTAGGCATGT..

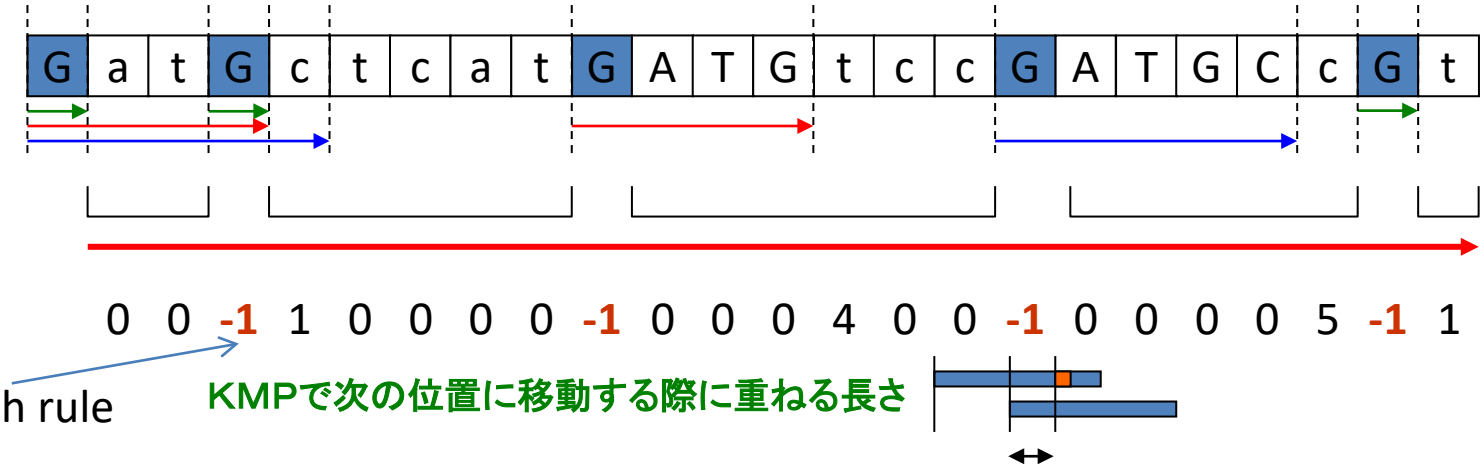
KMPの計算時間

- 計算量
  - 最悪でも線形時間
    - $O(m+n)$
      - n: テキストサイズ
      - m : パタンサイズ
    - 文字の比較回数は $2n$ 回以下
      - 比較が成功
        - » テキスト上のその場所は2度と比較しない（＝最大n回）
      - 比較が失敗
        - » パタンの先頭の位置が必ず進む（＝最大n回）
  - ただ、後述するBoyer-MooreやShift-Orの方が速いことが多い

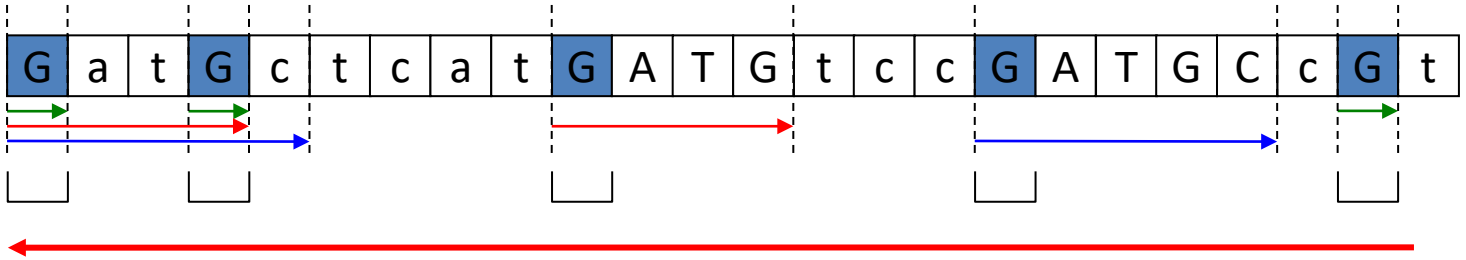
Colussi Algorithm (A Variation of the KMP)

- KMPの比較回数を$3n/2$回以下に減らしたアルゴリズム
  - 重ねる長さが0な場所のうちKnuth ruleが適用される場所の比較を後回しにする
    - 後回しにしたところは後ろからチェックしていく
  - スキップ量はKMPと少し異なる
  - これに対するpreprocessingも線形時間でできる。
  - 実際に速いかどうかはというと、、、、？？
  - 更に$4n/3$まで減らしたアルゴリズムもあり。（Galil−Giancarlo Algorithm）

ステップ１



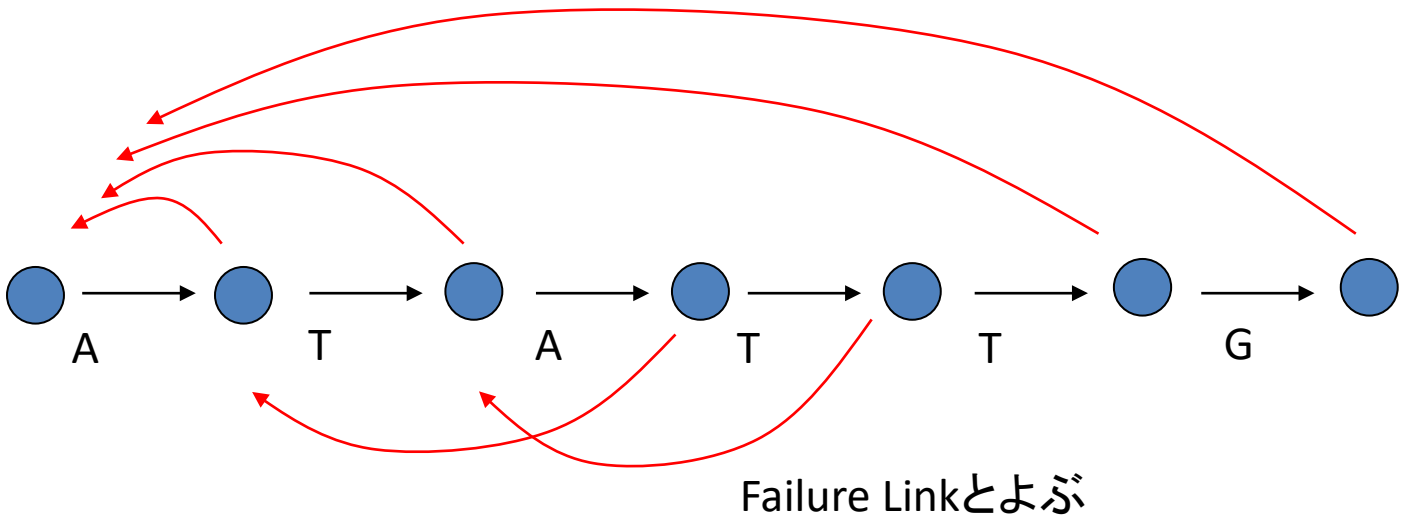
ステップ２



シフト量がKMPとは異なって来ることに注意

Morris-Prattアルゴリズムとオートマトン

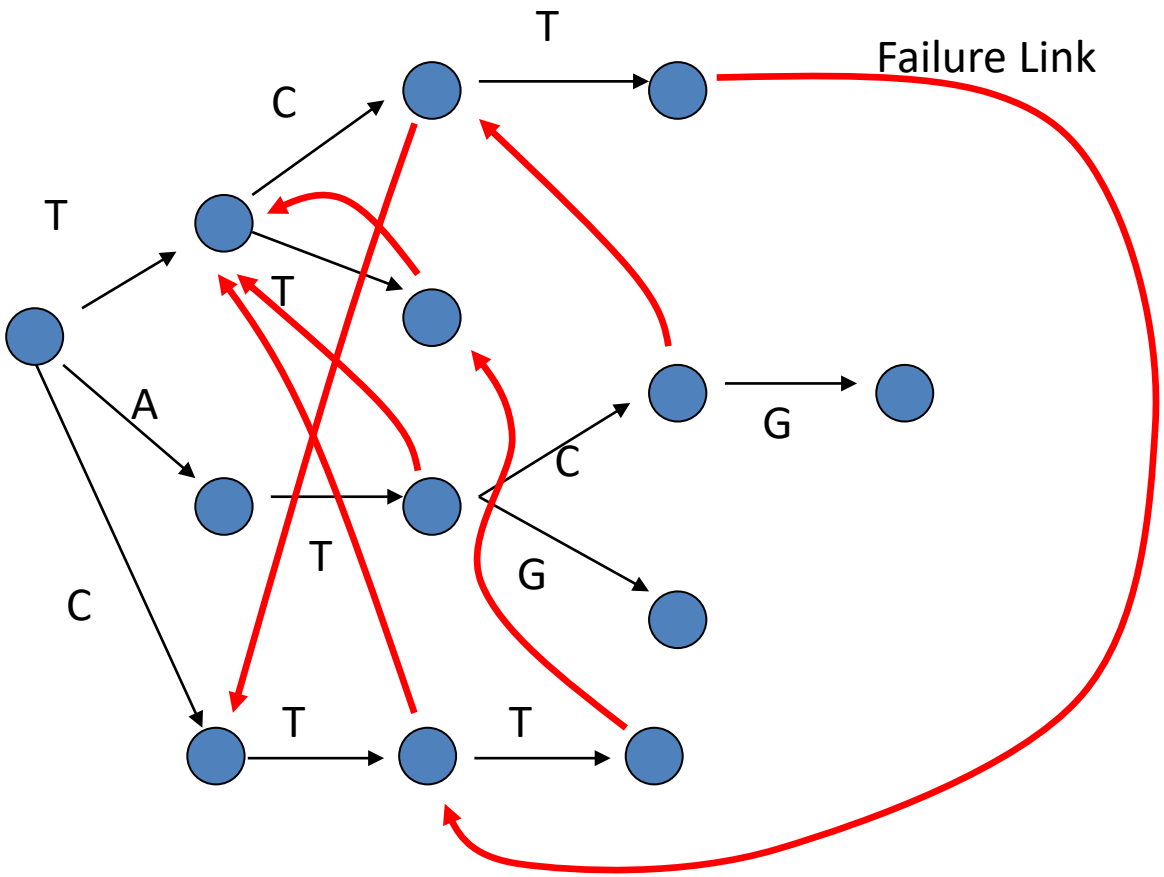
- MPのアルゴリズムはオートマトンで表現できる



赤リンクはテキストの対応文字が黒リンク上の文字でなかった時の遷移

Aho-Corasick (1)

- さらにそのオートマトンを複数パタンに拡張することが可能
  - 線形時間$O(km)$で作成可能！
  - 線形時間探索が可能！

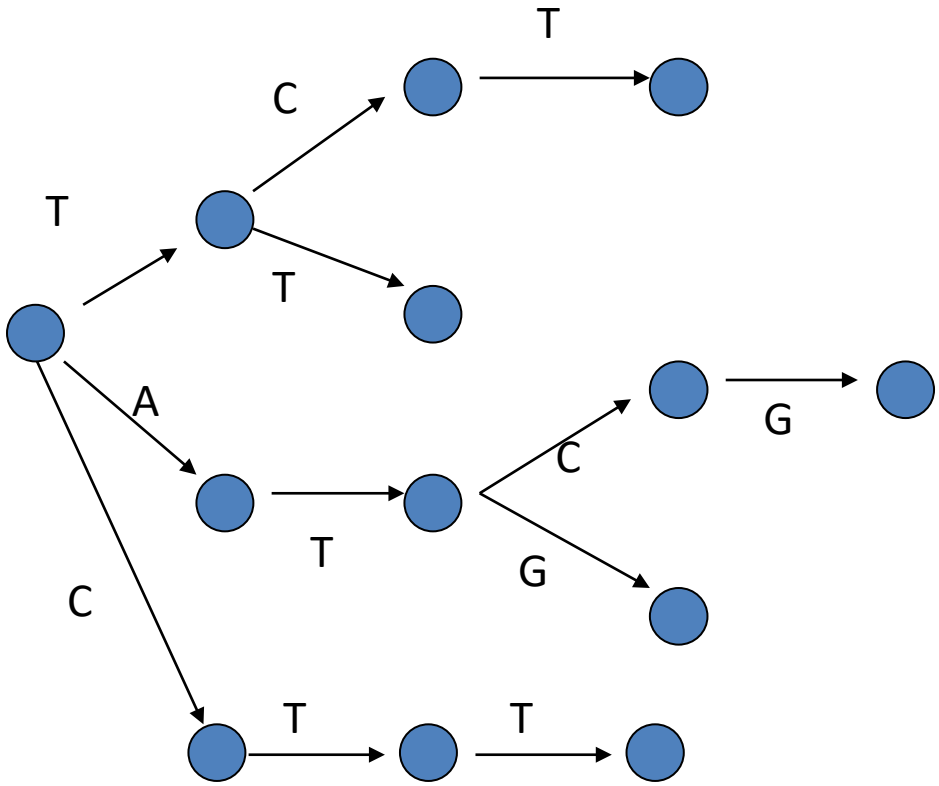


リンクのないものはルートへ

Knuth's rule/trickは考えない

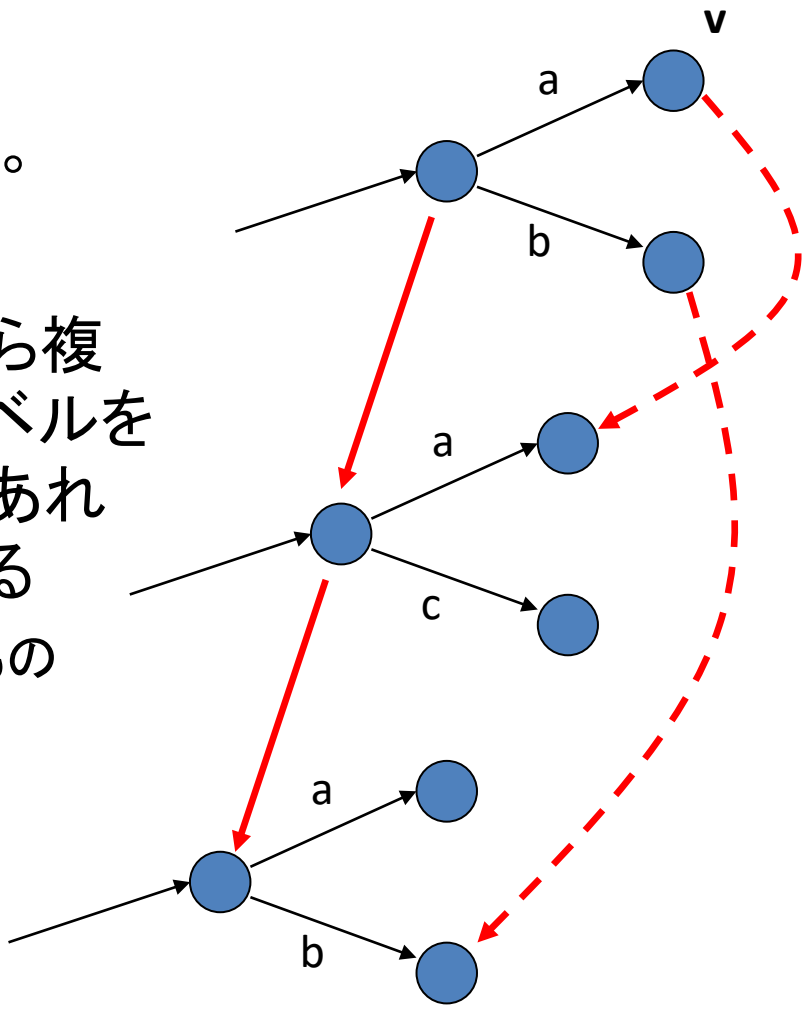
Aho-Corasick (2)

- まずkeyword treeを作成
  - アルファベットサイズが固定ならば線形時間 $O(M)$
    - M: パタン文字列の長さの和
  - 辞書検索にはこれで十分。



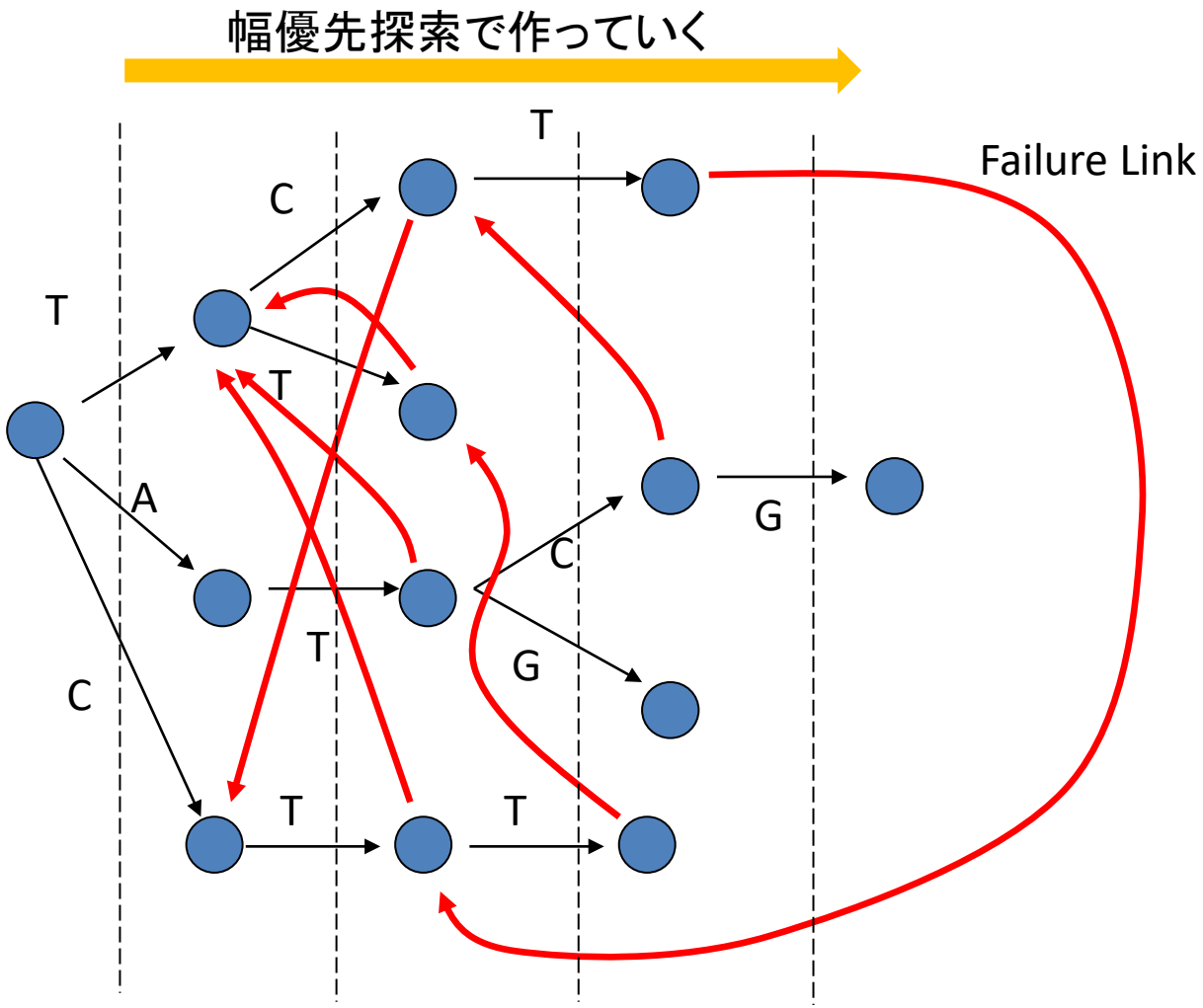
Aho-Corasick (3)

- 幅優先探索で次のように決定
  - rootから開始
    - rootにはfailure link はなし。
  - FailureLink(v)
    - 親のFailureLinkを（必要なら複数回）辿った際、vと同じラベルを持つ子供wを持つノードがあれば、wをFailureLink(v)とする
      - 複数ある場合は一番近いもの
    - なければroot



Aho-Corasick (4)

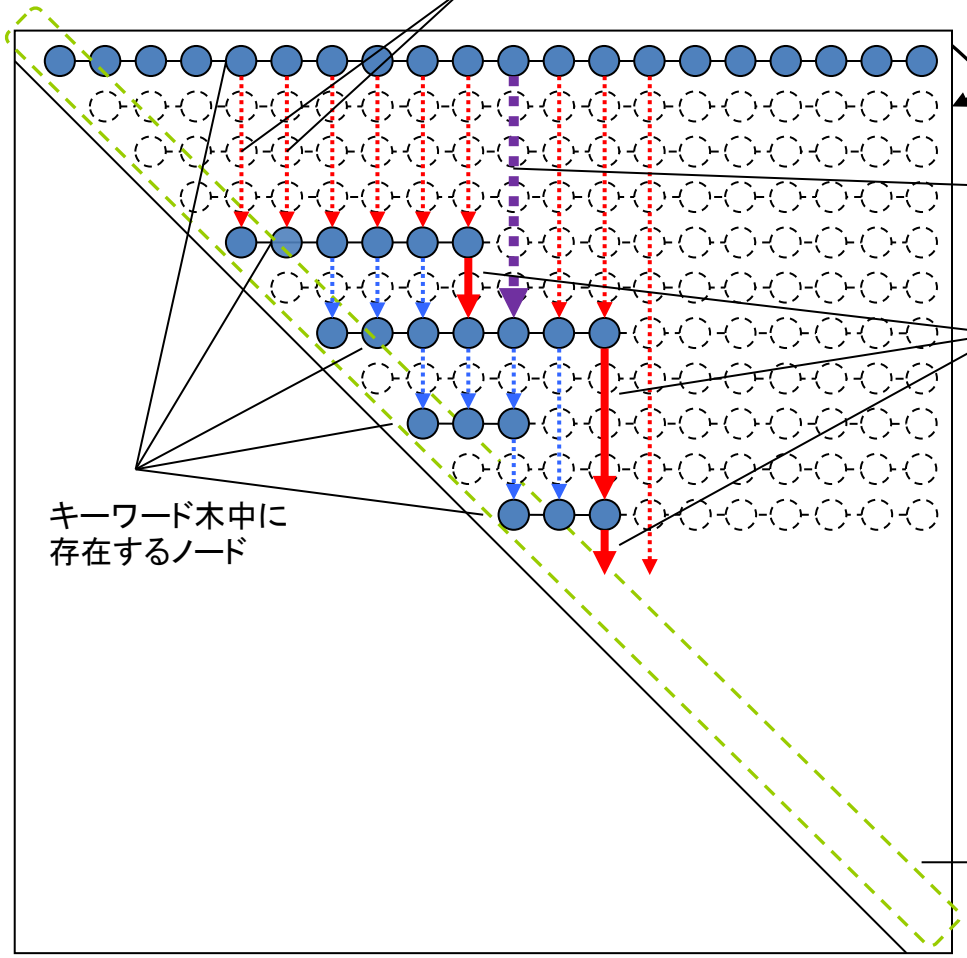
- というわけでこんなのができる



Aho-Corasick (5)

- 何故これが線形時間か？
  - ひとつのパタン（長さm）についてのみの計算量に着目

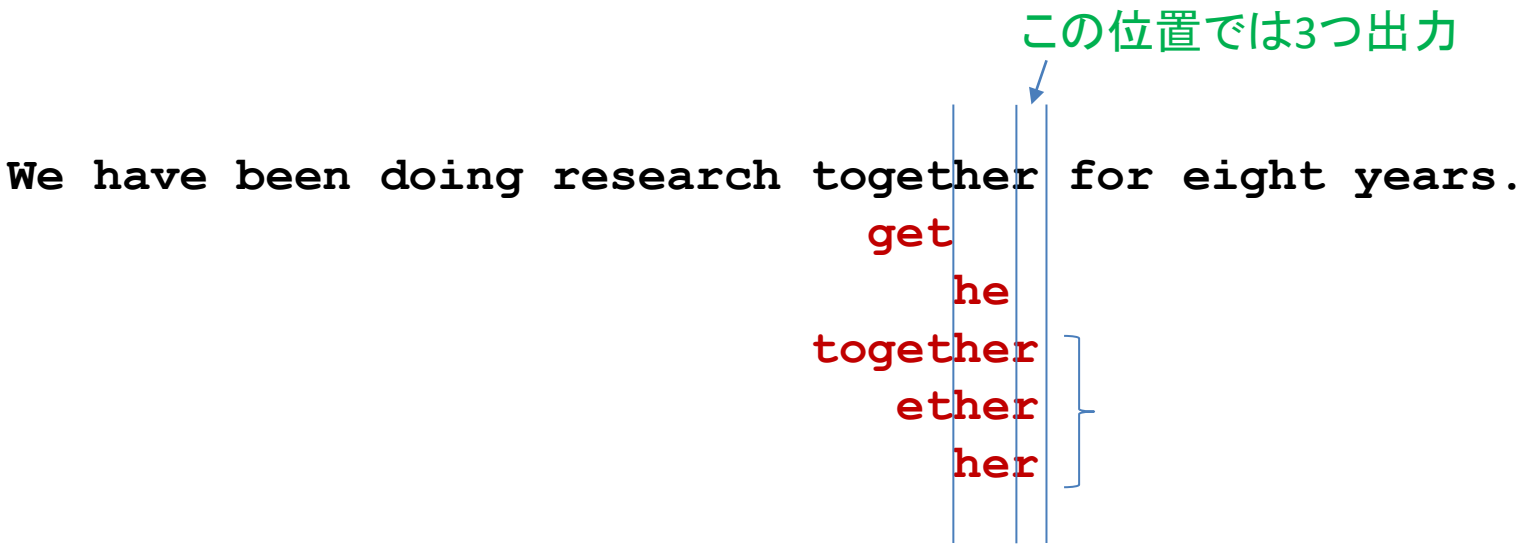
どんなに頑張っても$2m-2$回以上辿ることはない（m:そのパタンの長さ）



ある**長さm**のパタンのすべての suffix を表した図

Aho-Corasick (6)

- (Knuth-)Morris-Platt との違い
  - 同じ位置で、複数の出力をする必要がある場合がある
  - 一つのステートと複数の出力が対応する

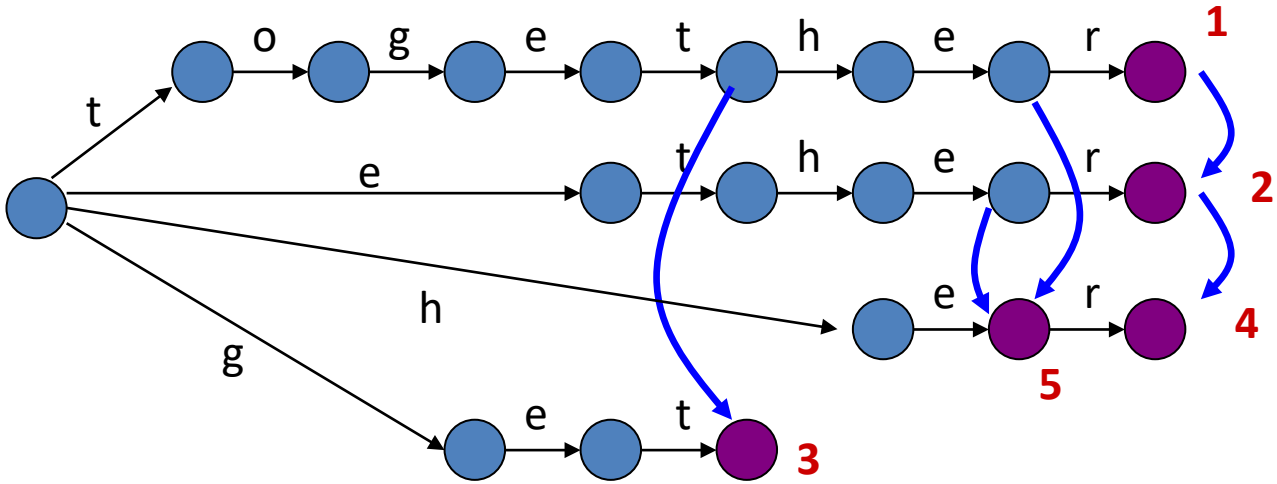


{together, get, he, ether, her}を探す

Aho-Corasick (7)

- OutLink(v) : vが出力すべき文字へのポインタ
  - 実際の探索時には、OutLinkを辿ることで、すべてのマッチする文字列idを出力できる
- OutLinkをみつけるアルゴリズム
  - すべての葉に対応する文字列idを割り当てる
  - それぞれの点から、FailureLink(v)からなる木を探索し、idを持つ祖先のうち一番近いノードへのリンクをセットする
    - Failure Linkからなる木をDFSで探索すればよい → 線形時間
  - なければ出力なし

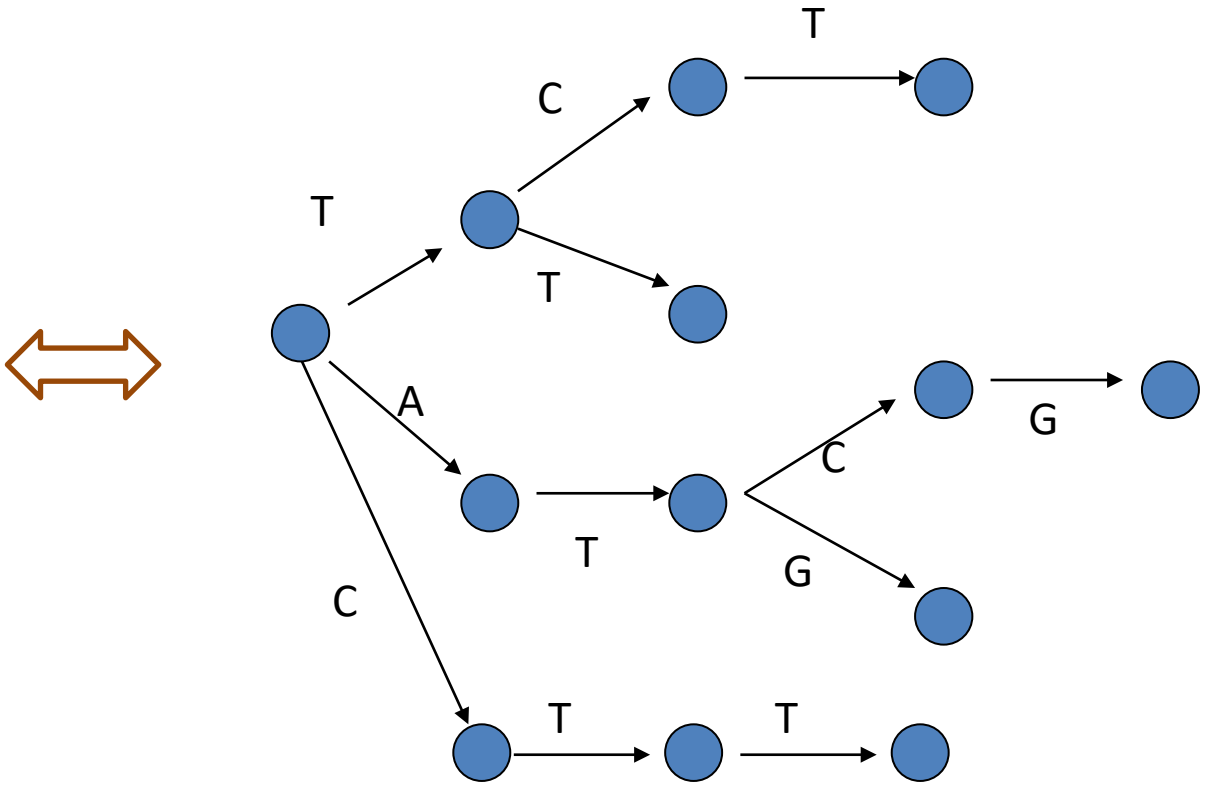
1 together
2 ether
3 get
4 her
5 he



正規表現との関連

- Aho-Corasickは正規表現探索の特殊ケース
  - 一般の正規表現の探索はもう少し計算時間が必要

(T(CT+T))+
(AT(CG+G))+
CTT

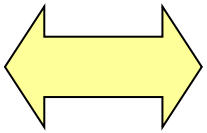


正規表現

正規表現(regular expression)とは

- 連結(concatenation)　A, B → AB
- 論理和(or)　A, B → A+B
- 繰り返し(0回以上、closure)　A → A*

AB(A+B)(AB+CD)*B



ABAB
ABBB
ABAABB
ABACDB
ABBABB
ABBCDB
ABAABABB
ABAABCDB
…

PROSITE

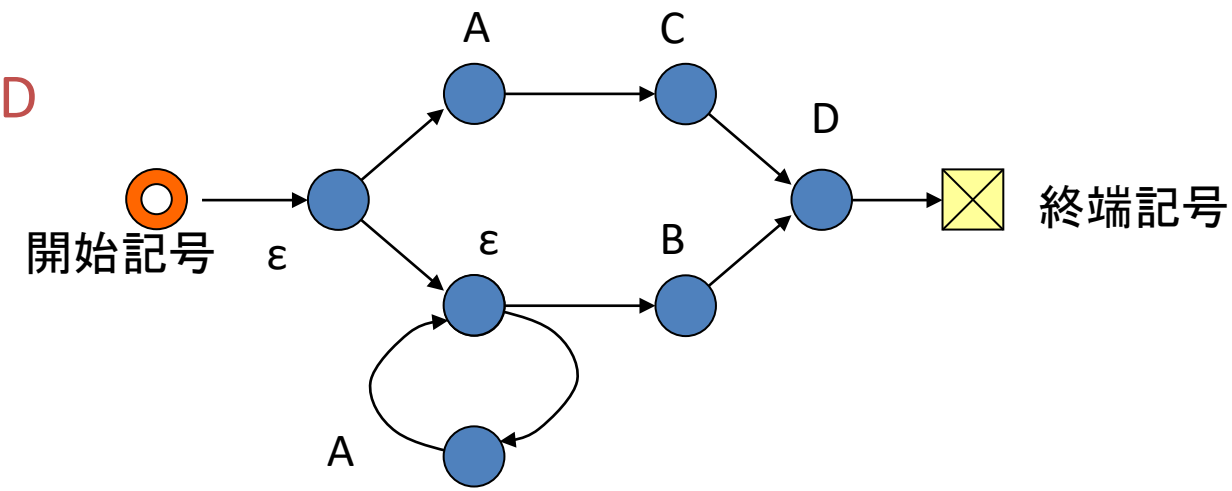
- 正規表現は頻出パタンの表現方法としてもよく用いられる。
- 例：PROSITEデータベース [Fulo et al., NAR, 2004]
  - 次の文字からなる
    - Σ：　文字（アミノ酸）
    - x：　どの文字でもよい
    - −:　連結（省略可）
    - [...]：　その中のどれか
    - {...}:　その中以外のどれか
    - (*n*):　その直前の文字を*n*回繰り返し
    - (*n*,*m*):　その直前の文字を*n*回以上*m*回以下繰り返し
    - <, >:　端にマッチ
    - .:　終了

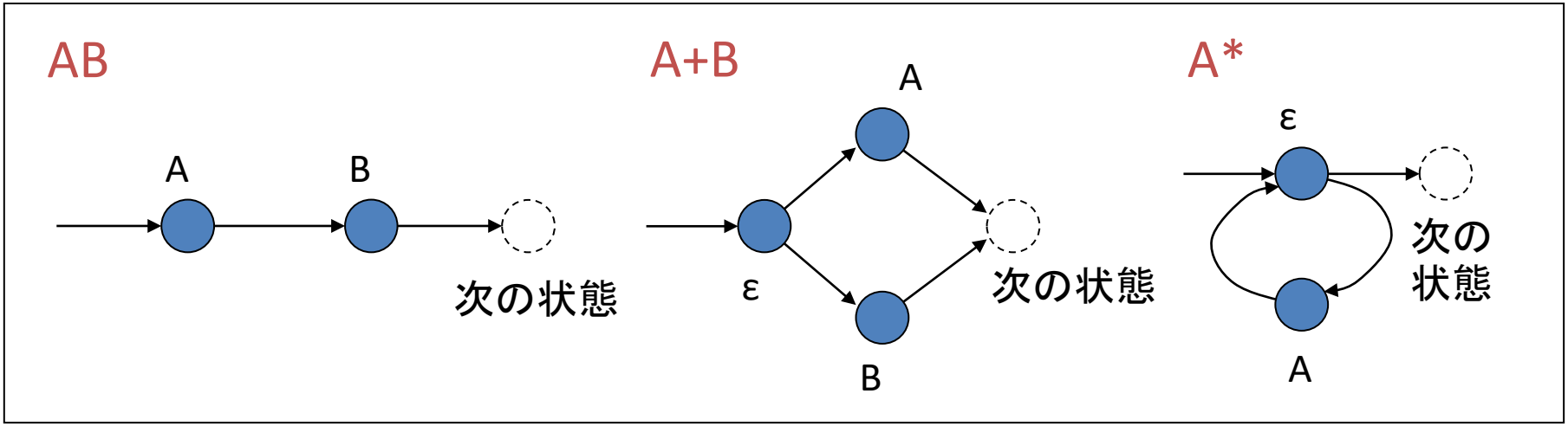
[ILM]-[FY]-K-x(4)-D-x(2,3)-[ILV]-x(1)-P-[KQ]

正規表現とオートマトン

■ 正規表現を表すオートマトンの作成 （線形サイズ）

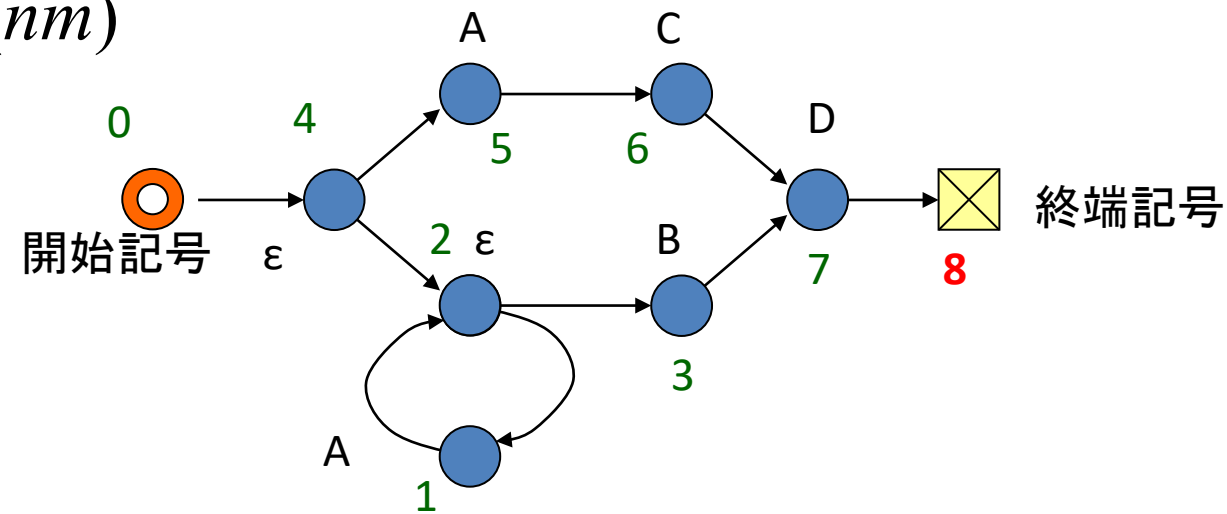
(A*B+AC)D





オートマトンによる正規表現探索

- $O(nm)$



CDAABCAAABDDACDAAC

00000000000000000000 ← いつでも開始可

113 11137 1 11

55 555 567556

8 8 ← 終端記号にたどり着ける＝パタン発見！

到達可能状態の集合

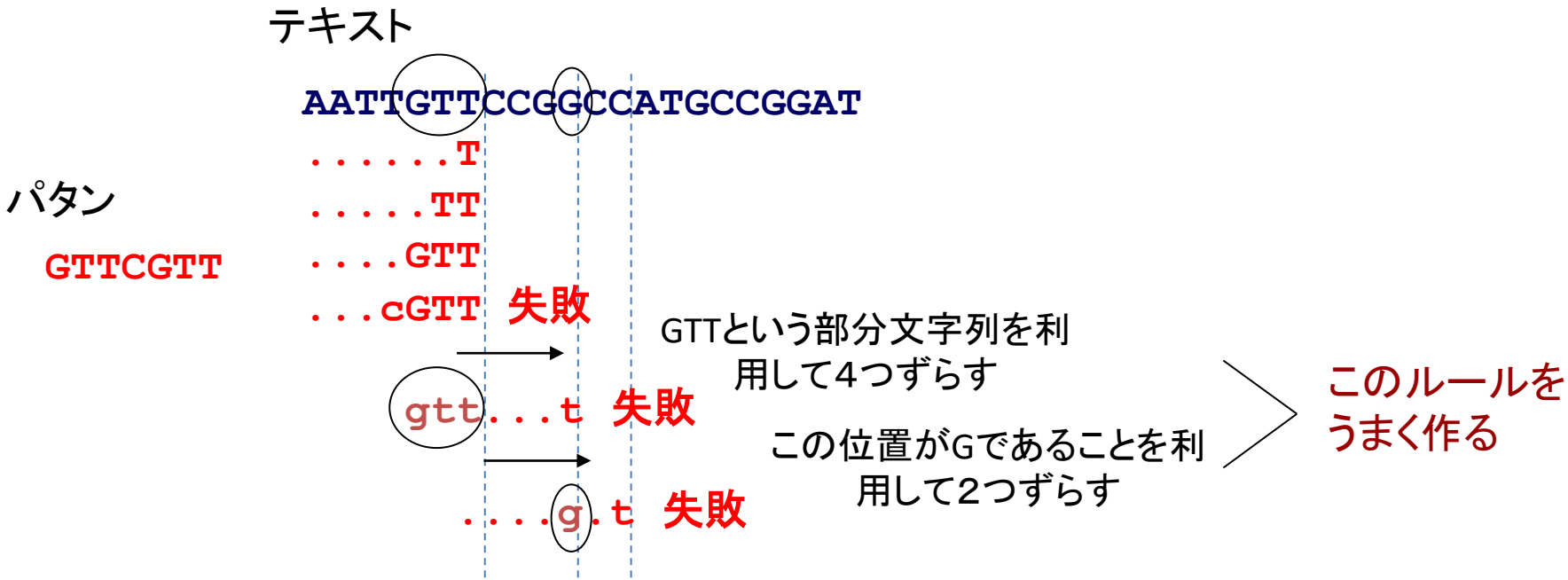
（ただし、この図では、開始、終端以外のε状態は含めていない）

計算の順序

Boyer-Moore (1)

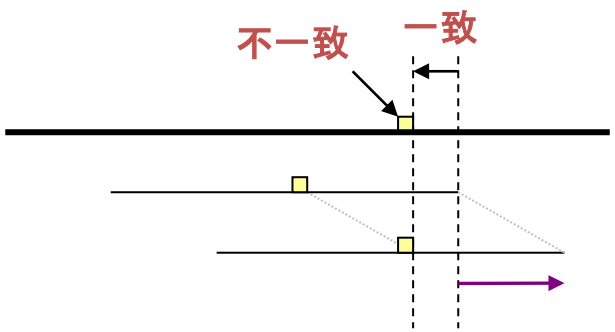
考え方

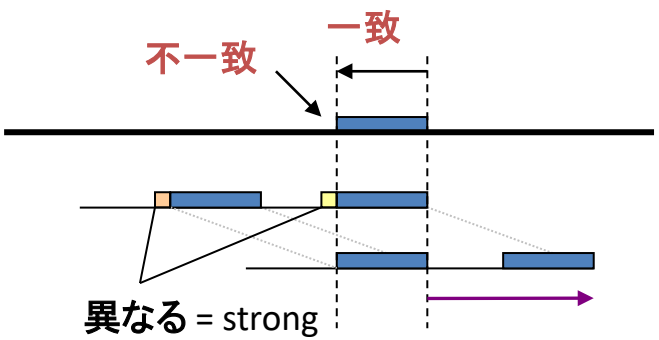
- テキスト上を前から順番にチェックするが、ある位置におけるパタンの出現のチェックを後ろから行う
- マッチしない文字があった場合
  - パタン文字列をいくつかずらす（パタン文字列の情報を利用）



Boyer-Moore (2)

- ずらすための二つのルール
  - 不一致ルール (bad character rule)
    - チェックした文字が*x*であれば、パタンの中の*x*という文字のうち最後の位置までずらせる
      - ただし、パタンの本当の最後の文字は含めない
      - 後戻りしてしまう場合のシフト量は1とする
    - その位置より前のxをすべての位置に関して記憶することも可能
      - シフト量は増えるが、効果は薄い　←　good suffix rule　が同様の効果を持つため
    - アルファベットサイズが大きければ、このルールだけでも高性能
      - *cf.* Horspool Algorithm　はこのルール(の変形版)のみ用いたアルゴリズム
  - 接尾辞一致ルール　((strong) good suffix rule)
    - 後ろk文字が一致した場合、パタン中のそれ以外の場所にそのk文字が存在する(いくつかある場合は最後のもの)時、そこを一致させるようにずらす
      - その前の一文字は異なるもののみを考える(originalではこの条件はなし)
      - パタンの先頭の部分に関しては、その先頭まで一致するだけでよい
  - 両者のうち大きい方をとる





Boyer-Moore (3)

不一致ルールの例

パタン

TTCCAAGTCGCC

この文字は除く

ここでマッチング失敗

テキスト

CCCTGTCCATGCCGTCAGCCC

TTCCAAGTCGCC

TTCCAAGTCGCC

最後のT

Boyer-Moore (4)

接尾辞一致ルールの例

- Knuth-Morris-Prattと考え方はほぼ同じ！

パタン

CGTATATCCAATATC

失敗

テキスト

AGTCCCTCGGTCCGATATCGACCCTCCCG

CGTATATCCAATATC

CGTATATCCAATATC

Boyer-Moore (5)

- 前処理
  - 不一致ルールは簡単
    - アルファベットサイズとの関連
      - 配列で持つ
        - アルファベットサイズが小さい場合
      - ハッシュを使う
        - アルファベットサイズが大きい場合
      - 平衡木を使う
        - アルファベットサイズが大きく、計算量が気になる場合
  - 接尾辞一致ルール
    - 線形時間
    - Zアルゴリズムを後ろ向きにかけることで計算できる
      - ただしパタンの先頭部分についての処理に若干注意が必要

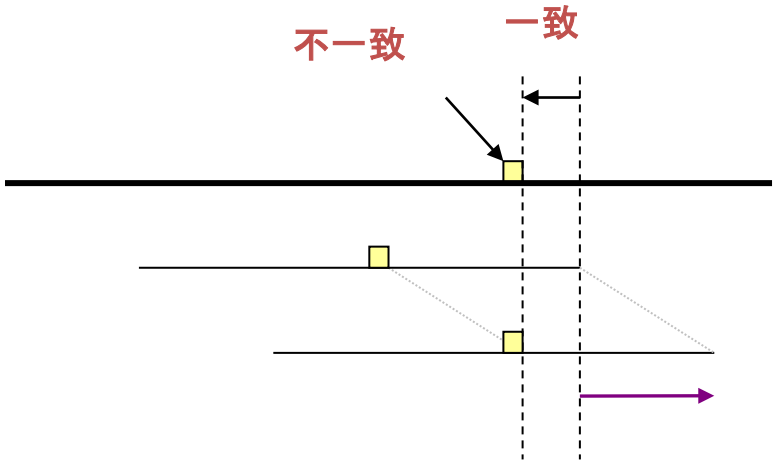
Boyer-Moore (6)

性能

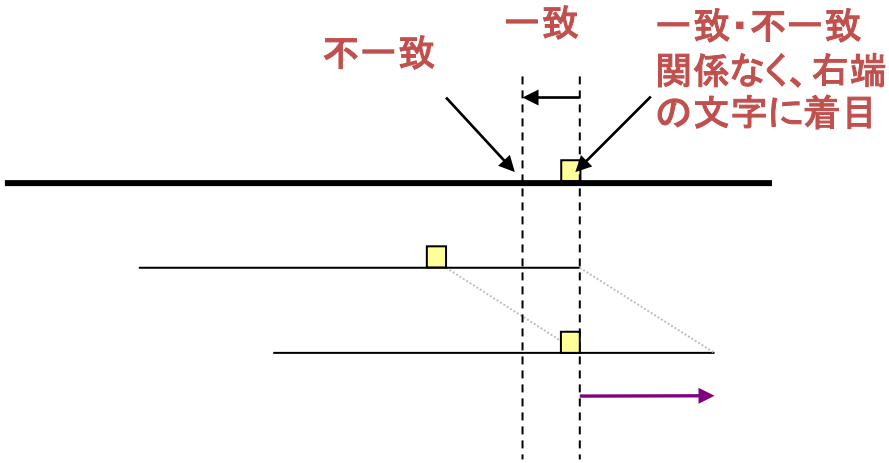
- 平均的にはなんと$O(n/\min(m, |\Sigma|))$
  - テキスト、パタンがランダム文字列であることを仮定
  - 言い換えると、スキップ量の期待値が$O(\min(m, \text{alphabet size}))$
- 最悪だと$O(nm)$
  - パタンの中に繰り返しが多く、アルファベットサイズが小さい場合にはあまり性能がよくない
  - ただし、最初の１出現だけを出力する場合であれば線形時間
    - 重なりのない出現をグリーディーに求める、などでも線形時間
  - $O(n)$になるように改良したアルゴリズムもいくつか存在
    - Turbo-BM (Crochemore et al. '92), Galil (1979), Smyth (2000), Apostolico-Giancarlo (1986), etc.
    - 繰り返し対策。繰り返しのないパタンであれば、逆に遅くなることも多い。

Horspool

- 不一致ルール（のバリエーション）のみを用いたアルゴリズム
  - 別に不一致したアルファベット（だけ）に着目する理由はない
  - 右端に着目した方が、無駄が少ない　（この技はBMで使ってもよい）
  - 平均計算量はBMと同じ



Boyer-Mooreの不一致ルール

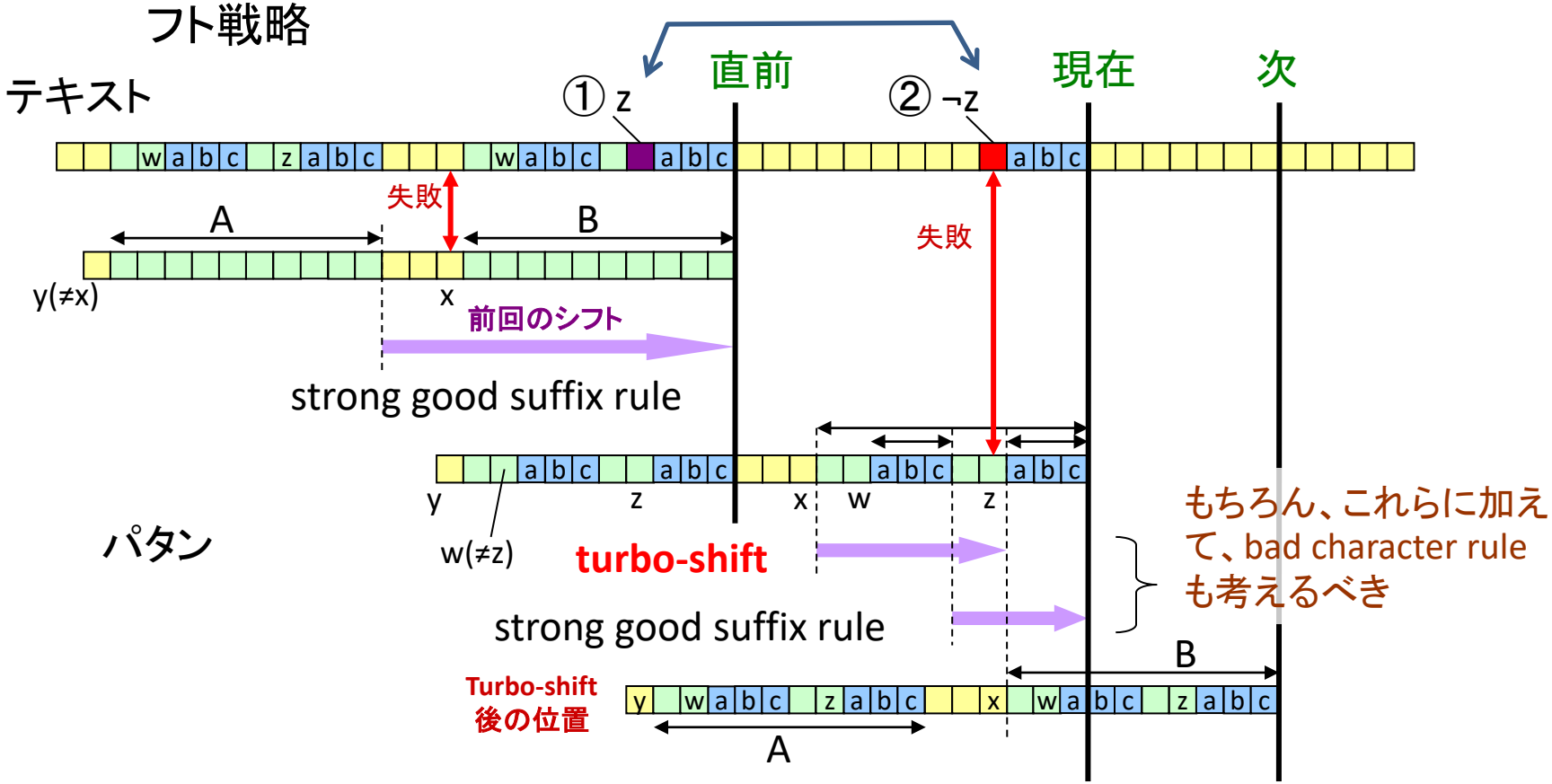


Horspoolの不一致ルール

Boyer-Mooreの最悪線形時間化： Turbo-BM アルゴリズム

Turbo-shift

- 直前に`strong' good suffix ruleを適用していた場合に、その時の一致長よりも今回の同ルールにおける一致長+シフト長が小さい時に使えるシフト戦略



Aの領域とBの領域が同じ文字列である一方で、① ■ と② ■ は異なる文字であるため、Bの領域を② ■ と重ねてもうまくいかない！

まとめ

- 文字列探索(照合)アルゴリズムのいろいろ
  - 前から
    - KMP, AC
  - 後ろから
    - BM, Horspool, Turbo-BM
- 次回
  - 力任せ系の高度化
    - Brute-force, Rabin-Karp, Shift-Or
  - Inexact matching algorithms
    - FFTを使った高速化
    - Alignment