修士論文

断片化BWTとその高速な圧縮・Maximal Exact Match検索への応用

平成30年2月1日提出

指導教員　　田浦 健次朗 教授

東京大学大学院 情報理工学系研究科
電子情報学専攻

48-166407 伊藤　大

概 要

我々は断片化 BWT (FBWT) という新たなデータ構造を作成し, 応用先を研究した. その結果二つの応用先を発見した. 一つは圧縮である. 広く使われている圧縮形式の Bzip2 を拡張し新たに Bzip2+を提案し, 圧縮率は Bzip2 より 1.4 倍最大悪くなるが, Bzip2 と比べて 1.2 倍から 1.9 倍圧縮の高速化を実現した. しかし, ほかの圧縮形式と比較した場合, 文字列が一様に分布しているものについて圧縮率と圧縮時間のトレードオフの中で優位性があるといえる結果になった.

もう一つは, ゲノム解析で重要な Maximal Exact Match (MEM) の検索である. 近年の技術発展により, ゲノムを安価にデータ化することが可能になりつつある. この状況で農業や医学に対しゲノムレベルでの解析, 品種改良や創薬, 効果的な薬の処方が可能になってきている. このゲノム解析において重要な役割を果たす課題として複数のゲノムを比較するというものがある. 比較にはシーケンスされデータになったゲノムのデータを比較し, 得点を出していく. この得点を出すアルゴリズムを直接用いるにはゲノムは大きすぎるため, 初めに文字列マッチングを行い, ある程度得点を出す範囲を絞り, 計算を行うことで現実的な時間で計算を行っている. ゲノムに対する文字列マッチングのアルゴリズムとデータ構造は古くから研究されているが, 本研究ではその文字列マッチングのうち MEM の検索に着目した. ゲノムを扱うアプリケーションで有名な BWA というものにも近年 MEM をベースとしたアプローチがとられるようになっており, また MEM 検索の高速化のみをテーマにした研究もなされている. 我々はこの MEM 検索において, FBWTMEM というデータ構造とアルゴリズムを作成した. FBWTMEM は名前のとおり FBWT を用いたデータ構造である. その結果は, 検索索引の時間を含めない場合既存手法よりも手軽に高速で省メモリなアルゴリズムの開発に成功した. MEM 検索においては最速のアルゴリズムのパラメータをチューニングしたものよりも半分程度のメモリ使用量で同等または 1.8 倍程度高速に動作した. また同程度のメモリ使用量では, 2 倍以上の高速化を達成することができた. ただし, 索引の構築を含めた場合一つの既存手法に劣ってしまう. しかし, 一つのゲノムに対して複数のゲノム, 2 から 5 つ程度の比較対象ゲノムが存在する場合我々の手法の方が有利となる結果が得られた.

目 次

図 目 次

表 目 次

第1章　序章

1.1　背景

本研究では Fragmented BWT[12] という新たなデータ構造を作成し, 応用先として圧縮とゲノム解析で重要な Maximal Exact Match の検索の二つにいて有用であることを発見した.

はじめに, 圧縮についてであるが, 圧縮技術は昔から研究が行われていて, 日常広く使われているものである. その圧縮の中で代表的なものに, 辞書式の zip や gzip[9], BWT[5] を用いた Bzip2[5] という形式がある. 本研究では Bzip2 に着目した. Bzip2 は Linux のソースや Wikipeida のダンプファイルの配布形式として使われていて, 一般的な圧縮形式と考えられる. 今回はこの Bzip2 の高速化をデータ構造の変更により成し遂げることを目指す.

Bzip2 では, テキストを数百 kB ごとに区切って, それぞれの塊に対して, Burrows-Wheeler transform (BWT) という可逆変換を施した後に, 圧縮を行う. この BWT による変換後の文字列は, 同じ文字が並びやすいという特徴を持つため, ランレングス圧縮に向いている. 近年, 我々の先行研究により, BWT を拡張した Fragmented BWT (FBWT)[12] という変換が提案された. FBWT はパラメータ k 個の同じ大きさの断片から構成されるデータ構造である. この先行研究では, BWT よりも FBWT の方が state of the art のアルゴリズムに対して 2 倍程度高速に構築できるということが示された. FBWT は, BWT と同様に同じ文字が連続しやすい特徴は継承されている. このため, FBWT を利用することで, Bzip2 より高速で同等程度の圧縮率を期待できる. 本研究では, Bzip2 に FBWT を移植し, FBWT 特有のパラメータを変更し, Wikipedia のダンプファイル, Linux のソースコード, ランダム, ヒトゲノムを対象に性能を計測, Bzip2 と比較した,

次にゲノム解析で重要な Maximal Exact Match の検索であるが, 近年ゲノム解析は急速に発達している. ヒトゲノム計画が完了した 2003 年当初では一人のヒトのゲノムをデータに起こす費用が 100 億円程度であった. それから現在までに技術が発展し, Next Generation Sequencing (NGS) という技術が今高速にまた安価にゲノムをデータにすることができている. ヒトゲノム一人分に対する費用は 10 万円台になり研究で手頃に, また一般にも手の届く価格になっている. NGS により手軽にゲノムデータを獲得することができ始めたことにより, 農業では遺伝子組み換え作物に対する解析により, 効率よく油を作りエネルギー問題を解決しようとするもの [19] や, 環境に存在するバクテリアの解析 [24] にも用いられている. また医学では, ヒトゲノムのコードの機能の解析 [20] や, ゲノム配列の差異による病気の研究 [22, 30] や, 効率的な創薬や処方の研究 [8], 異種間のゲノムを比較することによる先祖の系統進化を推定すること [7] ができている. 身近な事例としては, 2014 年には 9 万人以上のコード領域をシーケンスするという大規模な調査が行われた [6]. この結果, 健康な人でも 20 くらいの遺伝子は不活化していることが発見され, また, PCSK9 という遺伝子が不活性な人はコレステロール値が低いという発見もなされた. この PCSK9 を阻害する薬を作った結果, 60%もの人がコレステロール値を抑えることができたという [11]. ごく最近では CRISPR の技術によりゲノムエディティングもでき始めて [26] おり, ますますゲノムのコードの

機能を解析することの重要性が増している．このように広範囲の分野で今後ゲノムのコード解析はますます盛んに行われていくと期待ができる．

このゲノム解析に欠かせない技術として文字列比較・検索技術があり, その高速化は重要なテーマである．比較には編集距離やそれに類するスコアリングするアルゴリズム [27] が用いられるが, ゲノムは数 Mbps から数 Gbps のオーダーの大きさを持っており, スコアリングするアルゴリズム単体では実行時間が膨大になってしまう．そこではじめにある程度共通する位置を割り出し, その後その位置周辺に対してスコアリングを行う．これにより高速に比較が可能となる．

位置を検索するには, 一つのゲノムに対して索引を作りそれを用いて位置を検索する方法が一般的である．検索の索引は昔から研究されている分野である．一般的な文字に対しては suffix tree[15] を始め, suffix array[28], FM-index[23], Compressed suffix array[25] 等メモリを小さくかつ高速に動作させるアルゴリズムやデータ構造が開発されて来た．ゲノムに特化したアルゴリズムは suffix tree をベースとした MUMmer[16], sparse suffix array を用いた sparseMEM[13], compressed suffix array を用いた backwardMEM[21], enhanced sparse suffix array を用いた essaMEM[29], SSILCP を用いた slaMEM[10], 索引作成時間を短縮することで索引から検索までの全行程を含めた場合高速でハッシュをベースとした E-MEM[14] 等がこれまでに開発されて来た．

一方検索索引では我々は最近の研究で新たな索引 Fragmented Burrows-Wheeler transform (FBWT)[12] を提案した．このデータ構造は FM-index で用いられる Burrows-Wheeler transform (BWT)[17] というデータ構造を拡張したもので, FM-index をこれを用いてアルゴリズム的に拡張できる．従来の FM-index は BWT を用いて検索索引でよく用いられる suffix array は補助し, 高速な検索を可能にしたデータ構造である．一方, FBWT は sparse suffix array を補助するデータ構造と言える．suffix array はそのままでは大きくなってしまうためいくつか残してメモリを削減する．この一部を削除したものを sparse suffix array と呼ぶ．sparse suffix array と BWT を用いた検索も可能であるが, sparse suffix array から suffix array を復元する時間がかかってしまう．一方, FBWT を用いたものでは簡素な実装が可能であり, 検索結果が多くなる場合 BWT のものよりも有利となることがある．

1.2　本研究の目的と貢献

本研究の圧縮については, Bzip2 と圧縮率が同等で高速な圧縮を目指した．圧縮時間に対しては 1.2 倍から 1.9 倍程度高速になったが, 圧縮率は最大 1.36 倍程度低下, 解凍時間も最大 1.56 倍低下する結果となった．

一方 Maximal Exact Match については FBWT を用いた新たな索引を用い, 検索時間をさらに短縮することを目的としている．この検索については検索時間について高速な essaMEM をベースとし, さらなる高速化, 省メモリ化を実現した．具体的な結果の例としては, ヒトゲノムとマウスゲノム間の比較において essaMEM のパラメータを変化させた時の最速値と我々のデフォルトのパラメータのものを比較した時半分程度のメモリ消費で 1.8 倍, さらに同程度のメモリ消費で 7.4 倍の高速化が実現できた．ヒトゲノムとチンパンジーゲノム間では, essaMEM 最速値の比較で 1.4 倍, 同程度のメモリ消費で 5.15 倍高速に動作することが確認できた．検索索引作成から検索全てを含んだ実行時間で高速な E-MEM とは, シングルスレッドでは同程度の実行時間であった．しかし, スレッドが多くなるにつれ E-MEM の方が勝るようになった．しかし, 検索時間では我々

の手法が 5 倍程度高速なため多くのゲノムを一つのゲノムと比較したい場合我々の手法の方が有効であると言える.

1.3　本稿の構成

第 I 部では圧縮の高速化について記す. 2 章では Bzip2 にまつわる予備知識と我々の提案手法にかかわるアルゴリズムを記す. 3 章では我々の圧縮手法のアルゴリズムを示す. 4 章では各種圧縮形式や Bzip2 との比較を示し, 5 章で圧縮についてのまとめを行う.

次に第 II 部では MEM 検索について述べる. 6 章では本研究の問題の位置づけとなる文字列比較アルゴリズムの概要と我々が高速化を目指す問題の定義づけと全般的な計算方法を示す. 7 章では本研究や既存研究で使用される文字列検索データ構造とアルゴリズムを紹介する. 8 章では既存研究, 主に state of the art のアルゴリズムを紹介紹介する. 9 章では既存研究のパラメータを変化させた時の問題点を明らかにし, 高速化の糸口を説明する. 10 章では 9 章を踏まえ我々の手法を提案し, 11 章で評価を行う. 最後 12 章で MEM 検索についてまとめる.

第Ⅰ部

圧縮の高速化

第2章　予備知識

この章では本研究にかかわる基本知識を記す．初めに，本論文で用いる記法を紹介した後，FBWT と BWT の定義と実装で用いられている suffix array と fragmented suffix array というデータ構造を示す．次に，BWT の定義と BWT の逆変換，BWT による出力文字列が圧縮に向いていることの直観的な説明を与える．最後に FBWT の定義を記す．FBWT の逆変換については我々の先行研究 [12] では，FBWT の逆変換は述べられていないので，3 節で記す．

2.1　記法

$T[i]$ は文字列または配列 T の i 番目の文字または要素である．$T[i,j]$ は文字列 T の i 番目から始まり j 番目で終わる部分文字列である．i を省略した場合 ($T[,j]$) は j までの T の接頭文字列，j を省略した場合 ($T[i,]$) は i から始まる T の接尾文字列とする．また，ST とは，二つの文字列 S と T を接合した文字列であるとする．

2.2　suffix array と fragmented suffix array

対象とする文字列を T し，T の長さを n としたとき，suffix array とは，0 から $n-1$ までの整数を並べ替えた要素数 n の配列 (SA とする) で，0 以外のすべての $i\,(0 < i < n)$ について，$T[SA[i-1],] <_{lex} T[SA[i],]$ を満たす．ここで，$<_{lex}$ は，辞書順を示す．図 2.1 に $T =$ “mississippi\$” の suffix array を示す．\$は文字列中で一番小さい文字である．後述の BWT のため，各行に循環シフト文字列 $T[i,]T[,i-1]$ が入っている．\$が T の末尾にあるため，辞書順ソートに後半の $T[,i-1]$ は関係しない．これにより図の添字のソート後は suffix array となっている．

fragmented suffix array は，suffix array の要素のうち，k で割って l 余るものを集めたものである．これはつまり，T の k で割って l 余る位置から始まる接尾辞で先頭の位置をソートした配列である．この fragmented suffix array を $FSA_{l/k}$ と表記する．図 2.2 は $T =$ “mississippi\$\$\$\$”，$k = 5$ の時の FSA を示している．

2.3　BWT

対象とする文字列を T としたとき，この文字列の末尾に \$ を一つ付け加えた文字列を $T_\$$ とする．\$ は T の中のどの文字よりも小さい文字である．この $T_\$$ を用いて，Burrows-Wheeler transform (BWT) は以下によって定義される．

$$BWT[i] = \begin{cases} T_\$[SA[i]-1] & (SA[i] \neq 0) \\ T_\$[n-1] & (SA[i] = 0) \end{cases} \tag{2.1}$$

index	suffix
0	mississippi$
1	ississippi$m
2	ssissippi$mi
3	sissippi$mis
4	issippi$miss
5	ssippi$missi
6	sippi$missis
7	ippi$mississ
8	ppi$mississi
9	pi$mississip
10	i$mississipp
11	$mississippi

index	suffix array	suffix	BWT
0	11	$mississippi	i
1	10	i$mississipp	p
2	7	ippi$mississ	s
3	4	issippi$miss	s
4	1	ississippi$m	m
5	0	mississippi$	$
6	9	pi$mississip	p
7	8	ppi$mississi	i
8	6	sippi$missis	s
9	3	sissippi$mis	s
10	5	ssippi$missi	i
11	2	ssissippi$mi	i

図 2.1: mississippi$に対する suffix array と Burrows Wheeler transform.

FSA0/3	suffix	FBWT0/3
0	mississippi$	$
9	pi$mississip	p
6	sippi$missis	s
3	sissippi$mis	s

FSA1/3	suffix	FBWT1/3
10	i$mississipp	p
7	ippi$mississ	s
4	issippi$miss	s
1	ississippi$m	m

FSA2/3	suffix	FBWT2/3
11	$mississippi	i
8	ppi$mississi	i
5	ssippi$missi	i
2	ssissippi$mi	i

図 2.2: mississippi$$$$ に対する FSAs と FBWTs.

この定義からすぐに以下のことがいえる．文字列 $T_\$$ のすべての循環文字列を辞書順ソートして，それぞれの末尾の文字を集めた文字列を出力とするものである．ここで，i 番目から始まる循環文字列とは，$T_\$[i,]T_\$[,i-1]$ のことである．これを可視化したのが図 2.1 である．BWT はソートされた循環文字列を並べた行列の一番右にある列の文字列である．この事実は以降で述べる BWT の出力文字列が圧縮しやすいという性質を直観的に理解するうえで重要である．

2.4　inverse BWT

BWT には，LF mapping と呼ばれる性質がある．図 2.3 は LF mapping による逆変換を表している．赤い矢印が LF mappng を表す．LF mapping は逆変換では，前もって保存してある start の位置から LF mapping を連続で使い，元文字列を後ろから一文字ずつ復元していく．LF mapping の数式は以下である

$$LF\,(i) = C\,(BWT[i]) + Occ\,(BWT[i], i) \tag{2.2}$$

index	suffix array	suffix	BWT
0	11	$mississippi	i
1	10	i$mississipp	p
2	7	ippi$mississ	s
3	4	issippi$miss	s
4	1	ississippi$m	m
5	0	mississippi$	$
6	9	pi$mississip	p
7	8	ppi$mississi	i
8	6	sippi$missis	s
9	3	sissippi$mis	s
10	5	ssippi$missi	i
11	2	ssissippi$mi	i

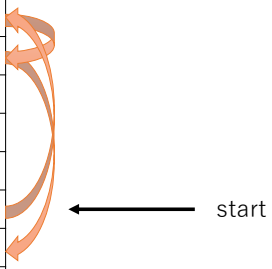


図 2.3: mississippi$に対する BWT の LF mapping

$C(c)$ は c 未満の BWT に出現する文字数である．$Occ\,(c, i)$ は，BWT の i 番目まで c の出現数である．元の文字列を復元できるのは，$LF\,(i)$ 関数に，以下の関係が成り立っているからである．

$$SA[LF\,(i)] = SA[i] - 1 \tag{2.3}$$

suffix array の要素はそれぞれ，循環シフト文字列の先頭に対応する，元の文字列の位置を示しているとみることができるので，$LF\,(i)$ 関数を連続で用いると，元の文字列を後ろから復元できることがわかる．

2.5　BWT の圧縮のメリット

BWT 後の文字列は同じ文字が連続して並びやすいという性質がある．これは，ほとんどのテキストにある文字の後に現れる文字に偏りがあるからである．例えば，英語なら，this，that，the，there などのように頻繁に th が出現する．入力とする文字列 T の循環シフト文字列 $T[i,]T[,i-1]$ のうち，h から始まるものは，末尾に t が出現しやすいということになる．BWT は全循環シフト文字列を辞書順ソートしたものなので，h から始まる循環シフト文字列のブロックが出来上がる．そのブロックの末尾の文字たちには t が高頻度で出現するため，BWT の h のブロックにあたるところでは t が並びやすくなるのである．Bzip2 ではこの性質を利用して，ランレングス圧縮やハフマン符号化を行って高い圧縮率を出している．

理論的な値は [18] を参照．

2.6 FBWT

対象とする文字列を T としたとき，この文字列に一つ以上の \$ を，文字列長が k の倍数になるまで付け加えた文字列を $T_{\$}$ とする．\$ は，T の中のどの文字よりも小さい文字である．n を $T_{\$}$ の長さとする．l 番目の FBWT$\left(FBWT_{l/k}\right)$ は以下によって定義される．

$$FBWT_{l/k}[i] = \begin{cases} T_{\$}[FSA_{l/k}[i] - 1] \\ \qquad (FSA_{l/k}[i] \neq 0) \\ T_{\$}[n-1] \\ \qquad (FSA_{l/k}[i] = 0) \end{cases} \tag{2.4}$$

FBWT はすべての $l\,(0 \leq l < k)$ によってできた断片の集合のことをいう．図 2.2 に “mississippi\$” の FBWT を示す．断片の数は 5 つとした．

第3章　提案手法

この章では FBWT の逆変換と，FBWT の性質，実装について述べる．

3.1　inverse FBWT

この節では FBWT の逆変換を述べる．FBWT も BWT と同じように，LF mapping と同様の性質を持つ．図 3.1 の赤い矢印がその性質を表す．ここでも元の文字列を後ろから復元するようにたどることができる．FBWT での LF mapping は数式であらわすと以下のようになる．

$$\begin{aligned} LF_k\left(l,i\right) =& C_l\left(FBWT_{l/k}[i]\right) \\ &+ Occ_l\left(FBWT_{l/k}[i],i\right) \end{aligned} \tag{3.1}$$

$C_l(c)$ は c 未満の $FBWT_{l/k}$ に出現する文字数である．$Occ_l\left(c,i\right)$ は，$BWT_{l/k}$ の i 番目まで c の出現数である．この $LF_k\left(l,i\right)$ 関数を用いると，以下の関係が成り立つ．

$$FSA_{l-1/k}[LF_k\left(l,i\right)] = FSA_{l/k}[i] - 1 \tag{3.2}$$

ここで，$FSA_{l-1/k}$ は，$l=0$ の時は，$FSA_{k-1/k}$ を意味するとする．これにより，元の文字列を復元することができる．

3.2　FBWT の性質

2.5 節で述べたように BWT は同じ文字列が連続しやすい．この理由がある文字の次の文字には偏りがあるという理由だった．今回の FBWT についても，同様のことが期待できる．それは，FBWT はパラメータ k ごとに文字列から文字をとっているだけであり，その隣の文字との出現頻度には偏りがあると期待できるからである．ただし，同じサイズの BWT と一つの断片が同程度に同じ文字が並びやすくなるためには，BWT よりも広範に文字の似た偏りが起こっていなければいけないことになることは留意しておかなければならないことである．

3.3　実装

今回は Bzip2 と比べるため，Bzip2 のソースコードを変更することにした．Bzip2 のソースは `http://www.bzip.org/downloads.html` より 1.0.6 のバージョンを扱った．

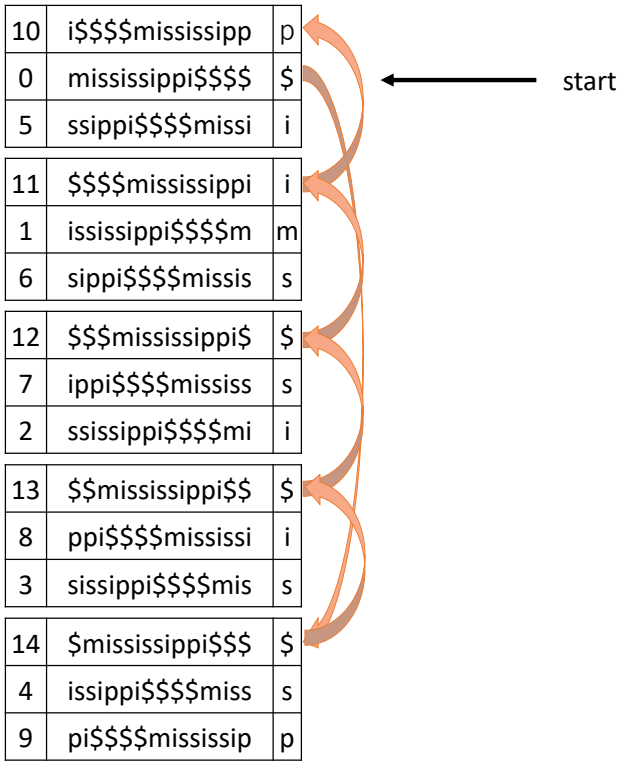


図 3.1: mississippi$$$$に対する FBWT の LF mapping

Bzip2 は圧縮時には，対象のテキストを指定された 100kB から 900kB の塊に分けてそれぞれ BWT にし，BWT を MTF(Move-To-Front) 法ハフマン符号化により圧縮を行っている．今回は BWT の変換を FBWT の変換に置き換えることで行った．Bzip2 の BWT の中心は quick sort であるが，一部 1byte の文字に最適化された部分があった．FBWT を構築するときは 1byte よりも大きい文字を扱わなければならないため，この最適化は無効化して行った．

解凍時は，BWT の逆変換を行うところを中心に変更した．

第4章　評価

この章では 3.3 節で示した実装と，元の Bzip2 の比較結果を示す．はじめに，一つの断片の大きさは 100kB，断片の数を 2 から 8 まで変化させた．それに対して，Bzip2 が BWT をするサイズは 100kB と 800kB のものを計測した．データは，ソースコード 4.1 により生成されたランダム文字列 100MB，[4] の Wikipedia のダンプファイルのうち，先頭 100MB，[2] からヒトゲノムのうち，先頭 100MB，[3] から，Linux kernel 4.9 の先頭から 100MB，[1] から Apache の 2017 年 6 月 30 日のログファイルを先頭から 100MB 使用した．また，それぞれのファイルについて，行をランダムに並び変えたものも私用している．並び変えたものを使う意図は，3.2 節での留意点を念頭に入れ，出現する文字列がある程度分散するようにし，議論をするためである．さらに，他の圧縮形式との比較も行った．実験を行った環境は Intel Xeon CPU E5-2660 で，OS は Ubuntu 14.04.5 ，カーネルは Linux 3.13.0 である．

List 4.1: ランダムなテキストの生成

```
1 % \begin{lstlisting}[caption=generate random text,label=generateRandomTxtsh,language=bash]
2  cat /dev/urandom | tr -dc 'a-zA-Z0-9' | fold -w $((2**8)) | head -n $((100*2**12))
```

各グラフの見方を述べる．初めにの 6 つの図，図 4.1a，図 4.1b，図 4.1c，図 4.1d，図 4.1e，図 4.1f の共通の見方として，Bzip2 はオリジナルの Bzip2 1.0.6，そのあとに続く 100kB と 800kB は，それぞれ Bzip2 が一度に BWT にするブロックのサイズである．Bzip2+は今回我々が Bzip2 を改変し，FBWT を組み込んだものである．Bzip2+に続く二つの数字は，一つ目が FBWT の大きさで，今回は一つの断片あたり 100kB である．二つ目の数字は断片数である．横軸は使用したデータセットである．図 4.1a，図 4.1b，図 4.1e，図 4.1f の縦軸は実行時間，図 4.1c，図 4.1d の縦軸はファイルサイズを示している．図 4.2a，図 4.2b，図 4.2c，図 4.2d はそれぞれ図 4.1a，図 4.1b，図 4.1e，図 4.1f のうちの主なタスクのそれぞれの実行時間である．図 4.2a，図 4.2b の other には，ファイル入出力も含まれている．図 4.2c，図 4.2d の decompress には，ファイル読み込みと解凍，output には，inverse BWT または，inverse FBWT の実行とファイル書き込みも含まれている．図 4.2e は圧縮ファイルサイズと解凍時間の関係を表している．横軸は圧縮ファイルサイズ，縦軸は解凍時間である．使用したデータは Wikipedia のダンプファイルである．このファイルのうち，先頭から 1MB，25MB，50MB，75MB，100MB それぞれとったものを扱った．それぞれのファイルに対して，Bzip2 のブロックサイズを 100kB で圧縮したものと，Bzip2+でブロックサイズを 100kB，断片数を 8 にして圧縮したものである．どちらも同じブロックサイズにしたのは，キャッシュの影響を排除するためである．

初めに，一つの断片の大きさを 100kB に固定した時のことを述べる．図 4.1a，図 4.1b は各データセットに対する圧縮の実行時間である．図 4.1b のものはすべて行を並び変えたものである．断片の大きさが 100kB のものは，Bzip2+ 200kB 2 から Bzip2+ 800kB 8 のデータである．これからわかるように，断片の数を増やすと実行時間が短縮されていくことがわかる．ヒトゲノムの断

片数 2 の時に実行時間が遅くなるのは，おそらく，BWT を行うときの 1byte 文字固有の最適化を無効にしているからであると考えられる．元の Bzip2 の一ブロック 100kB のものに比べると，ランダムでは最大 1.27 倍，Wikipedea では 1.53 倍，ヒトゲノムでは 1.43 倍，Linux のソースでは 1.58 倍，Apache のログファイルでは 1.91 倍高速に動作した．ランダムに行を並び変えたものでもほぼ同等の高速化を行うことができた．次に，図 4.1c，図 4.1d は圧縮したファイルの大きさである．図 4.1d のものはすべて行を並び変えたものである．ランダムとヒトゲノム，行を並び変えたものでは，それぞれのメソッド間での変化はあまり見られなかった．一方，行を並び変えていない Wikipedia や Linux のソースでは，断片の個数が多くなるにつれ，ファイルサイズが大きくなっていく傾向がある．Wikipedia では，最大 1.17 倍，Linux では最大 1.36 倍大きくなった．最後に，図 4.1e，図 4.1f は解凍実行時間である．これも，断片の個数が多くなるにつれ，遅くなる傾向がみられた．ランダムは最大 1.21 倍，ゲノムは 1.28 倍，Wikipeida は 1.37 倍，Linux のソースは 1.56 倍遅くなっている．

圧縮が高速になった要因については，図 4.2a，図 4.2b からわかるように，FBWT の構築が BWT のそれよりもはるかに高速であるからである．ランダムでは Bzip2 の 100kB のものに比べて，最大 5.84 倍，Wikipedia は 4.79 倍，ヒトゲノムは，2.92 倍，Linux のソースは，4.12 倍，BWT の構築より FBWT の構築が高速となっている．

解凍が遅くなっている原因について考える．図 4.2e からわかるように，Bzip2 と Bzip2+の解凍では圧縮ファイルサイズに対する解凍時間の比はあまり変わらない．具体的には Bzip2+の傾きは Bzip2 のものより 1.08 倍小さい．このことから，圧縮が遅い要因は，Bzip2+にするための実装の変更ではなく，圧縮率に原因があると考えられる．

圧縮率について，ランダムとゲノム，行を並び変えたものでは，断片の数を変化させても圧縮率には影響しなかった．一方，行を並び変えていない Linux のソース，Wikipedia のダンプファイルでは，圧縮率が著しく低下した．これについて，empirical entropy という，文字列の偏りを考慮したエントロピーを用いて考察する．まず，empirical entropy とは，以下のように定義される．

$$H_k(T) = \frac{1}{n} \sum_{w \in \Sigma^k} n_w \left[\sum_{i=1}^{h} \frac{n_{w\alpha_i}}{n_w} \log \left(\frac{n_{w\alpha_i}}{n_w} \right) \right] \tag{4.1}$$

k は，パラメータである．n はテキスト中文字数，Σ^k はテキスト中にある長さ k の文字列の集合，n_w は文字列 w の出現回数，$n_{w\alpha_i}$ はテキスト中で文字列 w の前にある文字 α_i の数である．このエントロピーによって，テキスト中でのある文字の前にある文字の出現頻度の偏りをとらえることができる．今回は，$k = 1$ の結果を表 4.1，表 4.2 に示す．データは，それぞれ断片が 100kB になるように，先頭から (断片数)x100kB とってきた．今回はそれぞれの断片についてエントロピーを知りたいので，断片の数おきの文字出現頻度を計算した．そのため，エントロピーも断片の数だけでるが，今回はその平均値を表には出している．一番左の数字が断片数である．

ここからわかることとして，ランダムなものとゲノム，行を並び変えたものに比べて行を並び変えていない Wikipedia や Linux のソースは断片数を大きくするとエントロピーがより大きくなっている．この傾向があるために，今回圧縮率が悪くなったのではないかと思われる．この傾向の直観的な理解として考えられるのは，例えば，Linux のソースは，小さいファイルを結合したものであるため，変数名等が局在していることがあげられる．

最後に，ほかの圧縮形式との比較を行う．図 4.3a，図 4.3b，図 4.3c，図 4.3d，図 4.3e，図 4.3f，図 4.4a，図 4.4b，図 4.4c，図 4.4d は各データについて圧縮時間を y 軸に圧縮率を x 軸にしたグ

表 4.1: Empirical entropy と断片数の関係。データは並び替えられていない。

	Random	Wikipedia	Genome	Linux	Apache
1	0.004049	0.002546	0.001158	0.002355	0.001823
8	0.004049	0.002632	0.001284	0.002408	0.001853
16	0.004049	0.002645	0.00129	0.002433	0.001862

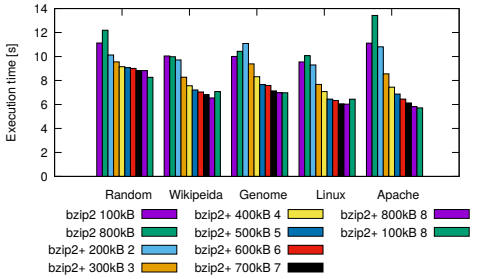
表 4.2: Empirical entropy と断片数の関係。データは並び替えられている。

	Random	Wikipedia	Genome	Linux	Apache
1	0.004049	0.002673	0.001407	0.002763	0.0019
8	0.004049	0.002676	0.001409	0.002791	0.0019
16	0.004049	0.002683	0.00141	0.002797	0.001906

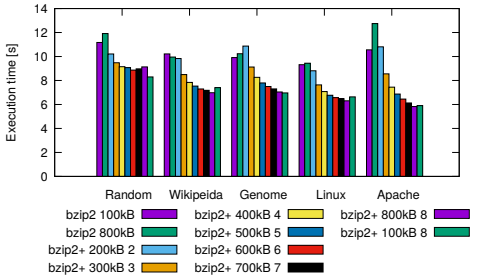
ラフである．比較した圧縮形式は，Bzip2 バージョン 1.0.6，gzip バージョン 1.6，xz バージョン 5.1.0alpha である．Bzip2 については一度に BWT するサイズを 100kB から 800kB に変化させた．xz はオプションに 1 から 9 をそれぞれ与えたものを表している．また，Bzip2+については，FBWT 一つの断片が 100kB になるようにし，FBWT 全体を 100kB から 800kB にしたもの (グラフの凡例は bzip2+ same fragment size) と，FBWT 全体を 800kB で断片数を 2 から 8 まで変化させたもの (グラフの凡例は bzip2+ same FBWT size) を示した．行を並び替えていないものについては gzip にほとんど負けている．一方，行を並び替えている方ではランダムのもの以外では圧縮率と圧縮時間のトレードオフの中で選択肢の一つとしてあり得る．つまり，広域で文字の分布が偏っていない文字列については，我々の手法はほかの手法に対して少し劣る圧縮率であるが，高速な圧縮時間を期待できるものであると考えられる．

解凍時間の比較を行う．図 4.5a，図 4.5b，図 4.5c，図 4.5d，図 4.5e，図 4.5f，図 4.6a，図 4.6b，図 4.6c，図 4.6d は各データについて解凍時間を y 軸に圧縮率を x 軸にしたグラフである．xz と gzip が圧縮率と解凍時間についてトレードオフがあり，Bzip2，Bzip2+についてはランダムのもの以外上回られていることがわかる．

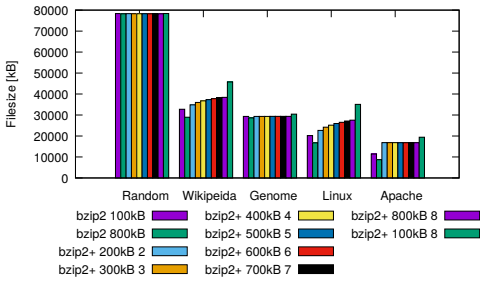
以上から，Bzip2+はあまり解凍されず，また文字列の分布が一様なものについて有用性があると考えられる．



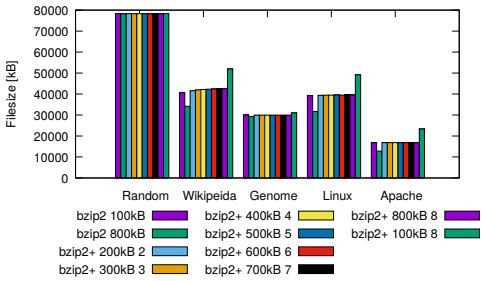
(a) 100MB のデータそれぞれに対する圧縮時間の比較



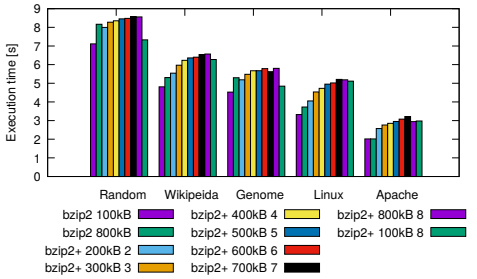
(b) 並び替えられた 100MB のデータそれぞれに対する圧縮時間の比較



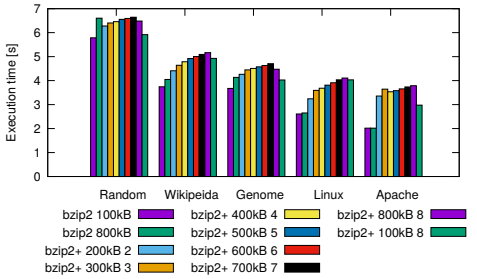
(c) 100MB のデータそれぞれに対する圧縮ファイルサイズの比較



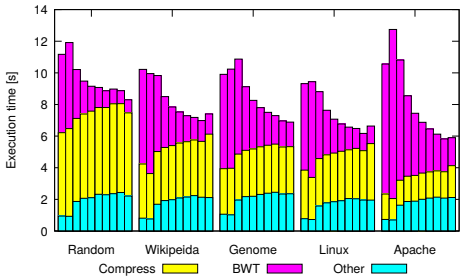
(d) 並び替えられた 100MB のデータそれぞれに対する圧縮ファイルサイズの比較



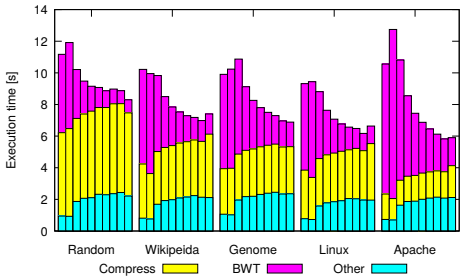
(e) 100MB のデータそれぞれに対する解凍時間の比較



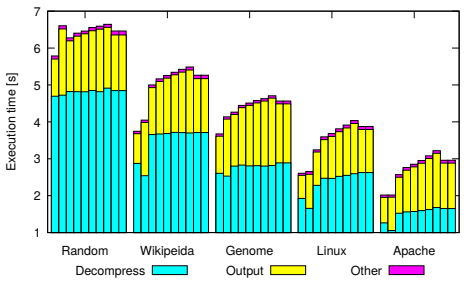
(f) 並び替えられた 100MB のデータそれぞれに対する解凍時間の比較



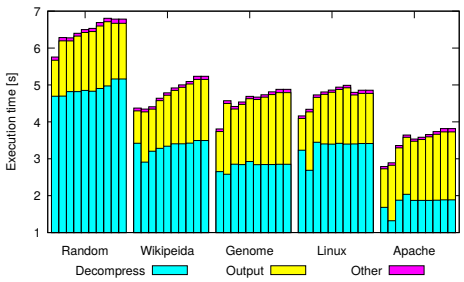
(a) 100MB のデータそれぞれに対する圧縮時間の内訳



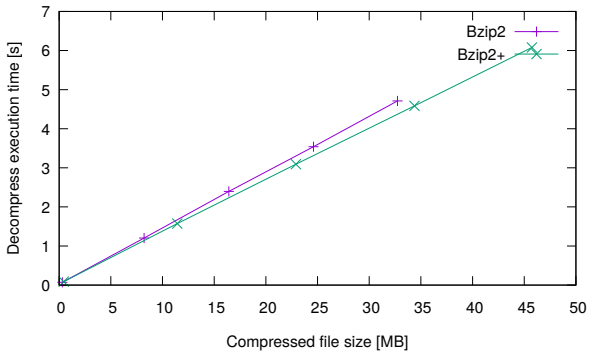
(b) 並び替えられた 100MB のデータそれぞれに対する圧縮時間の内訳



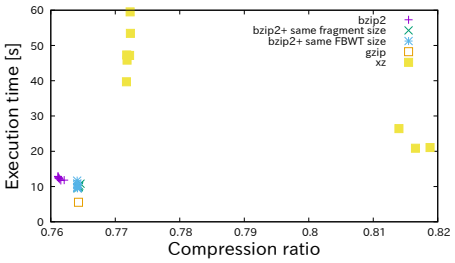
(c) 100MB のデータそれぞれに対する解凍時間の内訳



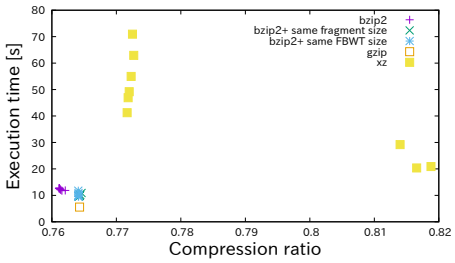
(d) 並び替えられた 100MB のデータそれぞれに対する解凍時間の内訳



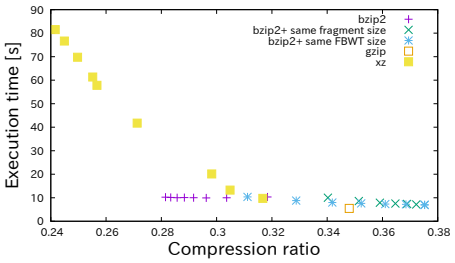
(e) Wikipedia の圧縮ファイルサイズと解凍時間の関係



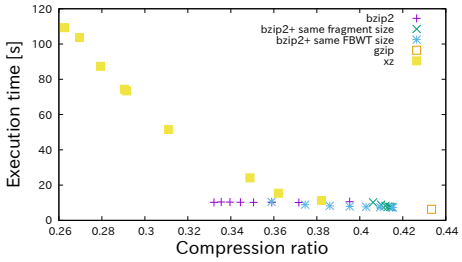
(a) ランダムテキストの各種圧縮形式の圧縮率と圧縮時間の比較



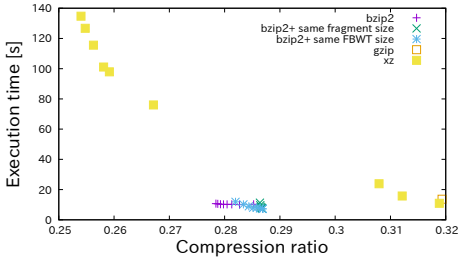
(b) 並び替えられたランダムテキストの各種圧縮形式の圧縮率と圧縮時間の比較



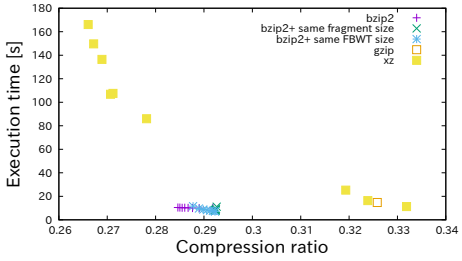
(c) Wikipedia のダンプファイルの各種圧縮形式の圧縮率と圧縮時間の比較



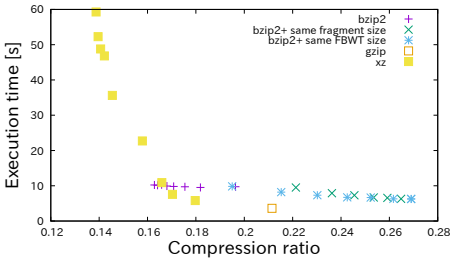
(d) 並び替えられた Wikipedia のダンプファイルの各種圧縮形式の圧縮率と圧縮時間の比較



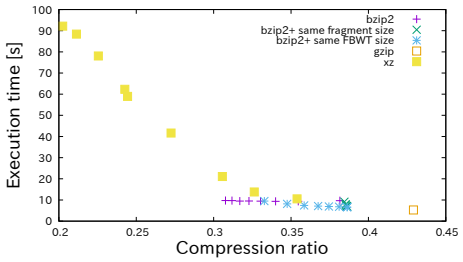
(e) ゲノムの各種圧縮形式の圧縮率と圧縮時間の比較



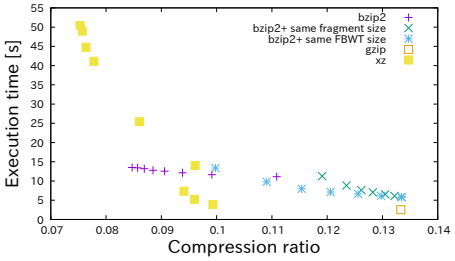
(f) 並び替えられたゲノムの各種圧縮形式の圧縮率と圧縮時間の比較



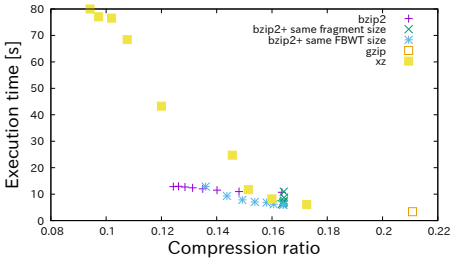
(a) Linux のソースファイルの各種圧縮形式の圧縮率と圧縮時間の比較



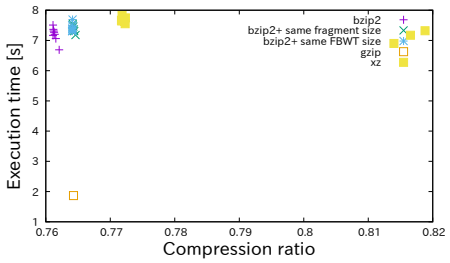
(b) 並び替えられた Linux のソースファイルの各種圧縮形式の圧縮率と圧縮時間の比較



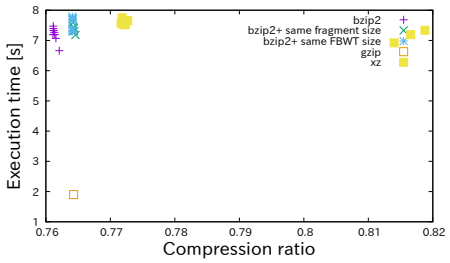
(c) Apache のサーバーログの各種圧縮形式の圧縮率と圧縮時間の比較



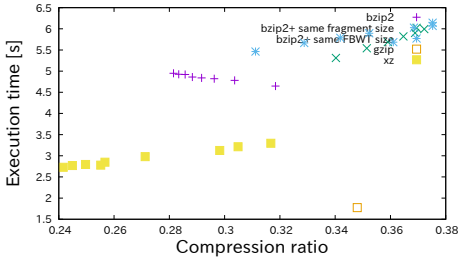
(d) 並び替えられた Apache のサーバーログの各種圧縮形式の圧縮率と圧縮時間の比較



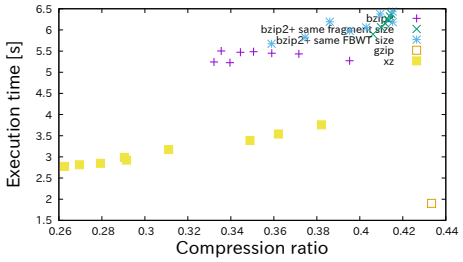
(a) ランダムテキストの各種圧縮形式の圧縮率と解凍時間の比較



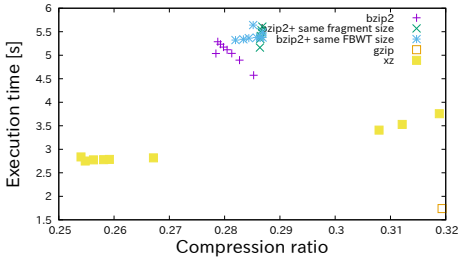
(b) 並び替えられたランダムテキストの各種圧縮形式の圧縮率と解凍時間の比較



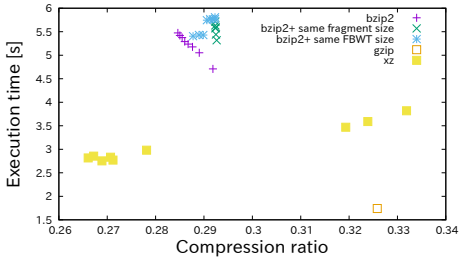
(c) Wikipedia のダンプファイルの各種圧縮形式の圧縮率と解凍時間の比較



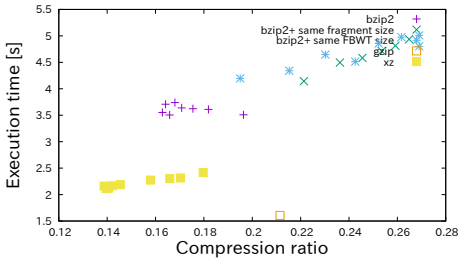
(d) 並び替えられた Wikipedia のダンプファイルの各種圧縮形式の圧縮率と解凍時間の比較



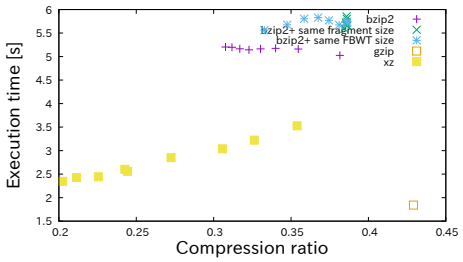
(e) ゲノムの各種圧縮形式の圧縮率と解凍時間の比較



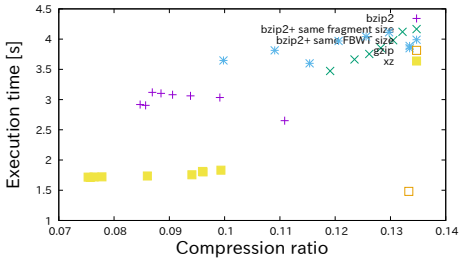
(f) 並び替えられたゲノムの各種圧縮形式の圧縮率と解凍時間の比較



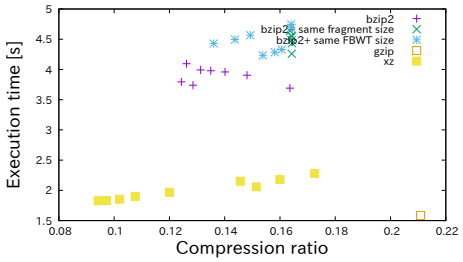
(a) Linux のソースファイルの各種圧縮形式の圧縮率と解凍時間の比較



(b) 並び替えられた Linux のソースファイルの各種圧縮形式の圧縮率と解凍時間の比較



(c) Apache のサーバーログの各種圧縮形式の圧縮率と解凍時間の比較



(d) 並び替えられた Apache のサーバーログの各種圧縮形式の圧縮率と解凍時間の比較

第5章　結論

文字列の出現に偏りの少ないテキストに用いると圧縮率の面から，我々の Bzip2+は Bzip2 と変わらない性能を出すことができる．圧縮時間面では，Bzip2 に比べて 1.2 倍から 1.9 倍高速にすることができる．また，xz や gzip に対しても優位性がみられる．しかし，解凍時間については劣ってしまう．一方，単語や文字列の局所性の高い Wikipedia や Linux のソースでは，高速に圧縮が可能であるが，圧縮率が低下してしまう．このことから，Bzip2+の用途として，文字列の出現に偏りの少ないものでかつ解凍があまり要求されないものについて有用であると考えられる．具体的な用途としてはサーバーのログが期待されるが，Apache のログではあまり優位性は確認されなかった．

第II部

Maximal Exact Match検索の高速化

第6章　文字列の比較アルゴリズム・ヒューリスティックスとMaximal Exact Match

この章では本研究の全体像として重要な, Smith-Waterman[27] のアルゴリズムという二つの文字列の比較のアルゴリズムを説明する. その後, Smith-Waterman のアルゴリズムに対するヒューリスティックで重要な *Maximal Exact Match* という文字列の部分一致とその計算手法を紹介する.

今回対象とする文字列の比較は, 二つの文字列の類似度を計算するものである. ゲノム比較では Smith-Waterman のアルゴリズムにより類似度が計算されることがおおいため, 初めにこのアルゴリズムを説明し, 次章以降のつなぎになるそのヒューリスティックスの方向性を述べる. 最後にそのヒューリスティックスに欠かせない文字列検索の問題, Maximal Exact Match を紹介する.

6.1　Smith-Watherman のアルゴリズムとヒューリスティックス

Smith-Waterman のアルゴリズムは二つの文字列の類似度をはかるものである. これは編集距離と似ているが, 数学でいう距離とは異なる. このアルゴリズムでは文字の欠損, 挿入, 変化に対してそれぞれ加点する点数をユーザーが決めることができる. これにより, ゲノムではあまり起こりえない変異や起こりやすい変異に対して区別して行うことができるためゲノム比較をする場合よく用いられる.

アルゴリズムは動的計画法で行われる. 以下の式を満たすようにテーブル H が作られ, その中から最大値を算出して類似度の高い部分を出す. S と T は比べる文字列としたとき, 計算量は $O(mn)$ となる. m は S の長さ, n は T の長さとする. $S[i], T[i]$ はそれぞれの文字列の i 番目の文字を示す.

$$H[i,j] = \max \begin{cases} H[i-1,j-1] + s(S[i], T[j]) \\ H[i-1,j] + W \\ H[i,j-1] + W \\ 0 \end{cases} \tag{6.1}$$

$H[i,j]$ は S の先頭から i 番目までと T の j 番目までの文字列を比較した時の取りうる最大の点数を示す. $s(a,b)$ は a と b が一致していた時と違っていた時のそれぞれの点数を返す. W は欠損または挿入に対する点である. 一番上の式 $H[i-1,j-1] + s(S[i], T[j])$ は $S[i], T[j]$ が一致または不一致していた場合, 二番目の式 $H[i-1,j] + W$ は T に欠損または S に挿入が存在していた時に対応する. 三番目の式は二番目の式と逆である. 図 6.1 に “thorough” と “though” という単語を比較した際のテーブルを示す. 文字が一致していた時の得点が 2, 不一致の時が -2, 欠損または挿入が -3 点としている. この例では, “thorough” と “though” それぞれ最後の文字までの文字列が

		t	h	o	r	o	u	g	h
	0	0	0	0	0	0	0	0	0
t	0	2	0	0	0	0	0	0	0
h	0	0	4	1	0	0	0	0	2
o	0	0	1	6	3	2	0	0	0
u	0	0	0	3	4	1	4	1	0
g	0	0	0	0	1	2	1	6	3
h	0	0	2	0	0	0	0	3	8

match	2
substi	-2
gap	-3

図 6.1: Smith-Waterman のアルゴリズムの例．“thorough” と “though” の比較である．substi はミスマッチを意味する．

もっとも類似度が高い結果になった．始まりの位置を計算することもできるが, 本論文ではかかわらないので説明を行わない．

Smith-Waterman のアルゴリズムは $O(mn)$ であるが, ゲノムは数 Mb から数 Gb 程度と非常に大きいため実行時間が非常に大きくなってしまう．これを抑えるためによく行われるヒューリスティックスがある．その方法の発想としては二つの文字列の間で長く一致した位置を起点としてその周辺に対して Smith-Waterman のアルゴリズムの計算をするというものである．この一致のさせ方において重要なのが本論文の対象である文字列検索技術が用いられる．次の 6.2 節では本論文で扱う一致の定義を行う．それを実現するデータ構造とアルゴリズムは 8 章や 10 章で記す．

6.2　Maximal Exact Match と計算アルゴリズム

二つの文字列間の一致において用いられる方向性として二つある．一つは固定長の文字列を片方からとりもう片方に対して検索して候補を出す．この方向性では候補が一部大量に出てきてしまった時に後の類似度計算が重くなってしまう．また固定長を長くしすぎると候補が少なくなりすぎてしまい精度が悪くなってしまう．そこで最短の一致の長さを決めておき, それぞれの一致の長さも後段に伝えるという方法が考えられる．これによりもし大量の候補が出てきてしまっても長いマッチから計算する等のヒューリスティックスを可能にする．近年のゲノムの一致問題では後者の方法がよく使われている．この一致のうちよく使われるものが Maximal Exact Match (MEM) である．MEM の定義は, 二つの文字列のそれぞれの部分文字列でそれ以上左右に一致を伸ばせない組である．この定義では短い一致も含まれるため通常ある一致長以上のものを出力する．図 6.2 は “Maximalexactmatch” と “alexthander” の間の MEM である．“alex” という文字列は互いに共

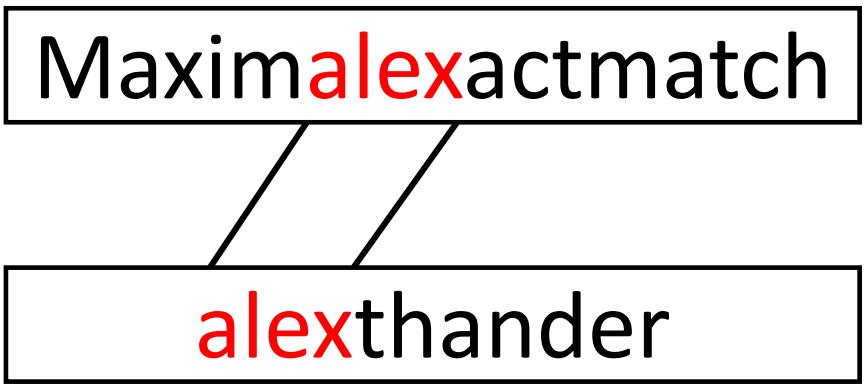


図 6.2: Maximal exact match の例

通し, かつそれ以上一致を伸ばすことができないためこれが MEM となる. また “a” や “e” も定義上 MEM となる.

MEM 検索のアルゴリズムにおいて便利な言葉を定義する.

- Right Maximal Exact Match (RMEM)
  文字列の右方向に一致を伸ばせない文字列のペア
- Left Maximal Exact Match (LMEM)
  逆に左方向に伸ばせない文字列のペア

次に二つの文字列間にある全ての MEM の計算について主流なアルゴリズムを紹介する. 対象の二つの文字列のうち片方に対して索引をつくり, もう片方の部分文字列を検索する. 索引を作った方の文字列を参照文字列 (R), 部分文字列出す方をクエリ文字列 (Q) と表すことにする. はじめに索引が参照文字列の任意の位置を計算できるものとして説明を行う. 検索索引の種類によってはある数の倍数の位置のみ返すものが存在するため, その時の計算もあとで紹介する. また, 以下の議論について参照文字列とクエリ文字列の立場を入れ替え, クエリ文字列のすべての位置から部分文字列を検索し始め, 参照文字列については $i = m(L + 1 - l)(0 \leq i <| R |)$ から始まる部分文字列のみで索引を作る方法でも成り立ち, E-MEM で用いられている手法である.

全ての MEM を検索する時, クエリ文字列の複数の位置から検索を始める. この検索開始位置は以下の MEM の最短の長さと完全一致の長さ, 検索索引が出力するデータに依存する. 検索開始からは大きく二段階に分かれる. はじめにある程度の長さ完全一致させる. その後完全一致により得られた候補に対して一致の長さを確認し, 確定する. ただし, 他の完全一致から得られた候補と重複するものは除去しなければならない. 部分文字列のクエリ文字列上での開始位置は実はクエリ文字列のすべての位置である必要はない. はじめの完全一致の長さを l, クエリ文字列では $Q[i, i + l - 1]$ が完全一致された部分文字列, MEM の最小の長さを L とする. この時, $i = m(L + 1 - l)(0 \leq i <| Q |)$ である. この i の取り方で十分であることを帰納法で証明する. MEM の左端に注目する. MEM の左端右端は対称であるから, 以下の議論では左端を考える. 右端は以下の左端を右端に変更すると同様に証明される. よってすべての左端がクエリ文字列中で網羅されていれば良いことになる. そこで $m = 0, 1$ の時を考える. $m = 0$ では $Q[1, i + l - 1]$ が左端

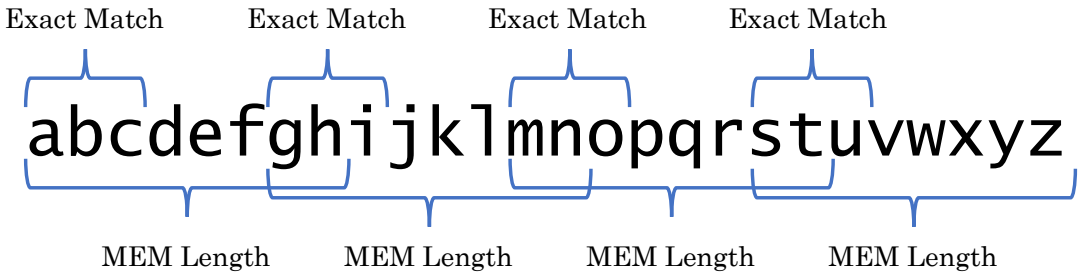


図 6.3: MEM 検索アルゴリズムの例. 最小の MEM の長さは 8, $l = 3$.

となる MEM が計算されないことになる. これらは $m = 1$ すなわち, $Q[1+L-l, i+L]$ の完全一致を行なった候補から得られる. 何故ならば, 左端が $Q[1]$ の最短の MEM は $Q[1, i+L]$ であるからである. 他の左端を持つものも $Q[1+L-l, i+L]$ を完全一致として持っているため網羅されている. $Q[i+l, L-l]$ を左端に持つ MEM もすべて $m = 1$ のもので網羅される. $m = n, n+1$ にしても同様である. はじめの完全一致の後は左右に一致を広げて MEM の長さを確定させるが, 片方の方向は別に行った検索のうち左隣の完全一致の始点まで比較することで十分である. ここで, 左方向が途中までの一致で良いとし, 右方向への一致は最後まで行うとする. この時 $i = (m-1)(L+1-l)$ の時と, $i = m(L+1-l)$ で計算される MEM を考える. 前者で計算され, 後者の完全一致の後候補となるものは, $Q[(m-1)(L+1-l), m(L+1-l)+l-1]$ を最低限一致しているものである. そのため, 後者を計算するときには左方向には $(m-1)(L+1-l)$ まで一致するものは前者で計算されているため計算する必要がなくなる. よって, 左側の一致は途中までで十分である. 上述のアルゴリズムですべての MEM を計算することができることをしめした. 具体的な索引の構造は第 8 章, 提案手法に譲る. 図 6.3 はクエリ文字列が "$abcdefghijklmnopqrstuvwxyz$", MEM の最小の長さが 8, $l = 3$ とした時の完全一致の部分文字列の位置である. このように検索開始位置はある程度飛ばすことができる. i の関係式から l が小さければ検索回数を減らすことができるが, 一般に l が小さくなるほど候補数が多くなってしまい後半の処理が重くなってしまうというトレードオフが存在することがわかる.

次に索引が K の倍数の位置のみ出力する時を考える. この時 $Q[i]$ が MEM の左端となるものを考えた時, 参照文字列で $Q[i]$ が K で割って p あまる位置 $(0 \leq p < K)$ 全てを計算しなければならない. 一回の検索では一つの p あまる位置のみが計算されるため, 同じ $Q[i]$ は K 回計算されなければならない. $l \leq L - K + 1$ とし, $i = m(L+1-l) + p(0 \leq i < \mid Q \mid, 0 \leq p < K)$ として計算すると全てを出すことができる. s は $l > 0$ がを満たす中の整数である. よって, K の倍数のみを出力する索引では同じ l でも K 倍の計算が必要となる.

第7章　文字列検索

本章では文字列検索のデータ構造とアルゴリズムを説明する．大規模な文字列を検索対象にむいたアプローチとして, 大規模な文字列を前もって検索しやすいデータ構造 (索引) にしておき, それを用いて検索を実行するものがある．このアプローチでよく用いられるのが, *suffix array* やそれを間引いた *sparse suffix array* に対して付随するデータ構造を用いたアルゴリズムである．今回の研究のライバル手法である essaMEM で用いられている *enhanced sparse suffix array (ESSA)*, 我々の手法で用いられている *fragmened Burrows-Wheeler transform (FBWT)* は sparse suffix array をベースにしたものである．sparse されていない suffix array に付属するデータ構造でそれぞれ対応するものが *enhanced suffix array (ESA)*, *Burrows-Wheeler transform (BWT)* である．以降の節では, まず suffix array の定義を示し, suffix array を用いた検索全体の目標を説明する．加えて sparse suffix array も紹介する．また, 以降の ESA, ESSA, BWT, FBWT との比較のため suffix array, sparse suffix array での検索も加えて説明する．そのあと ESA, ESSA, 最後に BWT, FBWT について述べる．

ここで, T をデータ構造をつくる元の文字列, n をその長さ, Q を T から検索したい文字列, q をその長さとする．$S[i]$ は文字列 S の $i+1$ 番目の文字を意味する．Σ は T または Q に現れる文字の集合である．また, 定義は suffix array の節で述べるが, ω-interval$[i..j]$ は文字列 ω に対応する suffix array の i から j の区間である．

7.1　suffix array, sparse suffix array と検索アルゴリズム

7.1.1　suffix array

suffix array は大規模な文字列に対して検索を行いたい場合に, その大規模な文字列に対して作られるデータ構造である．まず, suffix とは文字列の途中の場所から最後までの文字列である．suffix array の定義は, すべての suffix を鍵に, 各 suffix の先頭の位置を指す index を辞書順ソートした配列である．図 7.1 の suffix array 列に $T =$ “mississippi\$” の例を示す．“\$” は最小の文字とする．Burrows-Weeler transform との関連で, “\$” を末尾につけている．また suffix カラムでは suffix の後ろに prefix をつけているが, これも BWT の関係である．“\$” が末尾にあることにより, suffix の後ろの文字列は辞書順ソートにはかかわらないため, 図の suffix array は定義通りのものである．

suffix array を用いた検索を説明する．この検索の目標は ESA, BWT, ESSA, FBWT でも同じである．その suffix array を用いた検索の目標は, suffix の prefix が検索クエリ文字列と一致している suffix array の区間を特定することである．区間である理由は, suffix が辞書順ソートされているため, 検索クエリ文字列がもし存在していた場合, 検索クエリ文字列を prefix にもつ suffix が suffix array 上で連続に並ぶためである．この区間を Q-interval$[i..j]$ とする．

					child tab			occ tab				
index	suffix array	suffix	BWT	LCP	up	down	next	$	i	m	p	s
0	11	$mississippi	i	0			1	0	1	0	0	0
1	10	i$mississipp	p	0		2	5	0	1	0	1	0
2	7	ippi$mississ	s	1			3	0	1	0	2	1
3	4	issippi$miss	s	1		4		0	1	0	2	2
4	1	ississippi$m	m	3				0	1	1	2	2
5	0	mississippi$	$	0	2		6	1	1	1	2	2
6	9	pi$mississip	p	0		7	8	1	1	1	3	2
7	8	ppi$mississi	i	1				1	2	1	3	2
8	6	sippi$missis	s	0	7	10		1	2	1	3	4
9	3	sissippi$mis	s	2				1	2	1	3	5
10	5	ssippi$missi	i	1	9	11		1	3	1	3	5
11	2	ssissippi$mi	i	3				1	4	1	3	5

図 7.1: Suffix array, Burrows-Wheeler transform, longest common prefix, and child table の “mississippi” に対する例.

付属させたデータ構造なしでも suffix array は全文検索が可能である. 以降では suffix array にいくつかデータ構造を付加してアルゴリズム面での高速化をはかっているため, suffix array での全文検索についても説明する. suffix array の各要素と対応する suffix は上でも述べているようにソートされている. そのため suffix array と元の文を用いることで二分探索が可能となる. これにより区間を求める. 時間計算量は $O(q \log n)$ である. 図 7.1 の例で $Q =$ “si” という文字列を検索した時の結果は, suffix array の 8 番から 9 番の区間ということになる.

7.1.2　sparse suffix array

suffix array は 0 から $n-1$ までの整数を要素とするためメモリ量が大きくなってしまう. そこでメモリ削減のために suffix array のうち与えられた k の倍数の要素のみで構成された配列を用意して行う. もちろん k の倍数の位置で始まるものしか検索で出てこなくなってしまう. q が k よりも大きい場合は Q の途中から検索し始めて検索でててきたところを比較することで補い全文検索を行う. そのため, 時間計算量は $O(kq \log n)$ となり k 倍かかる.

7.2　enhanced suffix array, enhanced sparse suffix array と検索アルゴリズム

ESA は suffix array の区間を求めるのに二分探索を行わずにできるデータ構造を加えることによって高速化している. 区間を求める時間計算量は $O(q)$ である. suffix array の検索ではクエリ文

字列の先頭から処理していき一致する区間を更新していく．この区間の更新が二分探索だったものを定数時間で行うのがこのデータ構造である．直観的な説明を初めに行う．区間を更新する前後に注目する．前の区間の中の suffix の prefix ですべて共通する部分と異なり始める部分が存在する．その異なり始める位置がで次の区間となる．このデータ構造はこの次の区間の探索を現実的なメモリ量で実現している．

ESA が suffix array に加え持つデータ構造は *Longest common prefix (LCP)* array と child table と呼ばれるものである．LCP array はソートされたすべての suffix のリストで一つ前の suffix と先頭何文字が一致しているかを格納した配列である．図 7.1 の LCP 列が $T =$ “mississippi\$” の LCP array である．child table は up, down, nextIndex の 3 列で構成されている．それぞれ以下のように定義される．

$$\begin{aligned}\mathrm{child}[i].\mathrm{up} := \min\{q \in [0..i-1] \mid lcp[q] > lcp[i] \\ \text{and } \forall k \in [q+1..i-1] : lcp[k] \geq lcp[q]\}\end{aligned} \tag{7.1}$$

$$\begin{aligned}\mathrm{child}[i].\mathrm{down} := \max\{q \in [i+1..n] \mid lcp[q] > lcp[i] \\ \text{and } \forall \in [i+1..q-1] : lcp[k] > lcp[q]\}\end{aligned} \tag{7.2}$$

$$\begin{aligned}\mathrm{child}[i].\mathrm{nextIndex} := \min\{lcp[q] = lcp[i] \\ \text{and } \forall k \in [i+1..q-1] : lcp[k] > lcp[i]\}\end{aligned} \tag{7.3}$$

up と down は次の更新先の候補の区間のうち一番辞書順で早い区間の下端を示す．nextIndex は更新先の候補の次の位置を示す．これらの情報から全ての更新先候補を知ることができるので元の文字列にアクセスし文字を比較することで最終的に区間を更新する．このアルゴリズムは文字種類が多い時にはあまり有効ではないがゲノムのように少数の文字で構成される文字列については有効である．child table は一見 3 列も存在し大きくなるように見えるが, 実は 1 列におさめることができる．これにより現実的なメモリ消費量に抑えることができている．

メモリ消費量は, suffix array の一要素を 4byte で表現した時元の文字列 (n byte) と LCP array(mn byte), child array($4n$ byte), suffix array($4n$ byte) である．LCP array については小さい数字が多いため基本 1byte で表現し, 大きいものは別のデータ構造に入れているためこのような表記になっている．通常 m は 1 から 3 程度である．よって合計は $(9+m)n$ byte である．

enhanced sparse suffix array は sparse suffix array を LCP array と child table を作ったものである．メモリ消費量は $(1+(m+8)/k)$ byte である．

7.3 Burrows-Wheeler transform, fragmented Burrows-Wheeler transform と検索アルゴリズム

7.3.1 Burrows-Wheeler transform

BWT は T を変換した長さ n の文字列が出力である．そのため文字が小さいと BWT の文字列も小さくエンコードでき全体としてメモリが小さくできる可能性がある．BWT の定義は以下の

				occ tab				
index	suffix array	suffix	BWT	$	i	m	p	s
0	11	$mississippi	i	0	1	0	0	0
1	10	i$mississipp	p	0	1	0	1	0
2	7	ippi$mississ	s	0	1	0	2	1
3	4	issippi$miss	s	0	1	0	2	2
4	1	ississippi$m	m	0	1	1	2	2
5	0	mississippi$	$	1	1	1	2	2
6	9	pi$mississip	p	1	1	1	3	2
7	8	ppi$mississi	i	1	2	1	3	2
8	6	sippi$missis	s	1	2	1	3	4
9	3	sissippi$mis	s	1	2	1	3	5
10	5	ssippi$missi	i	1	3	1	3	5
11	2	ssissippi$mi	i	1	4	1	3	5

図 7.2: “mississippi” に対する BWT を基とする索引のデータ構造

数式である. $BWT[i]$ は BWT の文字列の i 番目, $SA[i]$ は suffix array の i 番目である.

$$BWT[i] = \begin{cases} T[SA[i]-1] & (i > 0) \\ T[n-1] & (i = 0) \end{cases} \tag{7.4}$$

図 7.1 の BWT の列は $T =$ “mississippi\$” に対する BWT 文字列を示す.

BWT を用いた全文検索を説明する. BWT による suffix array の区間計算の時間計算量は ESA 同様 $O(q)$ でできる. この計算ははクエリ文字列の後ろの文字から検索を行う. ω-interval$[i..j]$ がすでに一致した文字列と区間, c が次の文字の時, ソースコード 7.1 によって, $c\omega$-interval を決定することができる. ここで $C[c]$ は T 中の文字 c の個数, $Occ(c,i)$ は T の BWT 文字列の i 番目までの c より小さい文字の個数である.

List 7.1: Given $c \in \Sigma$ and ω-interval$[i..j]$, $backwardSearch(c,[i..j])$ returns $c\omega$-interval if exists, and NULL otherwise.

```
1  backwardSearch(c,[i..j])
2    i ← C[c] + Occ(c,i-1) + 1
3    j ← C[c] + Occ(c,j)
4    if i ≤ j then return [i..j]
5    else return NULL
```

$Occ(c,i)$ は BWT 文字列のみが存在すればできるが, 実行時間的にはよくない. しかし, すべての i に対してすべての文字の個数を持つテーブルは大きくなりすぎてしまう (7.1). そこである程度の間隔で各文字の個数をとっておき, その間は BWT 文字列で保存しておく. 計算するときは保存した文字の個数と BWT 文字列を数えることで行う. 図 7.2 は “mississippi” に対して作られた

FSA0/3	suffix	FBWT0/3
0	mississippi$	$
9	pi$mississip	p
6	sippi$missis	s
3	sissippi$mis	s

FSA1/3	suffix	FBWT1/3
10	i$mississipp	p
7	ippi$mississ	s
4	issippi$miss	s
1	ississippi$m	m

FSA2/3	suffix	FBWT2/3
11	$mississippi	i
8	ppi$mississi	i
5	ssippi$missi	i
2	ssissippi$mi	i

図 7.3: Fragmented suffix array and fragmented BWT の “mississippi” に対する例.

索引のデータ構造である. 灰色になっているところは実際には存在しないものである. occ のテーブルはここでは 3 つに一つ取られていて, 間のものは BWT の文字列を数えることで補う.

7.3.2　fragmened Burrows-Wheeler transform

この章の初めで述べた通り FBWT は sparse suffix array を補助する BWT ととらえることもできるが, 定義は *fragmented suffix array (FSA)* というデータ構造により行われる. そのため本節では FSA の定義をはじめにし, FBWT の定義を行う. そして FBWT による検索アルゴリズムも示す.

FSA の定義を述べる. FSA は suffix array と同様に T に対して作られるが, T に最小の文字 “$” を一つ以上末尾につけ長さを与えられた k の倍数にした $T_\$$ に対して作られる. この $T_\$$ の長さを $n_\$$ とする. FSA は k 個の配列で構成され, 各配列はそれぞれ k で割って l 余る位置から始まるすべての suffix を鍵に, その suffix の先頭 index をソートした配列である. k で割って l 余る位置のものの配列を $FSA_{l/k}$ と表記する. 図 7.3 の $FSA_{[012]/3}$ は $T =$ “mississippi” に対して $k = 3$ で作られた FSA である.

FBWT は FSA を使い以下のように定義される.

$$FBWT_{l/k}[i] = \begin{cases} T_\$[SA_{l/k}[i] - 1] & (i > 0) \\ T_\$[n_\$ - 1] & (i = 0) \end{cases} \tag{7.5}$$

図 7.3 の $FBWT_{[012]/3}$ は $T =$ “mississippi” に対して $k = 3$ で作られた FBWT である.

FBWT を用いた全文検索について述べる. FBWT も BWT と同様クエリ文字列の後ろから backward search により区間を狭めていく. ただし, FSA の全配列に対して区間を求めなければならないため, k 倍計算が多くなる. 文字列 ω に一致している $FSA_{l/k}$ 上の区間を ω-interval$_l[i..j]$ とする. c を次の文字としたときソースコード 7.2 に示される backward seach は ω-interval$_l[i..j]$ から $c\omega$-interval$_{l-1}$ を返す. ここで $C[l][c]$ は $FBWT_{l/k}$ の c の個数を返す. $Occ[l](c, i)$ は $FBWT_{l/k}$ の i 番目までの c より小さい文字の数を返す.

List 7.2: Given $c \in \Sigma$ and ω-interval$_l[i..j]$, $backwardSearch(c, l, [i..j])$ returns $c\omega$-interval$_{l-1}$ if exists, and NULL otherwise.

```
1  backwardSearch(c,l,[i..j])
2    i ← C[l][c] + Occ[l](c,i-1) + 1
3    j ← C[l][c] + Occ[l](c,j)
4    if i ≤ j then return [i..j]
5    else return NULL
```

第8章　関連研究

本章では6.2節で紹介したアルゴリズムを実現するデータ構造とアルゴリズムの関連研究を紹介する．索引の種類は大分するとハッシュによる方法とsuffix arrayによる方法がある．またアルゴリズムのアプローチとしては, 索引で出る位置の個数を多くするか, 少なくするかという方法でも分かれる．我々の手法はsuffix arrayを用いた, 索引で出る位置の個数が多いものと分類できる．本章では我々の手法と密接に関係する手法と, 現在の, suffix arrayを用いた検索で出る位置の個数が多いものと分類できるessaMEM[29]と, ハッシュを用いた検索で出る位置の個数が少ないものと分類できるE-MEM[14]という二つのstate of the artとされるアルゴリズムについて紹介する．

本章では最小のMEMの長さをLとする．索引を作った文字列を参照文字列, 参照文字列に対してマッチングする文字列をクエリ文字列と言い表す．

8.1　essaMEM

essaMEMはsparseMEM[13]のアルゴリズムを改良したアルゴリズムである．検索時間のみを見た時最も高速なアルゴリズムとなっている．索引をsparse suffix arrayからenhanced sparse suffix arrayとしたものである．essaMEMではsparseMEMのアルゴリズムを再現しているが, 検索を実現している複数のアルゴリズムのうち一部のみを使用すると高速であることを発見している．この節ではその高速なアルゴリズムを紹介する．またこのアルゴリズムは本研究で提案する手法の母体となっている．

enhanced sparse suffix arrayは7.2節で述べた通りsparse suffix arrayで行われる二分探索を定数時間で行うことができるデータ構造である．また, 6.2節で述べた通りMEMの計算アルゴリズムははじめの完全一致の長さによって候補数と検索回数のトレードオフが存在する．そこでessaMEMのアルゴリズムでははじめの長さを$L - sK + 1$としている．sはskipパラメータと呼ばれ, はじめの長さを調節できるパラメータ, Kはsparseのパラメータ, Lは最短のMEMの長さである．はじめの長さを調節できることによって高速化を図っている．論文中では$L - sK + 1 \geq 10$を満たす中でsが最大となる時が実験上最速であるとしている．具体的な計算は, はじめの完全一致をESSAで行い, 次の候補から確定させる段階をまずESSAで候補がなくなるまで行う(RMEM確定)．その後, はじめの候補の区間から左方向に一致を伸ばして確定させる(LMEM確定)．時間計算量ははじめの完全一致で$O(l)$, 次の候補からMEMの長さをそれぞれ確定させる段階が$O(occ * K + maxlen)$である．lははじめの完全一致の長さ, occは完全一致から計算された候補数, $maxlen$はRMEMを計算する時に行う中で最も長い一致の長さである．

本研究はessaMEMを母体とするためメモリ使用量についても比較のため記す．参照文字列の長さをn, sparseパラメータをKとした時, LCP arrayの要素はほとんどが1byteで表現されるため, それより大きいものについては別のデータ構造に入れているので, mn/K byteである．mは

通常 1 から 3 程度である．child table は一要素 4byte で表現されるため $4n/K$ byte, 参照文字列は 1byte 文字で表現しているため n byte である．sparse suffix array は $4n/K$ byte である．また, はじめの完全一致のはじめの数文字はハッシュによって計算されている．このハッシュは ω-interval を返すため, $2 \times 42^{2hashlen} = 2^{2hashlen+3}$ byte である．ハッシュの鍵は ATGC を 2bit エンコードされたヌクレオチドである．また, $hashlen$ はハッシュの鍵として使用しているヌクレオチドの個数である．よって合計は $(m+8)n/K + n + 2^{hashlen+3}$ byte である．

この手法ではのちの E-MEM よりも索引作成時間がかかるが, クエリ文字列を検索する時間は少なくすることが可能になっている．そのため, 一つのゲノムに対して多数のゲノムを比較したいときに有利なアルゴリズムとなっている．

8.2 E-MEM

E-MEM は索引にハッシュを用いたもので, 二つのゲノム間の MEM 全てを出力するとき, ハッシュの作成から検索まで全体の計算時間が最も高速なアルゴリズムとなっている．この手法は参照文字列からハッシュを作成する．essaMEM と違い, 参照文字列の部分文字列を飛び飛びでとる一方, クエリ文字列は全ての位置から部分文字列を取っていく手法である．ハッシュの鍵には $k = 28$ 個のヌクレオチドを 2bit エンコードしたものを用いる．参照文字列は $L - k + 1$ の倍数の位置から始まる k 個のヌクレオチドとその位置で作成される．

この手法では参照文字列から索引を作る時間を小さくすることができ, また省メモリである．しかし, クエリ文字列の検索開始位置は多いためそちらで遅くなる．そのため, ゲノムを二つのみ比べたい場合有利なアルゴリズムとなっている．

第9章　予備実験

本章では本研究でさらに改良する essaMEM について予備実験の結果を記す．essaMEM では 8.1 節で記した通り skip パラメータ s によりはじめの Exact Match の長さを静的に調整できる．そこで s に対する実行時間を計測した．計測環境は Intel Xeon CPU E5-2690 上で動く Red Hat Enterprise Linux server である．メモリは十分存在した．データセットには simulans (ハエの一種) と sechellia (ハエの一種), melanogaster (ハエの一種) と yakuba (ハエの一種), ヒト (hg19) とチンパンジー (panTro3), マウス (Mus musculus) (mm10) とヒトのゲノム間の MEM の計算を用いた．これらは Mbps, Gbps のオーダーの大きさのゲノムでありまた, MEM の出現個数の少ないものと多いものという組み合わせである．表 9.2 に各ゲノムの大きさと MEM の出現個数, また以下で議論される最短の実行時間を実現する skip パラメータと実行時間, skip パラメータ最大の時の skip パラメータと実行時間を示す．ハエのゲノムは Mbps オーダーのもので, simulans と sechellia は比較的多く MEM が出現している．melanogaster と yakuba は比較的少ない．一方, ヒトとチンパンジー, マウスは Gbps のオーダーのもので, ヒトとチンパンジーは比較的多く MEM が出現し, マウスとヒトは比較的少ない．これらの特徴を持つゲノムを網羅できれば実用にも叶うのではないかと考えられる．

図 figs. 9.1 to 9.4 はそれぞれ, 各データセットを skip パラメータを変化させた時の実行時間を表したものである．表 9.1 にそれぞれのデータの長さ, 最短の MEM の長さ, MEM の個数と平均の MEM の長さを記した．ハエの比較では, sparse パラメータは 1, マウスとヒト, ヒトとチンパンジーでは 3 とした．全ての結果に共通な性質は, skip パラメータが小さい時と大きい時には実行時間が比較的大きくなっていることである．essaMEM の論文では skip パラメータは取りうるうち最大, つまり最初の Exact Match は最短を選ぶとよいとされていたが, それとは異なる結果となった．表 9.2 から最速時の skip パラメータの時と最大の skip パラメータの時の実行時間に注目する．それぞれのデータセットについて最速の skip パラメータは最大でなくまた著しく性能が低下しているものもある．データによって skip パラメータは少し異なり事前に最速の skip パラメータを知ることが難しい．

skip パラメータが大きい時に性能が悪くなっている理由には, 6.2 節で述べた通り, MEM の計算アルゴリズムのはじめの Exact Match により計算される候補数が非常に多くなり, 後段が遅くなったことが考えられる．そのため, マッチの長さを工夫することでこの現象を抑えることができれば高速な動作を安定して高速なアルゴリズムを作成することが期待できる．

表 9.1: 各データセットの属性. len は長さを示す

id	reference	size[bp]	query	size[bp]	MEM len	# of MEMs	Avg. len of MEMs
1	simulans	135M	sechellia	162M	50	18643313	70.2097
2	melanogaster	140M	yakuba	162M	50	870653	82.0591
3	hg19	3.1G	panTro3	3.2G	100	132368058	127.01
4	mm10	2.7G	hg19	3.1G	100	554327	114.753

表 9.2: essaMEM の skip パラメータのうち最も高速に動作したもの

dataset id	skip (fastest)	fastest time[s]	skip (largest)	time[s] (largest skip param)
1	34	26.7778	40	54.2306
2	34	12.8621	40	48.8478
3	18	763.369	30	38685.2
4	18	442.055	30	18564.6

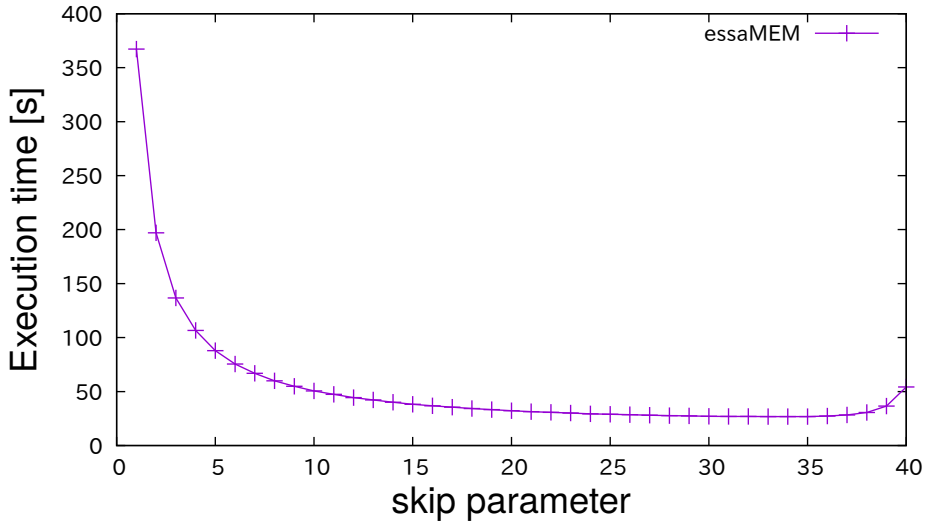


図 9.1: No. 1 のデータセットでの essaMEM の skip パラメータと実行時間の対応

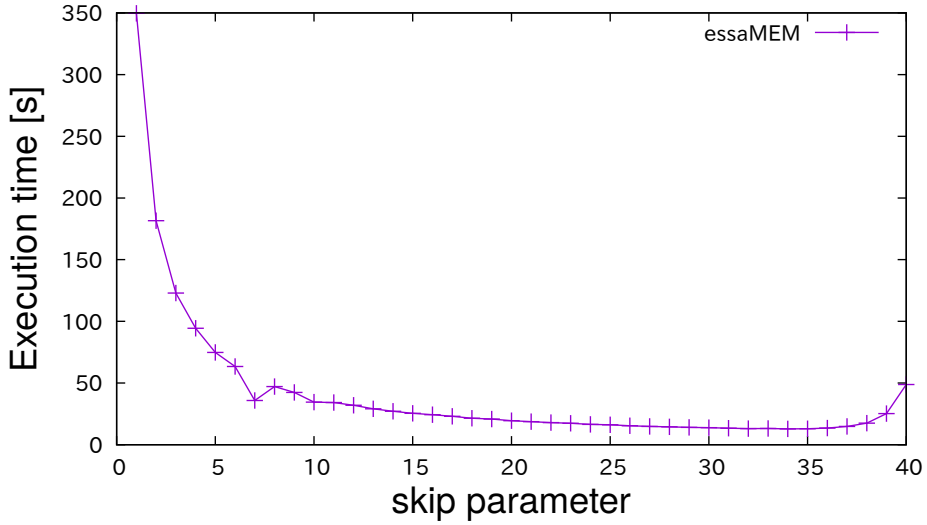


図 9.2: No. 2 のデータセットでの essaMEM の skip パラメータと実行時間の対応

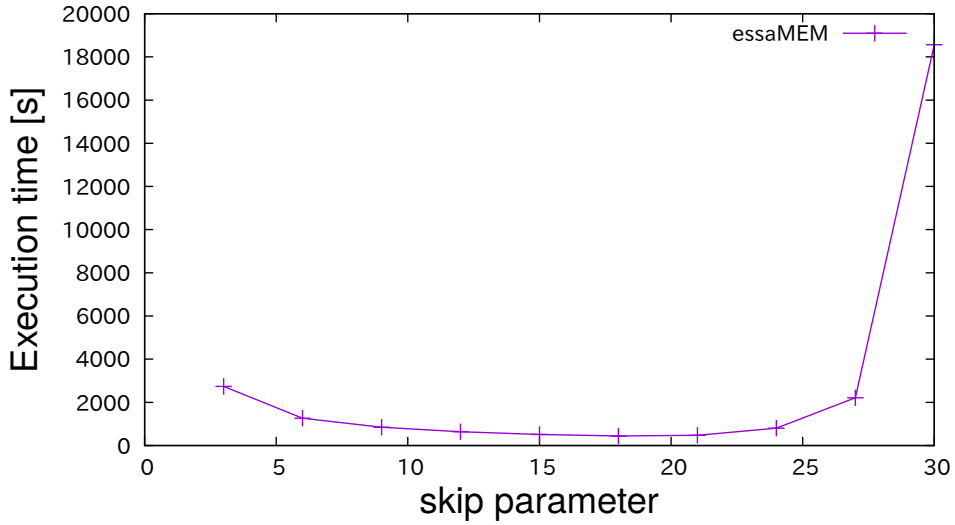


図 9.3: No. 3 のデータセットでの essaMEM の skip パラメータと実行時間の対応

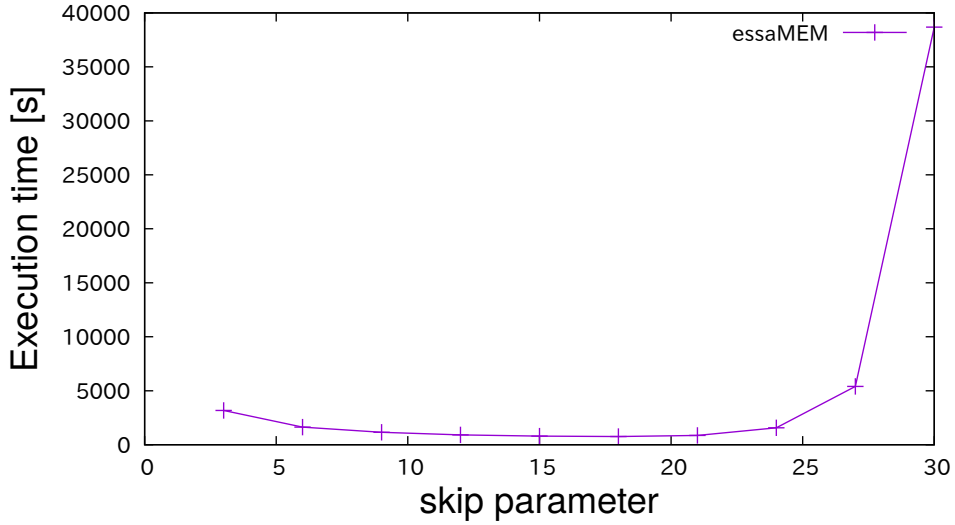


図 9.4: No. 4 のデータセットでの essaMEM の skip パラメータと実行時間の対応

第10章　手法

この章では本研究で提案する手法を説明する．また我々の手法を FBWTMEM と呼ぶ．FBWTMEM は essaMEM のアルゴリズムをさらに改善したものである．essaMEM との変更点は 2 つある．一つ目に索引を ESSA から FBWT にした点．もう一つは skip パラメータの自動チューニング手法である．一つ目の索引の変更はメモリの削減が目的である．二つ目の自動チューニング手法は高速化が目的である．

はじめに FBWT でのデータ構造とサイズを示す．今回のデータ構造は FBWT と Occ テーブルが融合したテーブル, ハッシュテーブル, C テーブル, FSA_0, 参照ゲノムである．n を参照ゲノムの長さとする．FBWT と Occ テーブルの構造について述べる．Occ テーブルの全要素を格納すると膨大なメモリが必要となる．そのため, ヌクレオチド ATGC のそれぞれの個数を 128 個飛びでサンプリングし, それぞれ 32bit で格納する．サンプリングされていない部分は FBWT から計算する．FBWT は ATGCN の 5 文字からなるので一要素 3bit で表現している．メモリアラインは (64 ヌクレオチド, ATGC それぞれのここまでの個数, 64 ヌクレオチド) の 512bit 一要素として並んだ構造をしている．これにより, Occ テーブルへのアクセス時にはヌクレオチドの個数とヌクレオチド一要素中のヌクレオチド数を一つのキャッシュラインで計算できるようになっている．図 10.1 はこのメモリアラインを図にしたものである．このテーブルのサイズは参照文字列の長さが n とした時 $3n/8 + 4 \times 4n/128 = n/2$ byte となる．ハッシュテーブルははじめの完全一致のうちのはじめの数文字 ω に対応する ω-interval$_l[i..j]$ を計算するものである．ハッシュの鍵は ATGC を 2bit エンコードされたヌクレオチドである．N を入れていない理由は, MEM の計算時には MEM の右端や左端として扱うためハッシュにアクセスする時点で鍵に N が含まれていた場合 MEM ではないとして弾き, ハッシュの鍵として含まれることが無いからである．このハッシュテーブルは全ての $FBWT_{l/K}$ に対して作成されるため $2 \times 4 \times 2^{2hashlen} K = 2^{2hashlen+3} K$ byte である．FSA_0 のみであるのはメモリを削減するためである．サイズは $4n/K$ byte である．C テーブルは各 $FBWT_{l/K}$ に存在する ATGC それぞれの個数であるため $4 \times 4 \times 4K = 64K$ byte である．これは参照ゲノムの長さに対して無視できる程度の大きさである．参照ゲノムは 4bit でエンコードしているため $n/2$ byte である．合計すると $4n/K + n + 2^{2hashlen+3} K$ byte である．sparse のパラメータが小さい時 essaMEM の $(m+8)n/K + n + 2^{hashlen+3}$ byte より小さくなる．また sparse のパラメータが大きい時でも $hashlen$ を調整することで essaMEM よりも小さくすることができる．

次に skip パラメータの自動チューニング抜きの MEM 計算のアルゴリズムを示す．ESSA ではクエリ文字列を先頭から計算していくが, FBWT では末尾から計算していくため, クエリの処理は essaMEM と逆の順番に処理される．それ以外の全体の流れは変わらない．MEM 計算のはじめの完全一致については $L - sK + 1$ だけ 7.3.2 節で表したアルゴリズムで計算する．ただし, 計算終了時に ω-interval が $FBWT_{0/K}$ 上のものをさすようにする．後半のそれぞれの候補が MEM である

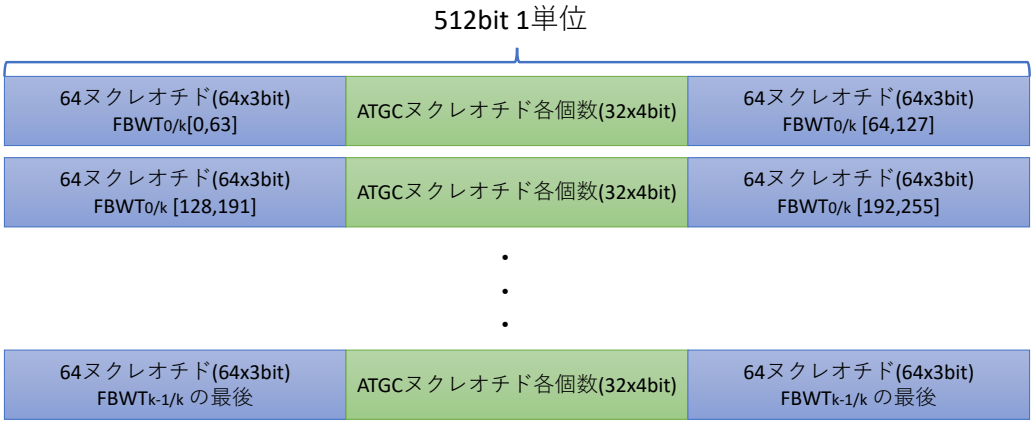


図 10.1: Occ テーブルと FBWT のメモリ配置

FSA0/2	suffix	FBWT0/2		FSA1/2	suffix	FBWT0/2
12	$$bacanabanaca	a		13	$bacanabanaca$	$
0	bacanabanaca$$	$		11	a$$bacanabanac	c
6	banaca$$bacana	a		5	abanaca$$bacan	n
10	ca$$bacanabana	a		9	aca$$bacanaban	n
2	canabanaca$$ba	a		1	acanabanaca$$b	b
4	nabanaca$$baca	a		3	anabanaca$$bac	c
8	naca$$bacanaba	a		7	anaca$$bacanab	b

図 10.2: backward search による MEM 特定の段の例．’a’-interval が与えられ, ’ca’-interval に更新する．更新後の区間の中の suffix array の要素の並び順は更新前の対応する要素と同じである．

かを調べる段階では, クエリの左方向の比較は索引を使ってできる．右側は参照文字列と比較して行う．左方向の索引を用いた検索を述べる．FBWT の各文字はその要素に対応する suffix の一つ前の文字である．そのため, interval の中の文字がクエリ文字列の次の文字と一致しているかどうかを調べることで左方向にその候補が伸びるか否かを知ることができる．そこで左方向の検索は索引を用いて interval を更新し, 更新毎に interval 内の FBWT の文字とクエリの文字を比較することにより行う．FSA の値の取り扱いについて述べる．$FSA_{0/k}$ のみ保存しているため, backward search を行い $FBWT_{l/k}(l \neq 0)$ の時 interval 中の FSA の値をとることができず, 参照文字列上の位置すぐに計算できない．しかし, interval の更新時に FBWT の文字とクエリ文字列の文字が同じだったものに対応する FSA の要素はその順番を保ちつつ更新後の interval の中の FSA の値になる．これは, 更新後の interval の中に対応する suffix は更新前のものの suffix に同じ文字を先頭につけたものでありかつ, FSA に対応する suffix はソートされているため, 更新後の残る候補の順番は変わらないのである．図 10.2 は “bacanabanaca$$” という文字列の $K = 2$ で FBWT を使用した索引について, 文字列 “a” に対応する interval とその FSA がわかっていたときの “ca” の interval とその

FSA の対応である．“a”-interval を更新し “ca”-interval としたとき, 残った候補は “a”-interval 中で存在した順序と同じ順序で並んでいることを示している．青い矢印が更新前後で対応する候補である．図からわかるように, “ca” から始まる suffix の後ろは “$$..”, “na..” である．これは, “a” の interval 中でも “$$..”, “na..” により順序が決まるため成り立つ．これにより全ての候補について左端右端を調べることができる．この方法の時間計算量は, 初めの完全一致で $O(l)$, 次の候補から MEM の長さをそれぞれ確定させる段階がおおよそ $O(occ * K + occ * maxlen)$ である．各変数は 8.1 節と同じである．essaMEM の MEM の長さを確定させる段では $O(occ * K + maxlen)$ であり後段は不利である．しかし, 実験上この不利は無視できる程度であった．

skip パラメータの自動チューニング抜きのアルゴリズムの全体像は上述の通りであるが, 実際には一つヒューリスティックスがある．それは, 候補数がある程度少なくなった場合参照ゲノムと比較するようにすることである．これにより少数の長い MEM が存在した場合の計算コストを抑えることに成功している． ソースコード 10.1 はアルゴリズムの後半の疑似コードである．$collectMEM$ に前段で計算した interval を入れる．l-$[i..j]$ は $FBWT_{l/K}$ 上の interval を示す．$checkCandidates$ が interval 中の FBWT の文字とクエリ文字を比較して左側の食い違い点を計算している．$leftAndRightSide$ は候補数が少なくなった際直接参照ゲノムと比較する部分である．$rightSide$ は右側の食い違い点を計算している関数である．MEM に発見された MEM が保存される．

List 10.1: Calculate MEM, given 0-$[i..j]$

```
 1  rightSide(sa,queryStartPos,leftMatchLength)
 2  // candidates which match k·s characters are found in previous MEM search
 3   for itr in [sa..sa + k· s]
 4     if query[queryStartPos + itr - sa] != reference[itr] then // compare reference and
          query
 5       if leftMatchLength + L - k· s + 1 + itr - sa \geq L then
 6       MEM.push(sa,leftMatchLength + L - k· s + 1 + itr - sa) // position of MEM and its
           length
 7
 8 leftAndRightSide(query,queryPos,candidate,l,matchLength)
 9   for sa in candidate
10     // left side match
11     countLeftside = 0
12     while query[queryPos - countLeftside] == reference[sa - countLeftside]
13       countLeftside++
14     rightSide(sa + matchLength, queryPos + matchLength, matchLength + countLeftside)
15
16
17 checkCandidates(c,l-[i..j],queryStartPos,leftMatchLength)
18   for itr in [i..j]
19     if FBWT_{l/k}[itr] = c then // this candidate can not extend to left
20       nextCandidateSA.push(currentCandidateSA[itr - i])
21     else // this candidate can extend more to left
22       rightSide(currentCandidateSA[itr - i], queryStartPos, leftMatchLength)
23   return nextCandidateSA
24
25 collectMEM(0-[i..j],queryStartSearchPos)
26   l-[i..j] ← 0-[i..j]
27   candidate ← FSA_{0/k}[i..j]
28   queryPos ← queryStartSearchPos
29   while !candidate.empty()
```

```
30      if j - i + 1 <= directlyCompareThreashold
31        leftAndRightSide(query,queryPos,candidate,l,queryStartSearchPOs - queryPos)
32        return
33      else
34        candidate ← checkCandidates(query[queryPos], l-[i..j],
35                                    candidate, queryStartSearchPos - queryPos)
36        l-[i..j] ← backwardSearch(query[queryPos], l-[i..j])
37        queryPos = queryPos - 1
```

最後に skip パラメータの自動チューニング手法を説明する．ここまでではessaMEM 同様に 9 章で述べた, skip パラメータの調節に難がある．そこで, 我々は skip パラメータを自動的に調節する手法を提案する．skip パラメータははじめの完全一致の長さを調節し, 高速化を測るものであった．この高速化が実現できる理由は, 候補数が多くなってしまうと後段の MEM を確定させる部分が遅くなってしまうことである．そこで, はじめの完全一致においてある程度候補が少なくなるまで動的に完全一致を長くする．具体的には候補数が事前に決めた閾値よりも多くなった場合一時的に skip パラメータを小さくするのである．ただし, 完全一致を長くすると一部 MEM が計算されなくなる．具体的な対策は以下のアルゴリズムによる．完全一致がクエリ文字列 Q の i 番目から始まったとする．デフォルトの skip パラメータを s とする．その後 $L-(s-s')K+1$ だけ完全一致したとする．この時 $Q[j](i-(L-k(s-s')+1)+1<j<i-(L-sk+1))$ から始まるか, $Q[j](i-(L-k(s-s')+1)<j<i+ks-1)$ で終わる MEM が検索されない．そこで, $Q[i-L+ks:i+k(s-s')]$ をはじめに完全一致する MEM 計算をすることで過不足なく MEM を計算することができる．ここで, s を小さくしすぎると, 補間するための計算での完全一致部分が少なくなりすぎ, そちらで候補数が多くなってしまう．そこで s の半分より大きい範囲で小さくすることにした．

第11章　評価

本章でははじめにFBWTMEMをessaMEMと比較しつつ優位性を示していく．次にessaMEM同様検索において高速な既存アルゴリズムと比較しする．最後に, 索引からゲノム比較まで全てを行うE-MEMとの比較を行う．E-MEMとの比較以外はIntel Xeon CPU E5-2690上で動くRed Hat Enterprise Linux serverである．メモリは十分存在した．E-MEMとの比較はIntel Xeon CPU E5-2660上で動くUbuntu 14.04 Linux serverである．こちらもメモリは十分存在した．suffix arrayの構築はhttps://sites.google.com/site/yuta256/saisを用いている．

図11.1, 11.2 ,11.3, 11.4は9章で示したデータセットを用い, essaMEM, FBWTに索引を変えただけのヒューリスティックスなしのもの, FBWTに索引を変え候補数が閾値よりも少なくなった時に参照ゲノムと比較するヒューリスティックスを加えたもの, さらにskipパラメータを自動チューニングする手法を加えたものについて初期のskipパラメータを変更した時の実行時間である．No. 1, 2のデータセットに対してはessaMEMとFBWTのsparseパラメータは1, No. 3, 4は3である．表11.1は今回の比較実験で使用されたFBWTMEMとessaMEMそれぞれのメモリサイズである．FBWTMEMのヒューリスティックスはメモリ使用量に関わらないため全てこの値である．この表からFBWTMEMの方がessaMEMよりも半分程度のメモリで済んでいることがわかる．次に索引をFBWTに変えただけのものとessaMEMの比較について述べる．実験結果の図から分かる通りessaMEMとほぼ同じ傾向を示し, 10節で述べた, MEM候補からMEMの長さを計算する段での不利な部分も隠れており, 実行時間もそれぞれのskipパラメータに対して五分五分の性能を出すことに成功している．参照ゲノムと直接比較するヒューリスティックスを加えたものは, No. 1のデータセットでは1.3倍程度の高速化が実現された．他のデータセットではほとんど変わらない性能であった．skipパラメータの自動チューニングを入れたものはskipパラメータが大きくなっても性能が大きくは悪化せず最速値と遜色ない値になった．No. 1のデータセットではskipパラメータ最大のときessaMEMよりも1.98倍, No. 2では5.22倍, No. 3では80.8倍, No. 4では62.7倍の高速化を実現できた．essaMEMの最速値との比較は, No. 1で0.977倍, No. 2で1.375倍, No. 3で1.401倍, No. 4で1.925倍高速に動作した．付録Aにはほかのゲノムでのデータも記載している．

以上より, メモリ使用量の面でも実行時間の面でも各データセットでessaMEMよりも優れた結果となった．9章でも述べた通り, これらのデータセットはそれぞれ別の特徴を持っているため, これら全てで良い性能が出たので実用でも十分essaMEMを超えることが期待できる．

次にほかの検索において重点を置いたアルゴリズムについて比較する．比較したアルゴリズムは, essaMEM, backwardMEM[21], slaMEM[10], E-MEMである．essaMEMのコマンドラインオプションは, -n, -child 1, -suflink 0, -kmer 10, -maxmatch and -skip <optimal> である．optimalは, essaMEMのskipパラメータのうち最も高速に動作した時のskipパラメータである．つまり, essaMEMはパラメータのうち最速のものを示している．sparseパラメータはデータセット1, 2

表 11.1: FBWTMEM と essaMEM のメモリ使用量. id 列は表 9.1 に対応する.

id	s	FBWTMEM size [kB]	essaMEM size [kB]
1	1	713144	1553324
2	1	743516	1606248
3	3	7812356	13899224
4	3	6671612	12154212

については 1, 2, 4, 8, 16, 32 である. データセット 3, 4 については 3, 4, 8, 16, 32, 64 である. backwardMEM にも sparse パラメータが存在し, 1, 2, 4, 8, 16, 32 を用いた. ただし, データセット 3, 4 については実行時間が長すぎたため計測を行わなかった. slaMEM はデフォルトの設定のみで計測した. E-MEM もデフォルトの設定のみで計測した. FBWTMEM は essaMEM と同じ sparse パラメータである. また, この実験では全てのヒューリスティックスを有効にしている. skip パラメータはデフォルトで行なっている. ハッシュに使用する鍵の長さは, ハッシュテーブルの大きさが $2 \cdot 4^1 1$ を超えないような長さにした. これら全ては 1 スレッドで動作を確かめている. データセットは 9 章と同一である.

図 11.5, 11.6, 11.7, 11.8 が結果である. x 軸がメモリ使用量, y 軸が実行時間である. すべてのアルゴリズムは検索の計算時間とメモリ使用量である. E-MEM についても検索時のものを示している. それぞれのアルゴリズムについて結果を比較すると,

- backwardMEM
  全てのデータについて, メモリ使用量, 実行時間全てにおいて FBWTMEM が上回る結果を示した.
- slaMEM
  FBWTMEM, E-MEM, essaMEM と張り合う結果を示している. しかし, どのデータセットについても最高の成績は出していない.
- E-MEM
  E-MEM は MEM の最小の長さが大きい時メモリが最も小さいものになった. しかし, どれも最速とはなっていない. E-MEM の論文では Gbps やそれ以上のゲノムについて扱いやすく, また索引の作成を含めた全体の流れで高速なアルゴリズムとされている. そのため, E-MEM との正当な比較は本章の最後で行う.
- essaMEM
  essaMEM は FBWTMEM と張り合う結果を出している. しかし, 同程度のメモリ使用量では FBWTMEM の方が勝る結果を出している. essaMEM 最速と FBWTMEM 最速の比較では, No 1 のデータセットでは, essaMEM の方が 1.15 倍高速, No 2 では 1.19 倍高速であった. 一方, No 3 では FBWTMEM の方が 1.02 倍, No 4 では 1.82 倍高速であった. 10 章で述べた通り, MEM の出現数が多い場合 essaMEM の方が有利になる傾向がある. しかし, 今回我々が用いたデータセットでは FBWTMEM の方が有利であることはいえる. また, essaMEM は skip パラメータを手動チューニングした結果である一方, FBWTMEM は skip パラメータをデフォルトのものを使っているため, 実際の使用では FBWTMEM の方が扱いやすいと

期待できる.

最後に E-MEM との比較をする. E-MEM は本章の冒頭で述べた通り索引の構築から検索すべて一貫した場合高速に動作するアルゴリズムである. FBWTMEM も E-MEM も索引の構築は単スレッドで行われ, MEM 検索が並列で行われる. 扱ったデータセットは表 9.1 の No 3. No 4. である. これらは Gpbs 程度の比較的大きいゲノムであるため, E-MEM の用途に適したものである. 表 11.2 は FBWTMEM と E-MEM の索引作成時間とメモリ使用量を表している. FBWTMEM については sparse パラメータを変化させたものを記している. E-MEM はデフォルトのものについて一つ, FBWTMEM の sparse パラメータの一番小さいものの列に示している. 表 11.3 は FBWTMEM と E-MEM の MEM 検索時間とメモリ使用量である. 表 11.2 の表の見方と同様であるが, スレッド数を 1, 16 で行ったものも示している. 図 11.9 は表 11.2 と表 11.3 の実行時間を組み合わせたものである. データセットとスレッド数の区分では, データセット No 3. でスレッド数 1 のものは, Test id 1 から 4 (FBWTMEM) と 17 (E-MEM), データセット No 4. でスレッド数 1 のものは, Test id 5 から 8 (FBWTMEM) と 18 (E-MEM), データセット No 3. でスレッド数 16 のものは, Test id 9 から 12 (FBWTMEM) と 19 (E-MEM), データセット No 4. でスレッド数 16 のものは, Test id 13 から 16 (FBWTMEM) と 20 (E-MEM) である. スレッド数が 1 の時は FBWTMEM と E-MEM どちらも同程度の実行時間であることがわかる. 具体的な数値としては, データセット No 3. のものでは 0.954 から 1.176 倍高速である. No 4. では 1.213 から 1.496 倍高速となった. しかし, スレッド数が 16 の時は, データセット No 3. で E-MEM の方が 3.540 倍から 4.514 倍, データセット No 4. で 1.862 倍から 1.834 倍高速となった. グラフから見て取れるように我々の手法は索引の構築に比重が置かれ, 検索時間が少ない. 一方 E-MEM はその逆で検索時間が多い. 並列度が高い検索時間に比重を置くことにより全体としては E-MEM の方が高速となった. このことから, 我々の手法は検索に多く時間がかかるとき, つまり比較対象のゲノムが多くあるとき有利になる. 例えば, データセット No 3. では 5 つくらいのゲノムをヒトゲノムと比較したいとき高速に動作することが言える.

表 11.2: FBWTMEM と E-MEM の検索索引作成時間とメモリサイズ．K 列は FBWTMEM の sparse パラメータ．E-MEM のデータは各データセットについて K の最も小さい行に示されている．

id	K	FBWTMEM		E-MEM	
		time [s]	size [MB]	time [s]	size [MB]
mm10	3	827.466	20143	10.091	2825
mm10	4	727.785	16487	-	-
mm10	8	625.862	11003	-	-
mm10	16	792.562	8392	-	-
hg19	3	938.663	23047	10.5048	3228
hg19	4	835.04	18868	-	-
hg19	8	699.846	12600	-	-
hg19	16	888.279	9597	-	-
panTro3	3	1106.42	23209	10.7031	3257
panTro3	4	922.352	19002	-	-
panTro3	8	757.151	12691	-	-
panTro3	16	919.616	9667	-	-

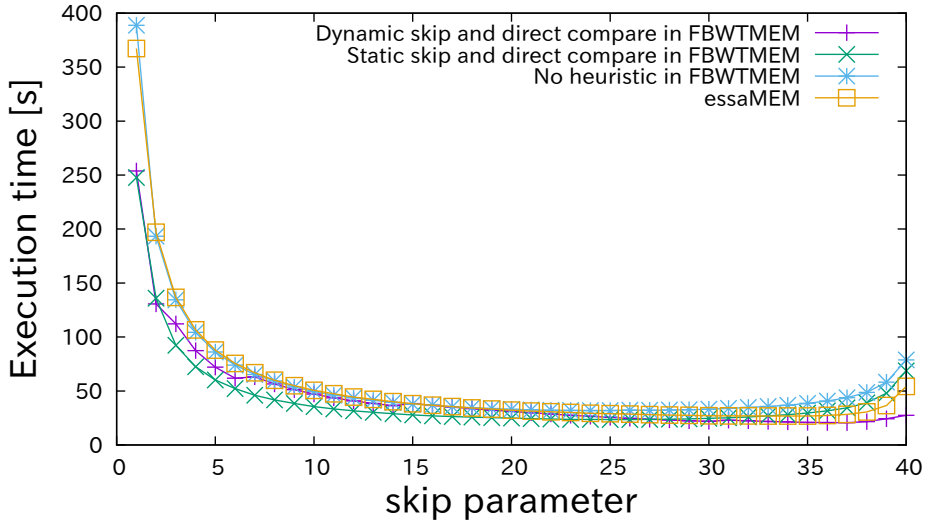


図 11.1: No. 1 のデータセットについて実行時間と skip パラメータの関係. essaMEM は child table を用いたもの. FBWTMEM はすべてのヒューリスティックスをなくしたもの, 直接比較するヒューリスティックスを加えたもの, さらに動的に skip パラメータを変更したものである. essaMEM と FBWTMEM のすべての試行の sparse パラメータは 1 である.

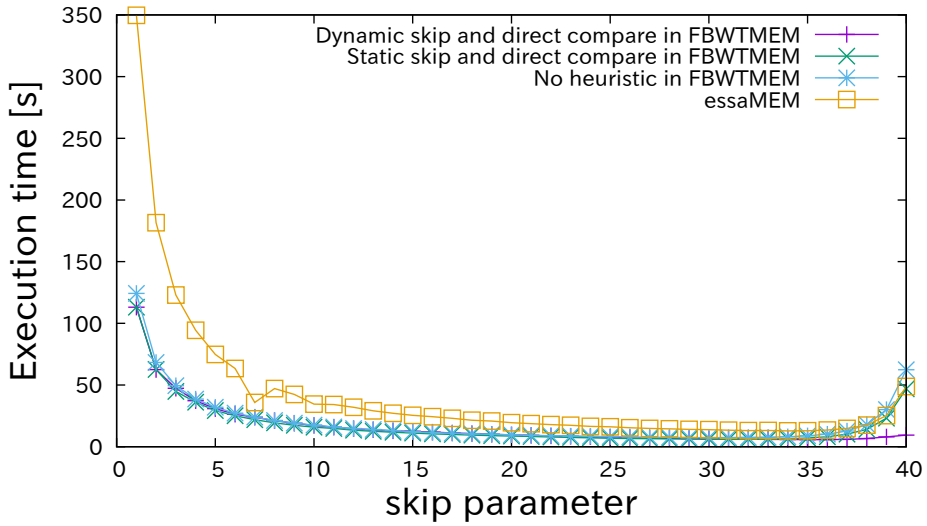


図 11.2: No. 2 のデータセットについて実行時間と skip パラメータの関係. essaMEM は child table を用いたもの. FBWTMEM はすべてのヒューリスティックスをなくしたもの, 直接比較するヒューリスティックスを加えたもの, さらに動的に skip パラメータを変更したものである. essaMEM と FBWTMEM のすべての試行の sparse パラメータは 1 である.

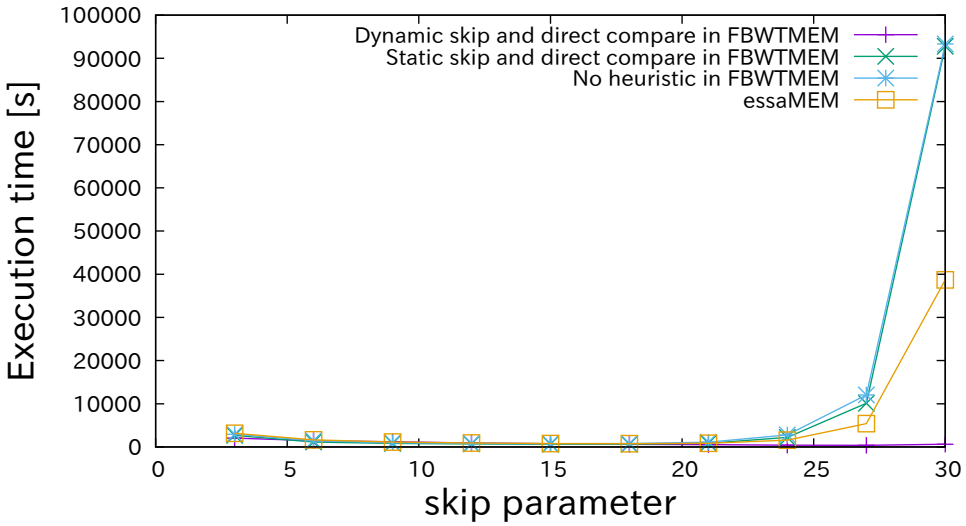


図 11.3: No. 3 のデータセットについて実行時間と skip パラメータの関係. essaMEM は child table を用いたもの. FBWTMEM はすべてのヒューリスティックスをなくしたもの, 直接比較するヒューリスティックスを加えたもの, さらに動的に skip パラメータを変更したものである. essaMEM と FBWTMEM のすべての試行の sparse パラメータは 1 である.

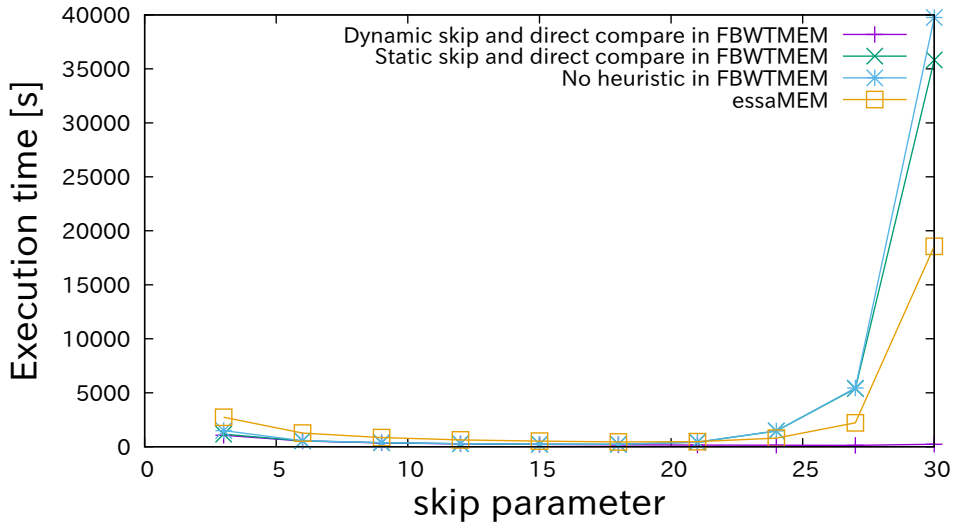


図 11.4: No. 4 のデータセットについて実行時間と skip パラメータの関係. essaMEM は child table を用いたもの. FBWTMEM はすべてのヒューリスティックスをなくしたもの, 直接比較するヒューリスティックスを加えたもの, さらに動的に skip パラメータを変更したものである. essaMEM と FBWTMEM のすべての試行の sparse パラメータは 1 である.

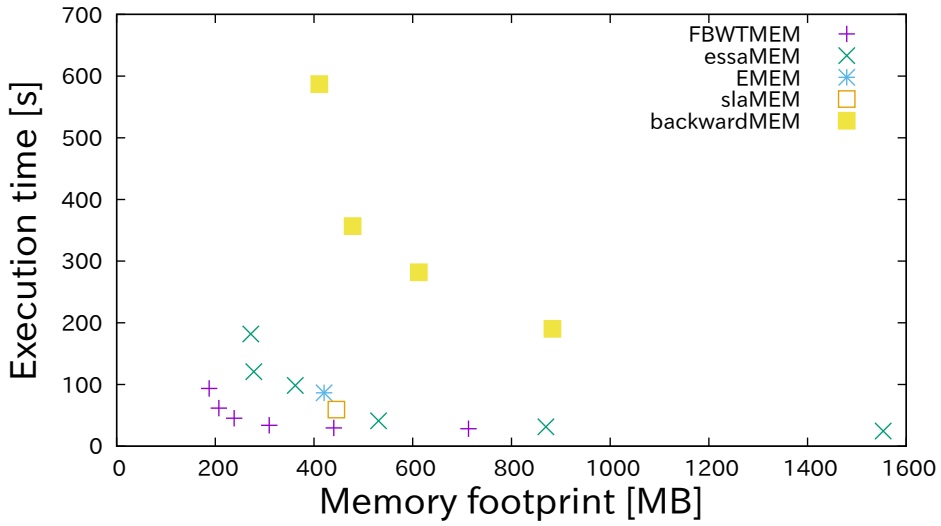


図 11.5: No. 1 のデータセットを用いた時の MEM 検索実行時間とメモリ使用量の比較

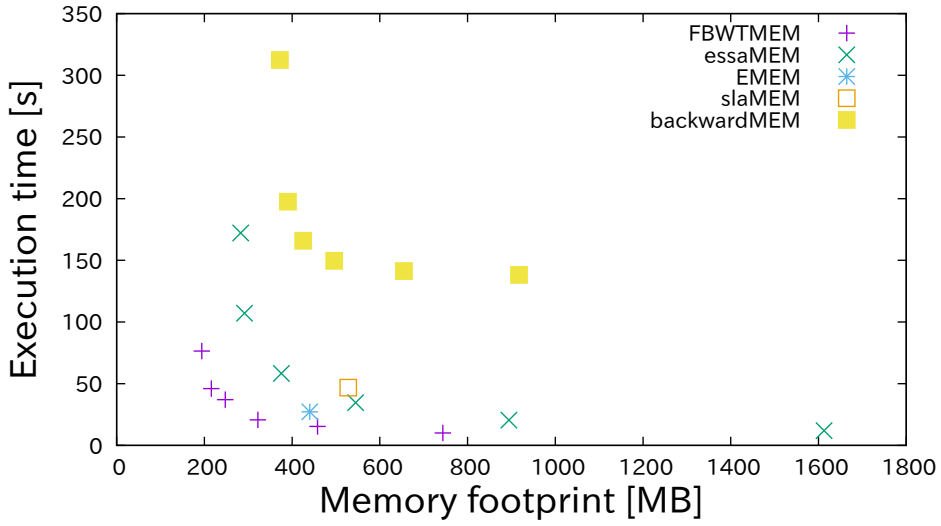


図 11.6: No. 2 のデータセットを用いた時の MEM 検索実行時間とメモリ使用量の比較

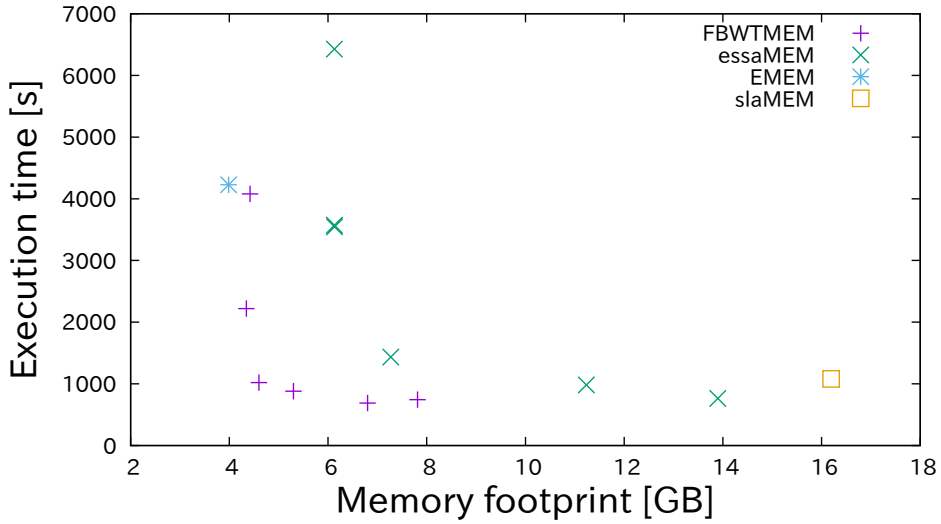


図 11.7: No. 3 のデータセットを用いた時の MEM 検索実行時間とメモリ使用量の比較

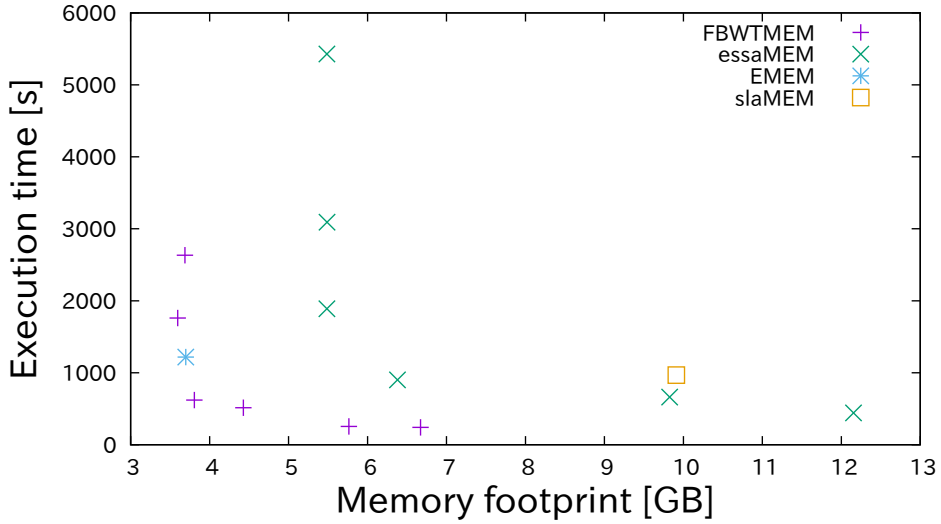


図 11.8: No. 4 のデータセットを用いた時の MEM 検索実行時間とメモリ使用量の比較

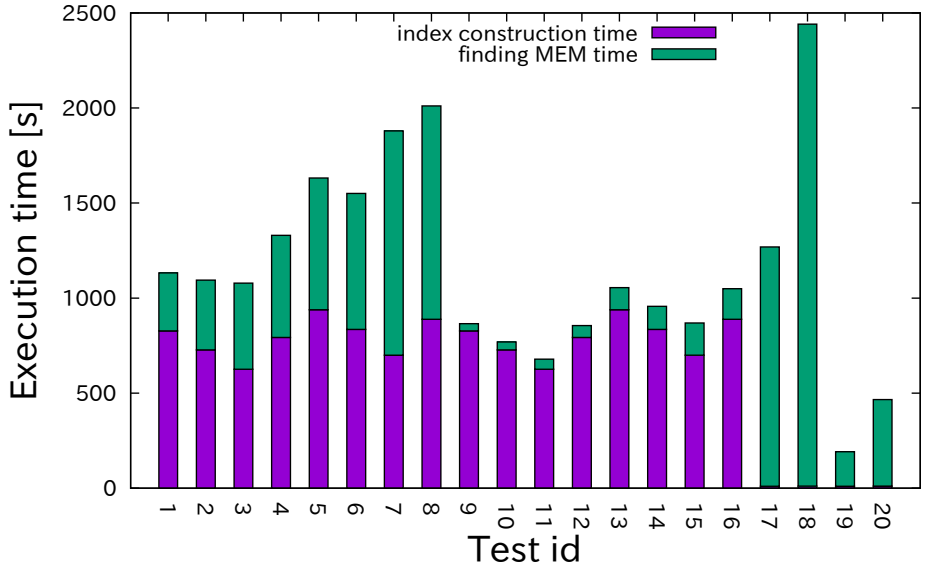


図 11.9: 検索索引作成から MEM 検索すべてを含んだ実行時間の比較. test id 列は表 11.3no test id に対応する.

表 11.3: FBWTMEM と E-MEM の MEM 検索時間とメモリサイズ. K 列は FBWTMEM の sparse パラメータ. E-MEM のデータは各データセットについて K の最も小さい行に示されている.

				FBWTMEM			E-MEM	
test id	id	thread	sparse	time [s]	size [MB]	test id	time [s]	size [MB]
1	3	1	3	304.787	6694	17	1259.279	3714
2	3	1	4	366.813	5787		-	-
3	3	1	8	453.051	4448		-	-
4	3	1	16	537.433	3828		-	-
5	4	1	3	692.73	10709	18	2430.6152	4100
6	4	1	4	715.243	9672		-	-
7	4	1	8	1179.38	8139		-	-
8	4	1	16	1122.73	7421		-	-
9	3	16	3	37.7707	6683	19	181.583	3715
10	3	16	4	41.4299	5778		-	-
11	3	16	8	52.6651	4440		-	-
12	3	16	16	63.0688	3820		-	-
13	4	16	3	116.034	10640	20	455.8382	4100
14	4	16	4	121.43	9600		-	-
15	4	16	8	168.832	8084		-	-
16	4	16	16	160.743	7430		-	-

第12章　結論

12.1　まとめ

本論文では, ゲノム比較においてよく使われる MEM 検索アルゴリズムの改良を行い, FBWT-MEM を提案し, 実装及び評価を行った. 我々は既存研究である essaMEM のアプローチを継承し, まず ESSA の代わりに FBWT を用いることで検索時のメモリ使用量の削減に成功した. さらに, essaMEM で用いられていた skip パラメータを動的にチューニングすることによりデフォルトのパラメータでゲノムの大小にかかわらずすべての既存手法を上回る結果を出した. ただし, 二つのみのゲノムを比較する場合, つまり索引の構築から MEM 検索全体の計算時間が重要な場合について E-MEM に劣ってしまうという欠点は存在する. しかし, 例えば Multiple Sequence Alignment のように, 多数のゲノムを比較し, 近縁種を見つけ出す方法に対しては有効なものであると考えられる.

12.2　今後の展望

今回の研究ではゲノム比較においてコアなアルゴリズムの高速化を行った. 対象としたのは検索時間の短縮であり索引構築の時間ではなく, そのため少数のゲノムを比較する場合は E-MEM に劣ってしまう. 実行時間の多くを占めるのは suffix array 構築であるが, suffix array 構築の高速化は困難な課題となっており, 非常に挑戦的なテーマの一つである. 一方, ゲノム比較に目を移すと, これからの技術の発展に伴い様々な種のゲノムデータが手に入っていくことになると期待される. そのためには高速な比較アプリケーションが不可欠となる. まとめでも述べたが, Multiple Sequence Alignment は多数のゲノムを比較するため, 我々の手法に好都合なものであると考えられる. この手法については別途調査が必要となるが, 非常に取り組みがいのある分野であると思う.

謝辞

初めに, 田浦先生にはとてもお世話になりました. 研究の方針等の相談から, 論文の校正, 発表まで根気強く指導いただきありがとうございました.

次に, IBM Research の井上さん, 卒論時からお世話になりましたが, 井上さんの博士修了後も面倒を見ていただき本当にありがとうございました. IBM でのインターンはとても刺激的で良い経験ができました.

研究室の先輩の岩崎さん, めりんぎさん, その他私の無為な雑談や企画に付き合ってくれたメンバーたちには感謝しきれません. また, IBM Research の小原さん, 伊藤愛さん, 上田高徳さん, 仲池さんもお昼の雑談や研究の相談等に乗っていただきとてもありがたかったです.

田浦研究室と IBM Research でサポートしていただいたすべての人にはとても感謝しております. ありがとうございました.

Publications

国際学会（査読あり）

[1] Masaru Ito, Hiroshi Inoue, Kenjiro Taura. Fragmented BWT : An Extended BWT for Full-text Indexing. *String Processing and Information Retrieval 2016*, Beppu, Japan, 2017/9.

国内学会（査読あり）

[2] Masaru Ito, Kenjiro Taura. Bzip2+ : FBWT を用いた高速な圧縮. *xSIG 2017*, Tokyo, 2017/4.

国内学会ポスター発表

[3] Masaru Ito, Kenjiro Taura. Bzip2+ : FBWT を用いた高速な圧縮. *xSIG 2017*, Tokyo, 2017/4.

Technical Paper

[4] FBWTMEM : computing maximal exact matches with FBWT. `http://domino.watson.ibm.com/library/CyberDig.nsf/1e4115aea78b6e7c85256b360066f0d4/eebd85612056be8e852581c300292b2e!OpenDocument\&Highlight=0,RT0981` *IBM Research Technical Paper*, 2017.

Bibliography

[1] Apache sercer log https://www.sec.gov/dera/data/edgar-log-file-data-set.html

[2] human genome http://hgdownload.cse.ucsc.edu/goldenPath/hg38/bigZips

[3] Linux source, https://www.kernel.org

[4] Wikipedia dump file https://dumps.wikimedia.org/enwiki/latest/enwiki-latest-pages-meta-current.xml.bz2

[5] Burrows, M., Wheeler, D.: A block-sorting lossless data compression algorithm. Algorithm, Data Compression (124), 18 (1994)

[6] Callaway, E.: Geneticists tap human knockouts. nature news (2014)

[7] Chain, P.S.G., Carniel, E., Larimer, F.W., Lamerdin, J., Stoutland, P.O., Regala, W.M., Georgescu, A.M., Vergez, L.M., Land, M.L., Motin, V.L., Brubaker, R.R., Fowler, J., Hinnebusch, J., Marceau, M., Medigue, C., Simonet, M., Chenal-Francisque, V., Souza, B., Dacheux, D., Elliott, J.M., Derbise, A., Hauser, L.J., Garcia, E.: Insights into the evolution of yersinia pestis through whole-genome comparison with yersinia pseudotuberculosis. Proceedings of the National Academy of Sciences 101 (2004)

[8] Daniels, M.A., Kan, C., Willmes, D.M., Ismail, K., Pistrosch, F., Hopkins, D., Mingrone, G., Bornstein, S.R., Birkenfeld, A.L.: Pharmacogenomics in type 2 diabetes: oral antidiabetic drugs. The Pharmacogenomics Journal pp. 399–410 (2016)

[9] Deutsch, L.P.: Gzip file format specification vertion 4.3 (1996), https://tools.ietf.org/html/rfc1952

[10] Fernandes, F., Freitas, A.T.: slamem: efficient retrieval of maximal exact matches using a sampled lcp array. Bioinformatics 30, 464–471 (2014), http://dx.doi.org/10.1093/bioinformatics/btt706

[11] Hall, S.S.: Genetics: A gene of rare effect. nature news (2013)

[12] Ito, M., Inoue, H., Taura, K.: Fragmented bwt: An extended bwt for full-text indexing. String Processing and Information Retrieval pp. 97–109 (2016)

[13] Khan, Z., Bloom, J.S., Kruglyak, L., Singh, M.: A practical algorithm for finding maximal exact matches in large sequence datasets using sparse suffix arrays. Bioinformatics 25, 1609–1616 (2009), https://doi.org/10.1093/bioinformatics/btp275

[14] Khiste, N., Ilie, L.: E-MEM: efficient computation of maximal exact matches for very large genomes. Bioinformatics 32(4), 509–514 (2015), http://dx.doi.org/10.1093/bioinformatics/btu687

[15] Knuth, D.E., Jr., J.H.M., , Pratt, V.R.: Fast pattern matching in strings. SIAM Journal on Computing pp. 323–350 (1974)

[16] Kurtz, S., Phillippy, A., Delcher, A.L., Smoot, M., Shumway, M., Antonescu, C., Salzberg, S.L.: Versatile and open software for comparing large genomes. Genome Biology 5(R12) (2004), https://doi.org/10.1186/gb-2004-5-2-r12

[17] M., B., D., W.: A block-sorting lossless data compression algorithm. Algorithm, Data Compression (124), 18 (1994)

[18] Manzini, G.: An analysis of burrows-wheeler transform. Journal of ACM 48, 407–430 (2001)

[19] Misra, N., Panda, P.K., Parida, B.K.: Agrigenomics for microalgal biofuel production: An overview of various bioinformatics resources and recent studies to link omics to bioenergy and bioeconomy. In: OMICS: A Journal of Integrative Biology. pp. 537–549 (2013)

[20] Oetting, W.S., Beroud, C., Brenner, S.E., Greenblatt, M., Karchin, R., Mooney, S.D., Sunyaev, S.: Non-cfoding variation: The 2016 annual scientific meeting of the human genome variation society. Human Mutation 38, 460–463 (2017)

[21] Ohlebusch, E., Gog, S., Kügel, A.: Computing matching statistics and maximal exact matches on compressed full-text indexe. Proceedings of the 17th Annual Symposium on String Processing and Information Retrieval. 21, 347–358 (2010)

[22] Okada, Y., Kubo, M., Ohmiya, H., Takahashi, A., Kumasaka, N., Hosono, N., Maeda, S., Wen, W., Dorajoo, R., Go, M.J., Zheng, W., Kato, N., Wu, J.Y., Lu, Q., consortium, G., Tsunoda, T., Yamamoto, K., Nakamura, Y., Kamatani, N., Tanaka, T.: Common variants at cdkal1 and klf9 are associated with body mass index in east asian populations. Nature genetics pp. 302–306 (2012)

[23] P., F., G., M.: Indexing Compressed Text. Journal of the ACM 52(4), 552–581 (2000)

[24] Rizzoa, L., Manaiab, C., Merlinc, C., Schwartzd, T., Dagote, C., Ployf, M., Michaelg, I., Fatta-Kassinosg, D.: Urban wastewater treatment plants as hotspots for antibiotic resistant bacteria and genes spread into the environment: A review. In: Science of The total Environment (2013)

[25] Sadakane, K.: New text indexing functionalities of the compressed suffix arrays. Journal of Algorithms 48, 294–313 (2004)

[26] Shipman, S.L., Nivala, J., Macklis, J.D., Church, G.M.: Crisprcas encoding of a digital movie into the genomes of a population of living bacteria. nature (2017)

[27] Smith, T.F., Waterman, M.S.: Identification of Common Molecular Subsequence. Journal of Molecular Biology pp. 195–197 (1981)

[28] U., M., G., M.: Suffix String Arrays : A New Searches Method for On-Line. Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms pp. 319–327 (1990)

[29] Vyverman, M., Baets, B.D., Fack, V., Dawyndt, P.: essaMEM: finding maximal exact matches using enhanced sparse suffix arrays. Bioinformatics 29(6), 802–804 (2013), http://dx.doi.org/10.1093/bioinformatics/btt042

[30] Wen, W., Cho, Y.S., Zheng, W., Dorajoo, R., Kato, N., Qi, L., Chen, C.H., Delahanty, R.J., Okada, Y., Tabara, Y., Gu, D., Zhu, D., Haiman, C.A., Mo, Z., Gao, Y.T., Saw, S.M., Go, M.J., Takeuchi, F., Chang, L.C., Kokubo, Y., Liang, J., Hao, M., Marchand, L.L., Zhang, Y., Hu, Y., et al: Meta-analysis identifies common variants associated with body mass index in east asians. Nature genetics pp. 307–311 (2012)

付 録A　その他データセットと数値

表 A.1: 各データセットの属性. len は長さを示す

id	reference	size[bp]	query	size[bp]	MEM len	# of MEMs	Avg. len of MEMs
5	fumigatus	29M	nidulans	30M	20	330150	25.9432
6	sapiencs21	46M	musculus16	95M	50	586296	55.0137
7	musculus16	95M	sapiencs21	46M	50	586296	55.0137
8	melanogaster	140M	sechellia	163M	50	2619476	85.9191

表 A.2: MEM 検索における各種アルゴリズムの実行時間．もし sparse パラメータが存在しない場合結果は最も小さい sparse パラメータの行に示されている．id 列は表 9.1 に対応している．s 列は sparse パラメータ, m はメモリ使用量, t は実行時間.

id	s	FBWTMEM		backwardMEM		essaMEM		slaMEM		EMEM	
		m[MB]	t[s]	m[MB]	t[s]	m[MB]	t[s]	m[MB]	t[s]	m[MB]	t[s]
1	1	713	28.408	883	190.377	1553	24.626	445	59.2582	420	86.483
1	2	440	29.616	612	282.054	869	31.529	-	-	-	-
1	4	309	33.876	478	356.74	530	41.064	-	-	-	-
1	8	238	45.283	410	586.937	362	98.566	-	-	-	-
1	16	207	61.844	378	1146.64	277	120.897	-	-	-	-
1	32	187	93.609	361	2347.93	271	181.884	-	-	-	-
2	1	743	10.017	917	138.215	1606	11.427	528	46.8108	440	27.241
2	2	458	15.406	654	141.288	887	21.333	-	-	-	-
2	4	322	20.665	496	149.576	536	36.402	-	-	-	-
2	8	247	37.115	425	165.848	365	64.360	-	-	-	-
2	16	215	46.036	390	197.498	282	109.382	-	-	-	-
2	32	194	76.460	371	312.328	283	173.165	-	-	-	-
3	3	7812	743.703	-	-	13899	763.369	16192	1080.46	3986	4227.33
3	4	6801	688.238	-	-	11235	983.305	-	-	-	-
3	8	5300	880.633	-	-	7272	1433.1	-	-	-	-
3	16	4598	1019.19	-	-	6130	3545.58	-	-	-	-
3	32	4346	2220.56	-	-	6130	3571.39	-	-	-	-
3	64	4420	4078.09	-	-	6130	6425.84	-	-	-	-
4	3	6671	241.755	-	-	12154	442.055	9911	966.823	3697	1218.62
4	4	5763	255.093	-	-	9826	661.148	-	-	-	-
4	8	4427	513.935	-	-	6379	901.597	-	-	-	-
4	16	3807	619.401	-	-	5485	1888.77	-	-	-	-
4	32	3595	1759.55	-	-	5486	3092.43	-	-	-	-
4	64	3687	2632.85	-	-	5486	5428.21	-	-	-	-
5	1	160	3.257	185	21.553	311	5.130	103	8.504	1737	9.666
5	2	100	4.923	129	24.310	182	10.213	-	-	-	-
5	4	74	6.057	101	21.979	117	17.688	-	-	-	-
5	8	56	11.524	85	23.046	84	27.093	-	-	-	-
6	1	359	3.918	415	69.977	718	6.822	242	26.4	147	13.295
6	2	264	6.008	324	72.036	449	7.835	-	-	-	-
6	4	223	7.696	277	67.324	316	16.419	-	-	-	-
6	8	198	12.299	256	68.525	251	25.865	-	-	-	-
6	16	189	22.035	243	71.618	218	30.637	-	-	-	-
6	32	179	41.696	239	77.325	202	52.625	-	-	-	-
7	1	561	2.836	642	33.614	1101	3.038	355	11.8836	282	10.833
7	2	366	3.815	449	33.134	635	4.599	-	-	-	-
7	4	273	6.623	352	34.105	402	7.0781	-	-	-	-
7	8	220	9.066	305	35.755	286	16.381	-	-	-	-
7	16	202	16.423	280	44.479	228	24.982	-	-	-	-
7	32	186	24.447	269	48.610	199	37.508	-	-	-	-
8	1	737	15.104	918	154.758	1604	13.110	521	44.3435	440	27.240
8	2	452	19.629	637	148.15	886	20.873	-	-	-	-
8	4	316	26.037	498	177.83	536	33.446	-	-	-	-
8	8	241	37.735	426	248.145	366	56.526	-	-	-	-
8	16	211	48.329	392	369.858	282	96.519	-	-	-	-
8	32	190	77.709	374	673.786	282	164.88	-	-	-	-

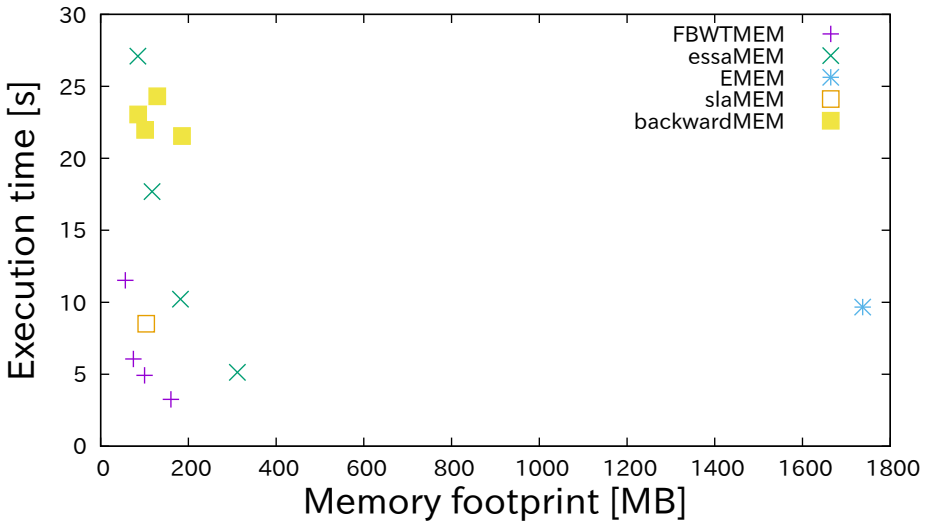


図 A.1: No. 5 のデータセットを用いた時の MEM 検索実行時間とメモリ使用量の比較

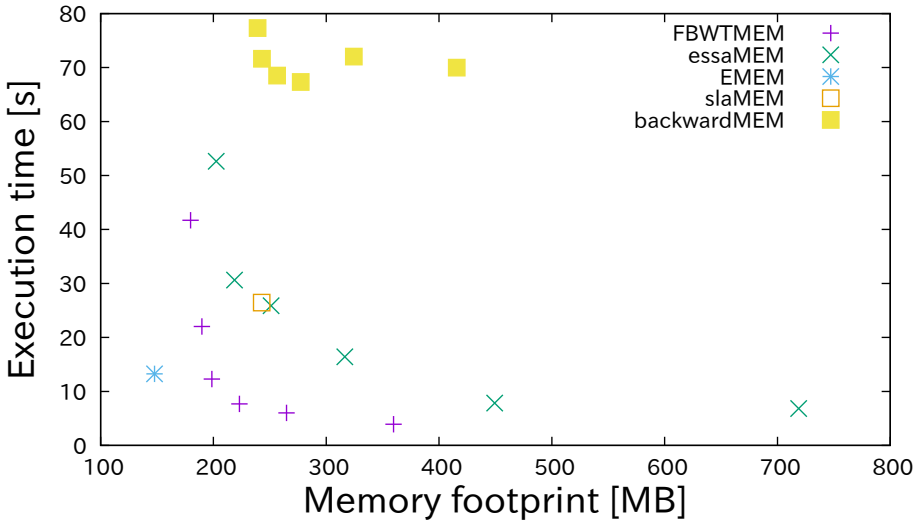


図 A.2: No. 6 のデータセットを用いた時の MEM 検索実行時間とメモリ使用量の比較

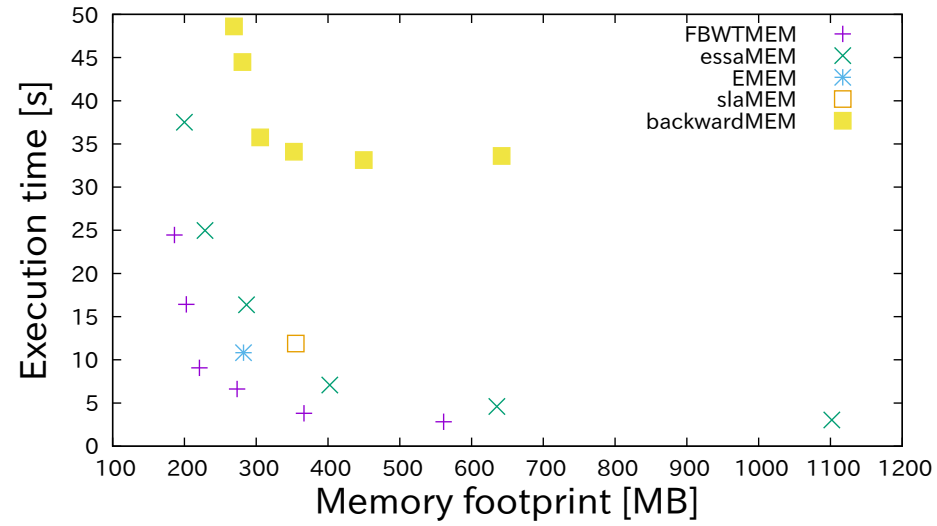


図 A.3: No. 7 のデータセットを用いた時の MEM 検索実行時間とメモリ使用量の比較

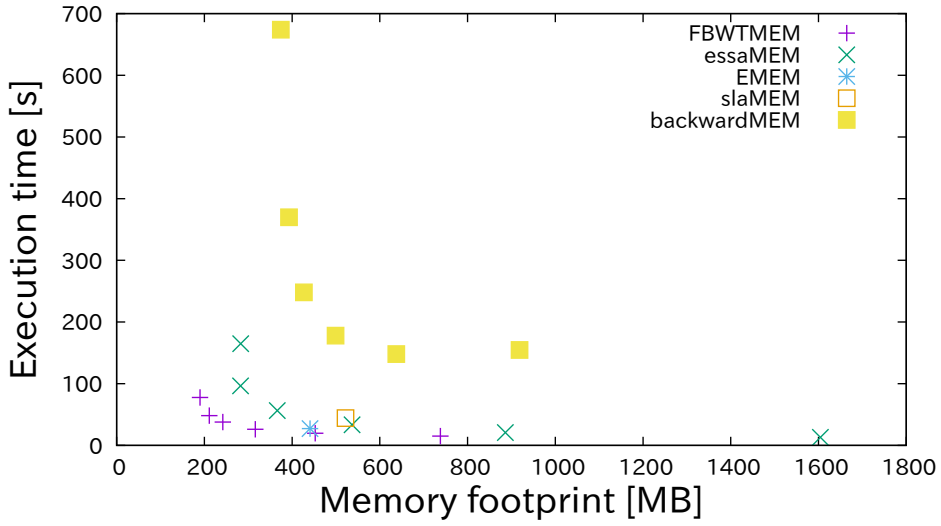


図 A.4: No. 8 のデータセットを用いた時の MEM 検索実行時間とメモリ使用量の比較

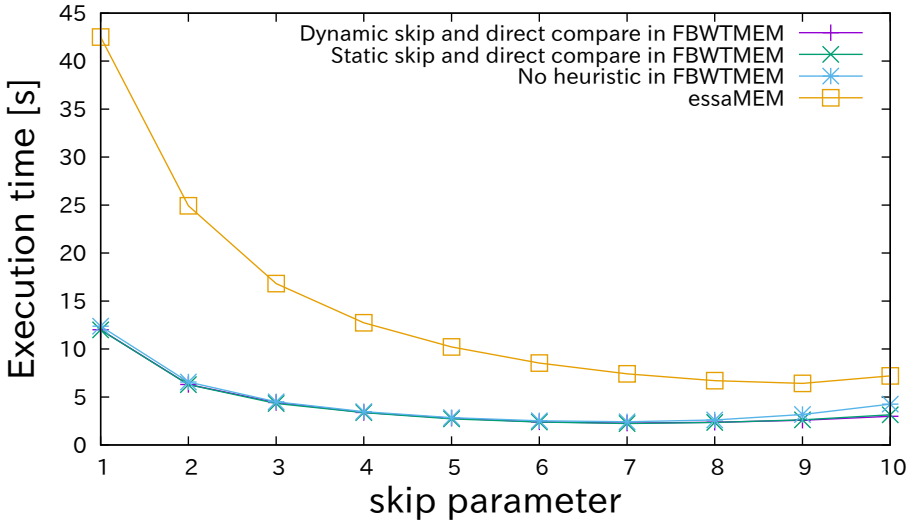


図 A.5: No. 5 のデータセットについて実行時間と skip パラメータの関係. essaMEM は child table を用いたもの. FBWTMEM はすべてのヒューリスティックスをなくしたもの, 直接比較するヒューリスティックスを加えたもの, さらに動的に skip パラメータを変更したものである. essaMEM と FBWTMEM のすべての試行の sparse パラメータは 1 である.

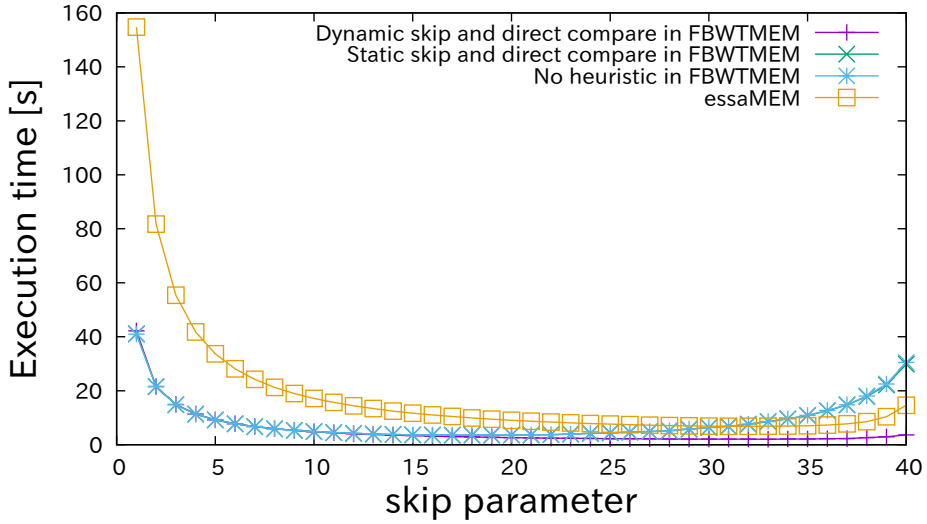


図 A.6: No. 6 のデータセットについて実行時間と skip パラメータの関係. essaMEM は child table を用いたもの. FBWTMEM はすべてのヒューリスティックスをなくしたもの, 直接比較するヒューリスティックスを加えたもの, さらに動的に skip パラメータを変更したものである. essaMEM と FBWTMEM のすべての試行の sparse パラメータは 1 である.

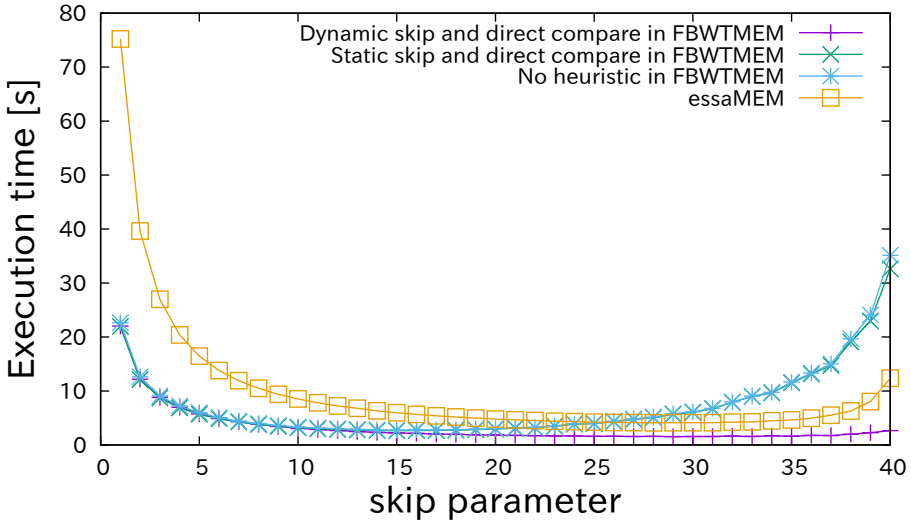


図 A.7: No. 7 のデータセットについて実行時間と skip パラメータの関係. essaMEM は child table を用いたもの. FBWTMEM はすべてのヒューリスティックスをなくしたもの, 直接比較するヒューリスティックスを加えたもの, さらに動的に skip パラメータを変更したものである. essaMEM と FBWTMEM のすべての試行の sparse パラメータは 1 である.

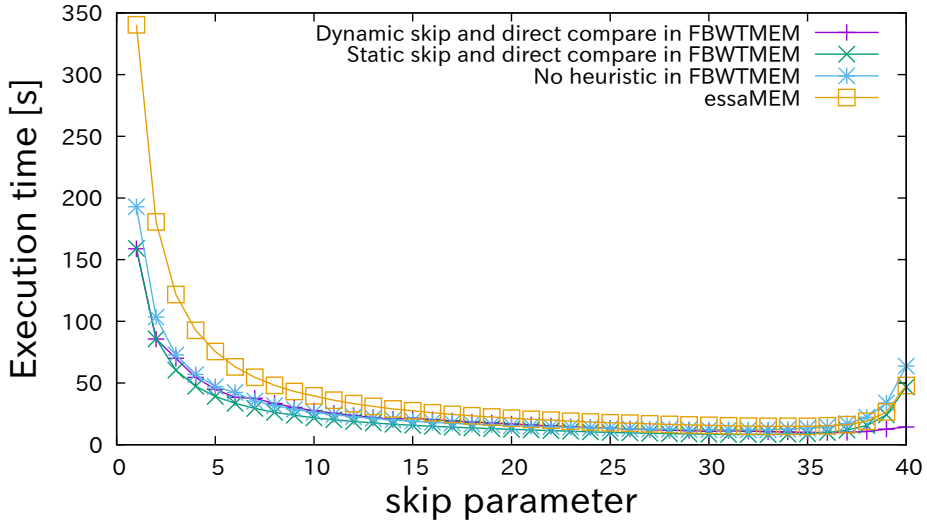


図 A.8: No. 8 のデータセットについて実行時間と skip パラメータの関係. essaMEM は child table を用いたもの. FBWTMEM はすべてのヒューリスティックスをなくしたもの, 直接比較するヒューリスティックスを加えたもの, さらに動的に skip パラメータを変更したものである. essaMEM と FBWTMEM のすべての試行の sparse パラメータは 1 である.