October 24, 2017
RT0981
Computer Science; Life Sciences 18 pages

Research Report

FBWTMEM : computing maximal exact matches with FBWT

Masaru Ito, Hiroshi Inoue, Megumi Ito and Moriyoshi Ohara

IBM Research - Tokyo
IBM Japan, Ltd.
19-21, Nihonbashi Hakozaki-cho
Chuo-ku, Tokyo 103-8510 Japan

IBM Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Abstract - Finding maximal exact matches (MEMs) is an important building block of many workloads including genome alignment. Hence its acceleration has a great impact on genome alignment and genome analysis workloads. In this paper, we developed a new algorithm, FBWTMEM, for this task, and experimentally showed that our algorithm outperforms previous ones. We achieved 1) a smaller memory footprint by using a recently-proposed data structure called FBWT and 2) a faster execution time by adaptively tuning the skip parameter.

1 Introduction

Maximal exact matches (MEMs) are exact matches between two sequences that do not match when lengthened from either the left or right side. MEM computation is an important task in sequence alignment and genome comparison.

Recent MEM-finding studies use index methods. The merit of this approach is the reusability of an index. However, the runtime memory consumption of the index is large, so recent studies have been trying to reduce it. Early studies based on index use suffix trees (Kurtz *et al.* (2004)) or enhanced suffix arrays (ESAs) (Abouelhoda *et al.* (2004)), but these application uses vast amounts of memory.

Sparse suffix arrays were used by Khan *et al.* (2009) to develop their algorithm, *sparseMEM*. A sparse suffix array consists of every K-th suffix of sequence. Their algorithm achieved a smaller memory footprint and shorter execution time than previous works. After *sparseMEM*, additional algorithms were proposed such as *backwardMEM* (Ohlebusch *et al.* (2010)) based on compressed suffix arrays, *essaMEM* (Vyverman *et al.* (2013)), based on enhanced sparse suffix arrays (ESSAs), *slaMEM* (Fernandes and Freitas (2014)), based on SSILCP, and *EMEM* (Khiste and Ilie (2015)), based on hashes. EMEM is the most efficient when the minimal MEM length is long, and for relatively short MEMs, the most efficient algorithm is essaMEM. essaMEM improved upon sparseMEM's design using ESSA, introduced skip parameters, and has a great impact on execution time. However, we found that the default skip parameter proposed by them is insufficient to achieve optimal performance, so we developed an improved optimization method.

In regard to memory consumption, *FM-index* (Ferragina and Manzini (2005)) can dramatically reduce it if the characters that appear in reference are similar to a genome. FM-index uses *Burrows-Wheeler transform* (Burrows and Wheeler (1994)), and we recently proposed a new index structure called *fragmented Burrows-Wheeler transform* (FBWT) (Ito *et al.* (2016)), which is an extension of BWT that consumes the same memory footprint.

In this article, we further optimize the sparse suffix array approach. We demonstrate the possibility of using FBWT to keep competitive performance with less memory, and report on our findings of further optimization of the skip parameter. These findings led our approach to alternative solutions to previous works for short and long MEMs. The experimental result shows our approach is more efficient than the other approaches in the view of both memory consumption and execution time.

2 Methods

We based our algorithm (FBWTMEM) on essaMEM with the intention of improving its performance. The developers of essaMEM implemented two approaches utilizing child tables and suffix-link simulations, respectively. As the former was considered to be the most effective approach, we decided to improve upon that. This section will briefly present our improvements, and more detailed information is outlined in the Supplementary Material.

We briefly describe the common approaches of essaMEM and FBWTMEM. An index is created, then MEMs are found by using the following two steps. First, some query positions for some length l are matched to find candidates. l can also controls the number of positions of the query, which starts the procedure to find MEMs. A larger l equals more positions, which leads to a slower execution. MEMs are finally extracted from the candidates. The second step is proportional to occurrence. The l parameter controls the trade off between the second step, the number of procedures, and the first step. When l is small, the number of positions to start procedures is small and leads to faster executions, but the second step becomes large because the number of candidates also increases. On the other hand, if l is large, the second step can be small, but the number of positions increases and leads to slower executions.

We describe two major improvements in our algorithm. One is its data structure and the other is the dynamic change of the skip parameter introduced in essaMEM. The first improvement affects the reduction of the index size. essaMEM uses reference genomes (n bytes) and ESSA, which consists of an LCP array ($m \cdot n/k$ bytes; m is usually between 1 and 2), sparse suffix array ($4n/k$ bytes), and child array ($4n/k$ bytes), where k is a sparse parameter. In total, essaMEM uses $n + (m \cdot n + 8n)/k$ bytes. On the other hand, the index of FBWTMEM is based on FBWT, which is an extension of BWT, an algorithm for exact matching used in FM-index. The FBWT-based index consists of an occurrence table ($occ_{l/k}(c, i)$), c table ($C_{l/k}(c)$), and sparse suffix array. The occurrence table returns the number of occurrences of c in the prefix of lth FBWT, whose length is $i - 1$. The c table returns the number of characters, which is smaller than c in lth FBWT. In our implementation, FBWTMEM uses $4n/k$ bytes for the sparse suffix array, $n/2$ bytes for the reference genome, and $3n/8 + 4 \cdot 4 \cdot n/128$ bytes for the occurrence table and c table ($o(n)$). We encode the reference genome to 4-bit characters. The occurrence table consists of a sampled occurrence table in every 128 elements and 3-bit encoded nucleotides. When we access the occurrence table, we sum up the nucleotides and add the sampled occurrence elements. In total, FBWTMEM uses $n + 4n/k + o(n)$ bytes. As a result, our method uses a smaller amount of memory. essaMEM also uses a hash table to calculate the suffix array interval of a specified prefix length of a query. Its key is a 2-bit encoded sequence. In the case of FBWTMEM, we have to create hash table for each FBWT sequence, which becomes k times larger. To avoid creating a table larger than that of essaMEM, we shrink the length of the prefix used for the key in our experiment. The performance is competitive to essaMEM using this method. For more details, see section 4 in the Supplementary Material.

The second improvement is the dynamic change of the skip parameter. essaMEM uses a static skip parameter, which means the length of exact matching in the first step is static. This approach sometimes affects performance because the occurrence of candidates depends on data. When there are many occurrences, the second step requires large execution times, and it is not good for either essaMEM or our method. A number of results related to this are in section 4 in the Supplementary Material. To avoid this phenomenon, we perform further exact matches in the first step to reduce the candidates. When we increase the exact matching length, additional MEM-finding procedures must be run, but this improves the performance in our experiments and the skip parameter tuning cost. For more details, see section 4 in the Supplementary Material.

3 Experimental Results

We measured the elapsed times of the MEM-finding functions by the *gettimeofday* function, and the memory size by read /proc/{$pid}/smaps. All tests were run on a Red Hat Enterprise Linux Server release 6.6 machine featuring an Intel Xeon CPU E5-2690 clocked at 2.90GHz with 100GB of RAM. We evaluated the performance of FBWTMEM against existing algorithms such as essaMEM, backwardMEM, slaMEM, and EMEM. We

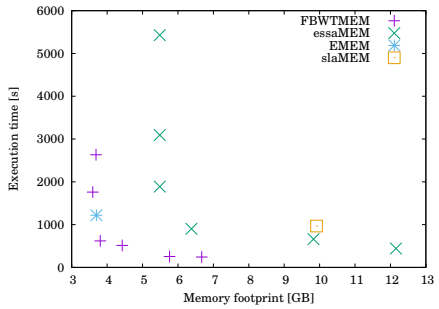

Fig. 1. Plot comparing time and memory footprint computing MEMs between Mus musculus (mm10) and Human genome (hg19). Minimal MEM length is 100

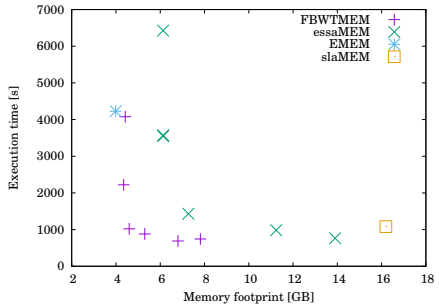

Fig. 2. Plot comparing time and memory footprint computing MEMs between Human genome (hg10) and chimpanzee (panTro3). Minimal MEM length is 100

used 7 datasets that included several Mbp and Gbp genome sequences, but we did not measure Gbp datasets for backwardMEM because we expected it would be extremely slow. We used the largest possible skip parameter value s for FBWTMEM. Our method demonstrated great performance for all datasets. The results of the mm10 versus hg19 and mm10 versus panTro3 datasets will be shown in this article. The other results can be found in the Supplementary Material. For the presented datasets, we set the hash key length to 10 and k set in the range (3, 4, 8, 16, 32, 64). The threshold switching for direct comparison with the reference in the second step of the algorithm was 10, and the threshold to decrease the s parameter in the first step of the algorithm was 10. For essaMEM, we used the optimal value s, the hash length was 10, and the K parameter is set to the same as above. For slaMEM and EMEM, we used -n option, which means to deal with nucleotides only. FBWTMEM has a better or competitive performance against the other algorithms for all data sets. In mm10 versus hg19 genome, FBWTMEM has the best performance. Our method is twice as fast as EMEM and three times as fast as essaMEM in comparable memory. The fastest sample in FBWTMEM was 1.8 times faster than that in essaMEM. When k is 16, 32 or 64, FBWTMEM uses almost same memory. This is because the sparse parameter does not influence the total memory size due to a much smaller suffix array than the occurrence table and reference. In hg19 versus panTro3 genome, FBWTMEM has the better performance. Our method is four times faster than EMEM and 1.9 times faster than essaMEM. However, the fastest sample in FBWTMEM was only 1.03 times faster than that in essaMEM.

Table 1 shows each dataset's properties. We were concerned that our algorithm would be slower when there are many MEMs because the second step of algorithm is not efficient compared with essaMEM. This discussion is in section 2.2 in the Supplementary Material. In hg19 versus panTro3, there are 240 times more occurrences than hg19 versus mm10, and the relative performance of FBWTMEM becomes lower than that of essaMEM as we described above. This fact indicates that FBWTMEM is inadequate with large occurrences of MEMs, but our method is useful for real world datasets seeing other datasets. A similar phenomenon can be found in the *Drosophila* datasets shown in the Supplementary Material.

Table 1. Datasets property. seq1 is indexed sequence and second column is its size, seq2 is query sequence and forth column is its size, MEM len means minimal length of MEMs

seq1	size[bp]	seq2	size[bp]	MEM len	# of MEMs
hg19	3.1G	panTro3	3.2G	100	132M
mm10	2.7G	hg19	3.1G	100	554k

4 Conclusion

FBWTMEM is more efficient than previous works for real datasets. It achieves a smaller memory footprint than that of essaMEM, is competitive to EMEM, and performs faster than previous works using default parameters. There is a concern about a performance decrease with many MEM occurrences, but our tests show that FBWTMEM is useful. Our algorithm performs well even when the genome is large, so it is valuable for future large-genome analysis.

References

Abouelhoda, M. I., Kurtz, S., and Ohlebusch, E. (2004). Replacing suffix trees with enhanced suffix arrays. volume 2, pages 53–86.

Burrows, M. and Wheeler, D. (1994). A block sorting lossless data compression algorithm. *SRC Research Report*, (124).

Fernandes, F. and Freitas, A. T. (2014). slamem: efficient retrieval of maximal exact matches using a sampled lcp array. *Bioinformatics*, **30**, 464–471.

Ferragina, P. and Manzini, G. (2005). Indexing Compressed Text. *Journal of ACM*, **52**(4), 552–591.

Ito, M., Inoue, H., and Taura, K. (2016). Fragmented BWT: An Extended BWT for Full-Text Indexing. *Proceedings of the 23rd Annual Symposium on String Processing and Information Retrieval.*, pages 97–109.

Khan, Z., Bloom, J. S., Kruglyak, L., and Singh, M. (2009). A practical algorithm for finding maximal exact matches in large sequence datasets using sparse suffix arrays. *Bioinformatics*, **25**, 1609–1616.

Khiste, N. and Ilie, L. (2015). E-MEM: efficient computation of maximal exact matches for very large genomes. *Bioinformatics*, **32**(4), 509–514.

Kurtz, S., Phillippy, A., Delcher, A. L., Smoot, M., Shumway, M., Antonescu, C., and Salzberg, S. L. (2004). Versatile and open software for comparing large genomes. *Genome Biology*, **5**(R12).

Ohlebusch, E., Gog, S., and Kügel, A. (2010). 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.

Vyverman, M., Baets, B. D., Fack, V., and Dawyndt, P. (2013). essaMEM: finding maximal exact matches using enhanced sparse suffix arrays. *Bioinformatics*, **29**(6), 802–804.

Supplementary Material
"FBWTMEM : computing maximal exact matches with FBWT"

Masaru Ito *, Hiroshi Inoue, Megumi Ito and Moriyoshi Ohara

1 Preliminary

We first annotate some variables. R is the reference sequence, $R_{k\$}$ is a sequence with an additional character ,\$, padded to R until the length becomes a multiple of k. $\mid R_{k\$} \mid = n$ is the length of the $R_{k\$}$, $R_{k\$}[i:j]$ is subsequence of $R_{k\$}$ started from i-th position and end at j-th position of $R_{k\$}$, $R_{k\$}[i]$ is a i-th character of $R_{k\$}$, Q is the query, and $\mid Q \mid = m$ is the length of the Q. A *maximal exact match* (MEM) between R and Q are common subsequences that cannot be extended from either the left or right side. The MEM-finding task retrieves the positions of MEMs whose lengths are over L.

1.1 Fragmented Burrows-Wheeler transform

A *fragmented suffix array* (FSA) is a k sized set of numeric sequences. The k parameter is given by the user. l-th sequence of FSA ($FSA_{l/k}$) is a permutation of $\{i \mid i = k \cdot q + l, q \in \mathbb{N}, 0 \leq i < n\}$, and has the following inequality.

$$R_{k\$}[FSA_{l/k}[i] : n-1] <_{lex} R_{k\$}[FSA_{l/k}[i+1] : n-1] \quad (0 \leq i < n/k - 1) \tag{1}$$

where $a <_{lex} b$ means b is lexicographically greater than a. We assume that when $l < 0$ or $l >= k$, the l-th sequence is the $l \mod k$ sequence.

A *fragmented Burrows-Wheeler transform* (FBWT) is also a k sized set of sequences defined by the following equation.

$$FBWT_{l/k}[i] = \begin{cases} R_{k\$}[FSA_{l/k}[i] - 1] & \big(FSA_{l/k}[i] > 0\big) \\ R_{k\$}[n-1] & \big(FSA_{l/k}[i] = 0\big) \end{cases} \tag{2}$$

$FBWT_{l/k}$ denotes l-th sequence of FBWT. Figure 1 shows an example of FBWT for "bananabanana" and the size of FBWT is 3. \$ is the smallest character. The left column shows the FSA, and the right column shows the FBWT. The strings in center column are suffixes. Because \$ is the smallest character, any character after it does not influence the lexicographical order.

The task of searching a query in a text can be described as finding intervals in FSA. Thanks to lexicographical sorting, all suffixes whose prefix matches the query are contiguously located in each FSA. The goal of searching is to find the intervals whose elements point to suffixes whose prefix is equal to the query, and get the values of the elements.

We demonstrated that it is possible to find the intervals using FBWT (Ito *et al.* (2016)), and this process is called *backward search*. The following algorithm 1 shows the backward search procedure.

Listing 1: Given character c and interval l-$[i..j]$ corresponding to w in l-th sequence of FBWT, $backwardSearch(c, l\text{-}[i..j])$ computes interval of cw in $l-1$-th sequence of FBWT if it exists, if not, return $\perp$

1 *backwardSearch*(*c*,*l*-[*i*..*j*])

2 $i \leftarrow C_{l/k}[c] + Occ_{l/k}(c, i-1) + 1$
3 $j \leftarrow C_{l/k}[c] + Occ_{l/k}(c, j)$
4 **if** $i \leq j$ then **return** $(l-1)$-$[i..j]$
5 **else return** $\perp$

$C_{l/k}[c]$ returns the number of characters smaller than c in l-th FBWT. $Occ_{l/k}(c, i)$ returns the number of c in $FBWT_{l/k}[0 : i-1]$.

$FSA_{0/3}$		$FBWT_{0/3}$
12	$$$bananabanana	a
9	ana$$$bananaban	n
3	anabanana$$$ban	n
6	banana$$$banana	a
0	bananabanana$$$	$

$FSA_{1/3}$		$FBWT_{1/3}$
13	$$bananabanana$	$
7	anana$$$bananab	b
1	ananabanana$$$b	b
10	na$$$bananabana	a
4	nabanana$$$bana	a

$FSA_{2/3}$		$FBWT_{2/3}$
14	$bananabanana$$	$
11	a$$$bananabanan	n
5	abanana$$$banan	n
8	nana$$$bananaba	a
2	nanabanana$$$ba	a

Figure 1: FBWT for "bananabanana"

2 Method

2.1 Finding MEMs

Our approach is similar to that of essaMEM. First, we create an index from a reference genome. The size of the index can be change by the sparse parameter. Then, exact matching from some query positions for some length l is executed to find candidates. l can also control the number of positions of query, which starts the procedure. A larger l equals more positions, which leads to a slower execution. MEMs are finally extracted from the candidates. The second step is proportional to occurrence. The l parameter controls the trade off between the second step, the number of procedures, and the first step. When l is small, the number of positions to start procedures is small and leads to faster executions, but the second step becomes large because the number of candidates also increases. On the other hand, if l is large, the second step can be small, but the number of positions increases and leads to slower executions.

There are two major differences between our algorithm and essaMEM. One is that our index is based on FBWT as opposed to an enhanced suffix array in essaMEM. As a result, our index uses less memory than essaMEM, but we achieve a competitive performance. The other is that our approach dynamically changes parameter l, while essaMEM maintains it as a static value. When there are a lot of candidates, the second step reaches a bottleneck. However, the number of candidates depends on the query sequence. A number of queries have few candidates, but some have many candidates. Hence, we introduce a method to dynamically change the parameter. We found method has an impact on both execution time and parameter tuning cost. As a result of the dynamic change, FBWTMEM outperforms essaMEM and others.

2.2 Detail of Algorithm

In this section, we first describe the detail of algorithm, and then discuss its time complexity compared to that of essaMEM. The space complexity is described in the main paper. In the first step, similar to essaMEM, candidates are found by matching $l = L - k \cdot s + 1$ characters starting from $Q[i]$, where k is the size of FBWT or sparse parameter of essaMEM, and s is the skip parameter introduced in essaMEM. First intervals are calculated by hash and then calculated by index. After the process starts from $Q[i]$, the next process starts from $Q[i - k \cdot s]$. In essaMEM, the next process starts at $Q[i + k \cdot s]$ because it searches forward from the start, but our method search backward from the end, so the next process starts at $Q[i - k \cdot s]$. Our method stores $FSA_{0/k}$, and we adjust the end of matching at $FBWT_{0/k}$. While essaMEM fixes a s parameter for the whole procedure, our algorithm dynamically changes it in accordance with the number of candidates, which means if there are a lot of candidates, our algorithm can decrease s at that time. A number of MEMs will be lost when s is decreased, so we start additional MEM-finding procedures with matching $L - k \cdot s' + 1$ characters from $Q[i + k \cdot (s - s')]$ where s' is the number of times s decreases. When s' is significantly large, this step in the additional procedures cannot reduce the number of candidate sufficiently, and can lead to much slower executions because of the next step. Hence, we decided that s' would be less than half of s. The correction is described at the end of this section.

The second step is to determine if the candidates are longer than L and their length. To calculate the left side, our algorithm uses a backward search like backwardMEM. Once the interval is updated, the next character of query is compared with characters in FBWT corresponding to the interval. If the character does not match, the candidate cannot further extend the left side, while if it matches, the candidate can extend left more. When the number of candidates becomes low enough during the backward search, we halt the search and switch to directly comparing the candidates to the reference. In our implementation, you can set this parameter to the ratio of the number of first candidates, in which the maximum value is 1000. When you set the parameter to 10, FBWTMEM directly compares the candidates to the reference when the number of candidates becomes under 1% of that of the first candidate. We added this heuristic because it is reasonable to expect that the interval size rarely changes when the matching length is long. essaMEM avoids this traversal with no reduction of interval, so this heuristic is needed to outperform essaMEM. After the left side is calculated, the right side is calculated, and $k \cdot s$ characters between the reference and query are directly compared. If they match more to those on the right side, the candidate has been extracted by the previous MEM-finding procedure. Algorithm 2 shows the procedure. $collectMEM$ function is the main function. Because we have only $FSA_{0/k}$, we hold the array interval during the calculation. We used c++ vector for the container of the candidates, and the erase method when the candidate is end.

Listing 2: Calculate left side 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 ≥ L then
6        MEM.push(sa,leftMatchLength + L − k\cdot 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],
              candidate, queryStartSearchPos − queryPos)
35        l-[i..j] ← backwardSearch(query[queryPos], l-[i..j])
36        queryPos = queryPos − 1
```

Next, we discuss the time complexity compared to essaMEM. The first step of the algorithm is to traverse FBWT, so its time complexity is $O(L - k \cdot s + 1)$. The last step needs to check all elements in the interval after traversing FBWT, so the time complexity is bound by the total number of occurrences. We assume that m is the maximal matching length, in which the worst case is when the interval size does not decrease until the matching length reaches m. The worst time complexity is $O(m \cdot occ)$, where occ is the first interval size, while that of essaMEM is $O(m + occ)$. The time complexity is the same as that of essaMEM in the first step but worse in the second step than that of essaMEM, but our experiments show that our method is competitive to essaMEM.

The correctness of the algorithm is described in the paper of essaMEM when the s parameter is static, so we discuss lost MEMs and how to find them when s is decreased. Let's compare the MEMs that matched $query[i - (L - k \cdot s + 1) + 1 : i]$ and $query[i - (L - k \cdot (s - s') + 1) + 1 : i]$ in the first step. The characteristic of lost MEMs is that they start from $query[j]$ $(i - (L - k \cdot (s - s') + 1) + 1 < j < i - (L - k \cdot s + 1))$ or end at $query[j]$ $(i - (L - k \cdot (s - s's) + 1) < j < i + k \cdot s - 1)$. These MEMs can be found by matching $query[i - L + k \cdot s : i + k \cdot (s - s')]$ in the first step. Hence, our algorithm can find all MEMs.

2.3 Implementation

We used a suffix array construction implementation from https://sites.google.com/site/yuta256/sais. The other implementation is our original. You can find our application at https://github.com/MasaruIto/FBWTMEM, which can be compiled using C++14.

3 Experimental Results

We measured the elapsed times of the MEM-finding functions by the *gettimeofday* function, and the memory size by read /proc/{$pid}/smaps. All tests were run on a Red Hat Enterprise Linux Server release 6.6 machine featuring an Intel Xeon CPU E5-2690 clocked at 2.90GHz with 100GB of RAM. We evaluated the performance of FBWTMEM against existing algorithms such as essaMEM (Vyverman *et al.* (2013)), backwardMEM (**?**), slaMEM (Fernandes and Freitas (2014)) and EMEM (Khiste and Ilie (2015)). MUMmer (Kurtz *et al.* (2004)), Vmatch (http://www.vmatch.de/), sparseMEM (Khan *et al.* (2009)) was outperformed by essaMEM, so we did not measure them. Table 1 shows the datasets, and Table 2 shows the results of each algorithms and datasets. We used the largest value of s under $L - s \cdot K + 1 \geq hashLen$ for FBWTMEM. For Mbp size genomes, the hash size becomes too large to ignore when the size of FBWT increases, so we decrease $hashLen$ until the size is under $2 \cdot 4^{11}$bytes, and we used the k parameter in 1, 2, 4, 8, 16, and 32 if possible. For Gbp datasets, we used $hashLen = 10$, and the k is in 3, 4, 8, 16, 32, and 64. The threshold switching to directly compare with the reference in the second step of the algorithm is 10, and the threshold to decrease the s parameter in the first step of the algorithm is 10. For essaMEM, we used an optimal value of s, the hash length is 10, and the K parameter is same as above. The command line options are -n, -child 1, -suflink 0, -kmer 10, -maxmatch and -skip <optimal>. For backwardMEM, we used the K parameter in 1, 2, 4, 8, 16, and 32 if possible for Mbp size genomes, and did not apply it for Gbp genomes. The command line

Table 1: Datasets property. MEM len means minimal length of MEMs

id	reference	size[bp]	query	size[bp]	MEM len	# of MEMs	average length of MEMs
1	fumigatus	29M	nidulans	30M	20	330150	25.9432
2	sapiencs21	46M	musculus16	95M	50	586296	55.0137
3	musculus16	95M	sapiencs21	46M	50	586296	55.0137
4	simulans	135M	sechellia	162M	50	18643313	70.2097
5	melanogaster	140M	sechellia	163M	50	2619476	85.9191
6	melanogaster	140M	yakuba	162M	50	870653	82.0591
7	hg19	3.1G	panTro3	3.2G	100	132368058	127.01
8	mm10	2.7G	hg19	3.1G	100	554327	114.753

options are -n and -maxmatch. For EMEM, and the command line option is -n. For slaMEM, the command line option is -n.

Figures 2, 6, 7, 3, 4, 9, 5 and 8 show the results of each datasets and each algorithm and their parameters. FBWTMEM has a better or competitive performance against the other algorithms for all datasets. Discussions about each algorithm are as follows.

- backwardMEM
  backwardMEM was clearly outperformed by other algorithms.
- slaMEM
  slaMEM has a competitive performance against FBWTMEM, essaMEM, and EMEM in our datasets. However, it has a single sample and is not the best for any datasets.
- EMEM
  EMEM has a small memory consumption for large lengths of minimal MEMs. In datasets in which the minimal length of MEMs is 50, EMEM has a competitive performance against essaMEM and slaMEM, and in Homo sapiencs21 versus Mus musculus16 datasets, it uses the smallest amount of memory. However, it does not have a best execution time and is outperformed in Gbp datasets in these experiements. According to the EMEM paper, it can deal with larger datasets, so it is useful for such genomes.
- essaMEM
  essaMEM has a competitive performance against FBWTMEM, but is outperformed in all trials in comparable memory footprint. In the No. 4 and No. 5 datasets, the fastest sample of essaMEM is 1.15 times faster than our method, while 1.1 times faster in No. 6. In No. 7, described in the paper, the acceleration ratio shrinks compared to that in No. 8. These datasets have many MEMs compared to others shown in Table 1. When there are more occurrences, FBWTMEM may be outperformed by essaMEM, however, FBWTMEM is useful for real datasets.

4 Impact of heuristics

In this section, we discuss the performance of essaMEM and FBWTMEM in detail. For each trial, each method uses $k = 1$, hash key length is 10, threshold of switching matching to directly compare with the reference in step two of the algorithm is 10, threshold of the number of candidates to dynamic reduction of skip parameter is 10, and all datasets are used. The memory footprint in FBWTMEM is about twice as small as essaMEM.

Figures 10, 11, 12, 13, 14, 15, 16 and 17 show the results of each dataset for essaMEM with a child table and no suffix link, FBWTMEM without any heuristics, FBWTMEM with direct compare heuristic, and FBWTMEM with direct compare heuristic and dynamic skip parameter heuristic. We discuss the

comparison of the FBWTMEM heuristics and compare those with essaMEM. First, we describe the impact of the direct compare heuristic. Comparing the no-heuristic one and the direct-compare-heuristic one, there is some effect in datasets No. 1, No. 4, No. 5, and No. 6 in figures 10, 13, 14 and 15, while there is little effect in datasets No. 2, No. 3, No. 7, and No. 8 in figures 11, 12, 16 and 17. The major difference of the datasets, regardless of effect, is the length of MEMs. Table 1 shows the average length of the MEMs for each dataset. The datasets having an effect has relatively longer MEMs than the minimal MEM length. Longer MEMs lead to a longer execution time of the second step of the algorithm. The traverse with FBWT is memory inefficient, while directly comparing is efficient. This result suggests it is effective when the number of candidates is small.

Next, the impact of the dynamic skip parameter. The largest skip parameter is not the fastest one in the no-heuristic one and the direct-compare one. The skip parameter controls the exact matching length in the first step of the algorithm and can control the trade off between the first and second step. The large one fails to fully utilize the trade off, but the fastest one depends on the datasets. On the other hand, the dynamic-skip-parameter one reduces the decrease of performance in the largest skip parameter. Although the largest one is not always the best, you can achieve a better performance.

Finally, we compare our algorithms with essaMEM. The no-heuristic and direct-compare FBWTMEM outperforms or competes with essaMEM with a small skip parameter. However, with a large parameter, FBWTMEM is usually outperformed. However, the dynamic one outperforms essaMEM with all samples. essaMEM has the same phenomenon in the largest skip parameter, so it can be competitive to our method if it uses the dynamic skip parameter. However, our method uses less than twice that of essaMEM, so our method is more useful.

References

Fernandes, F. and Freitas, A. T. (2014). slamem: efficient retrieval of maximal exact matches using a sampled lcp array. *Bioinformatics*, **30**, 464–471.

Ito, M., Inoue, H., and Taura, K. (2016). Fragmented BWT: An Extended BWT for Full-Text Indexing. *Proceedings of the 23rd Annual Symposium on String Processing and Information Retrieval.*, pages 97–109.

Khan, Z., Bloom, J. S., Kruglyak, L., and Singh, M. (2009). A practical algorithm for finding maximal exact matches in large sequence datasets using sparse suffix arrays. *Bioinformatics*, **25**, 1609–1616.

Khiste, N. and Ilie, L. (2015). E-MEM: efficient computation of maximal exact matches for very large genomes. *Bioinformatics*, **32**(4), 509–514.

Kurtz, S., Phillippy, A., Delcher, A. L., Smoot, M., Shumway, M., Antonescu, C., and Salzberg, S. L. (2004). Versatile and open software for comparing large genomes. *Genome Biology*, **5**(R12).

Vyverman, M., Baets, B. D., Fack, V., and Dawyndt, P. (2013). essaMEM: finding maximal exact matches using enhanced sparse suffix arrays. *Bioinformatics*, **29**(6), 802–804.

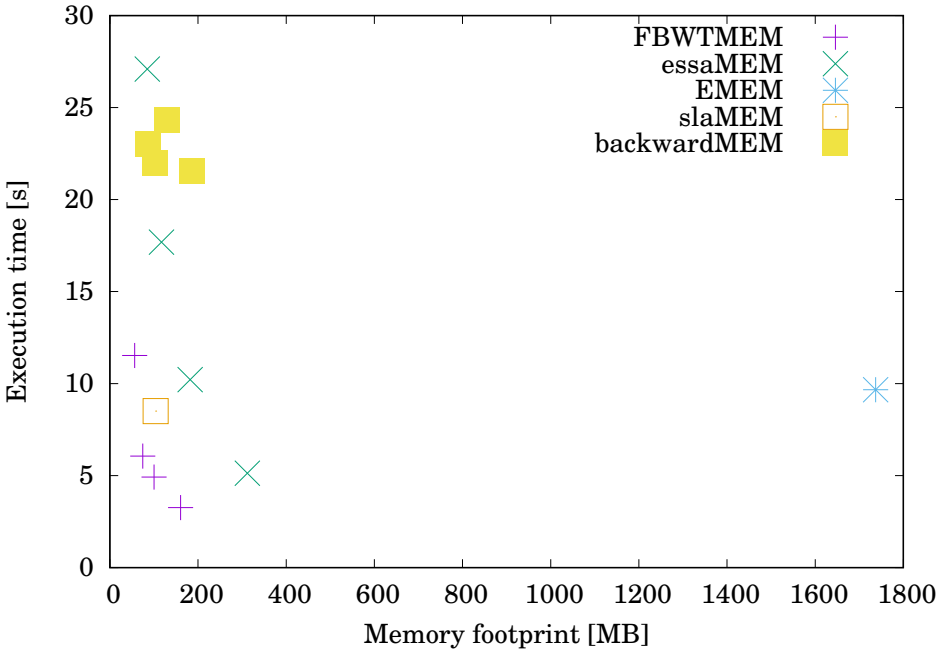

Figure 2: No. 1 dataset comparing time and memory footprint of MEMs computed between Aspergillus fumigatus versus Aspergillus nidulans. Minimal MEM length is 20

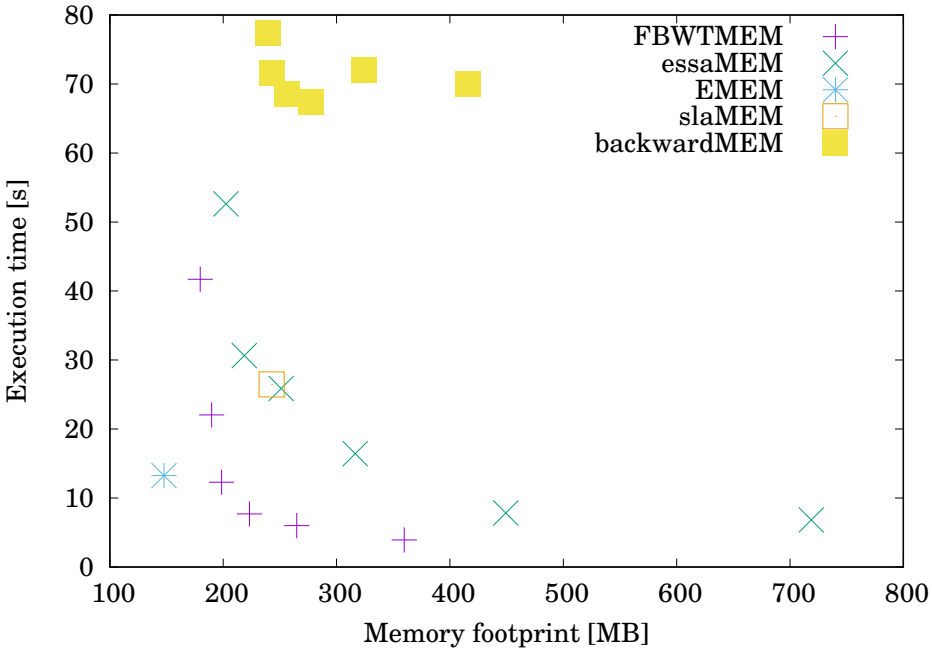

Figure 3: No. 2 dataset comparing time and memory footprint of MEMs computed between Homo sapiencs 21 and Mus musculus 16. Homo sapiencs 21 is indexed. Minimal MEM length is 50

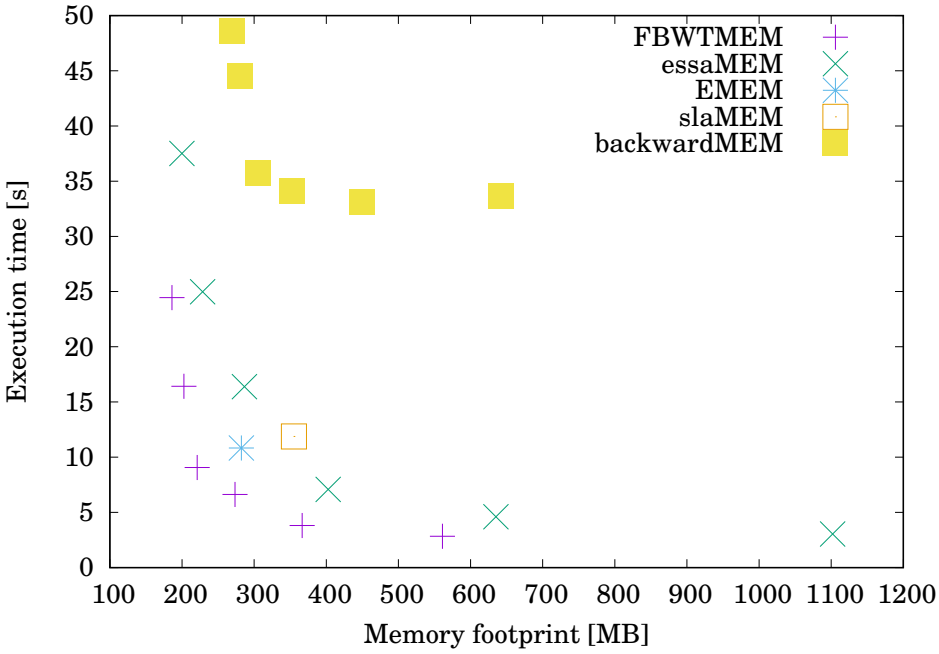

Figure 4: No. 3 dataset comparing time and memory footprint of MEMs computed between Mus musculus 16 and Homo sapiencs 21. Mus musculus 16 is indexed. Minimal MEM length is 50

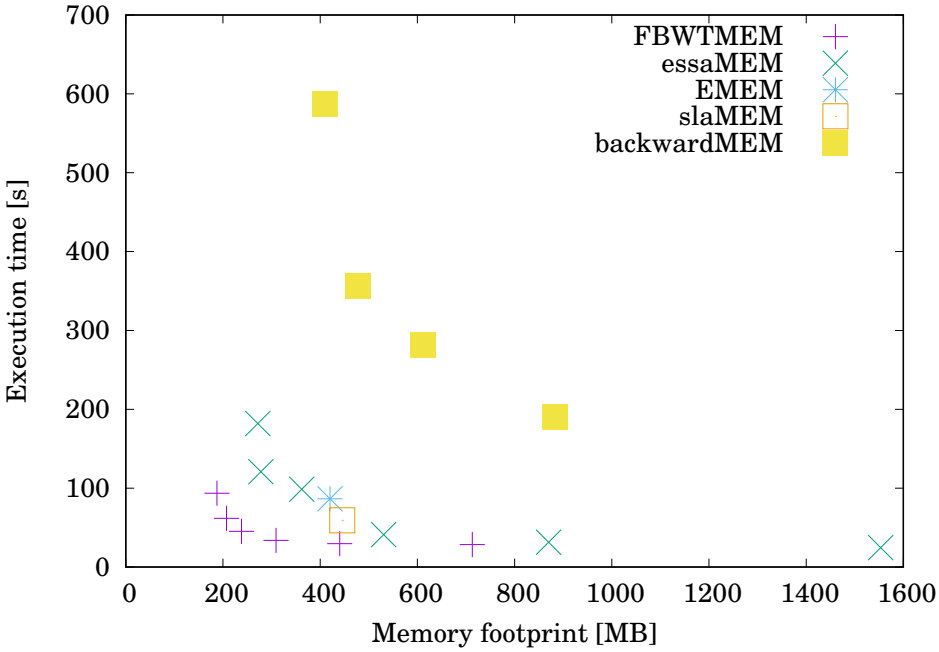

Figure 5: No. 4 dataset comparing time and memory footprint of MEMs computed between Drosophila simulans and sechellia. Minimal MEM length is 50

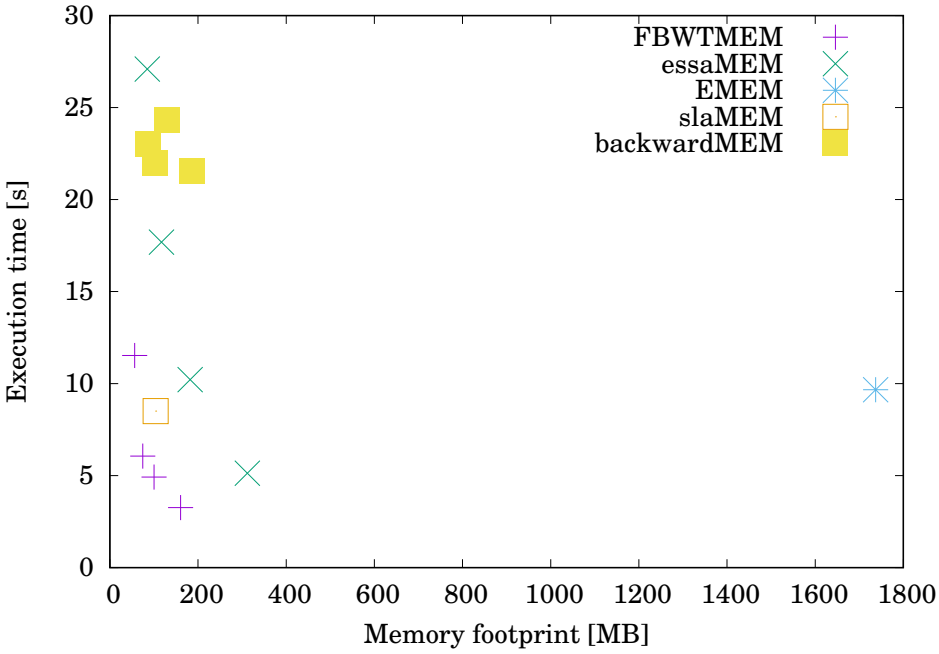

Figure 6: No. 5 dataset comparing time and memory footprint of MEMs computed between Drosophila melanogaster and sechellia. Minimal MEM length is 50

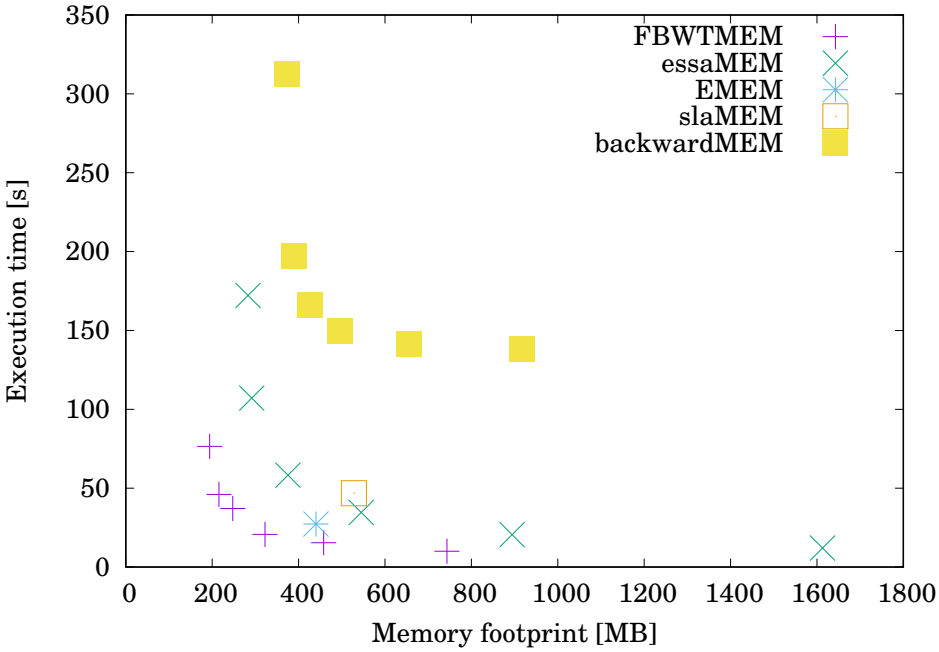

Figure 7: No. 6 dataset comparing time and memory footprint of MEMs computed between Drosophila melanogaster and yakuba. Minimal MEM length is 50

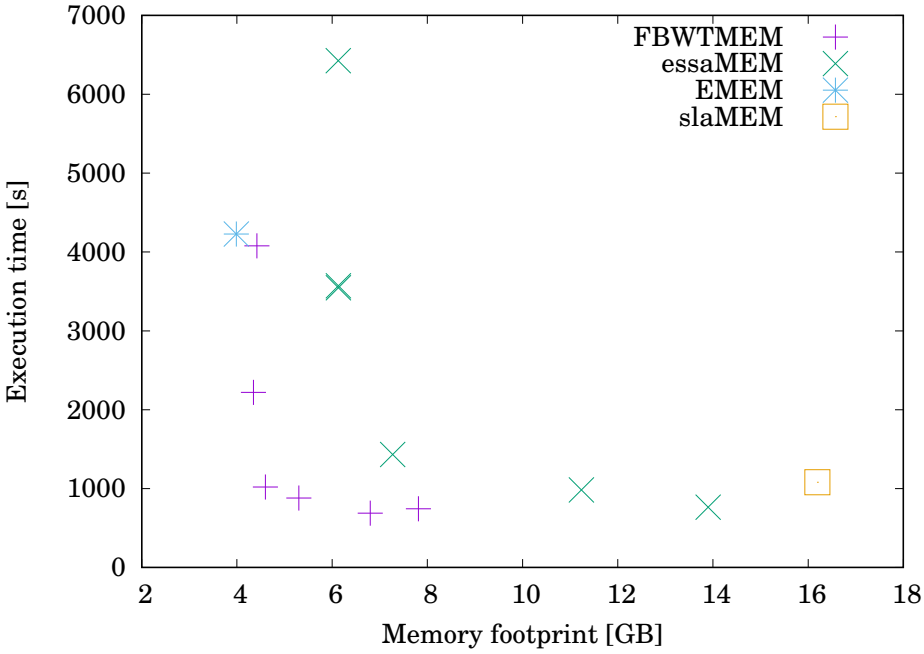

Figure 8: No. 7 dataset comparing time and memory footprint of MEMs computed between Human genome (hg19) and Chimpanzee (panTro3). Minimal MEM length is 100

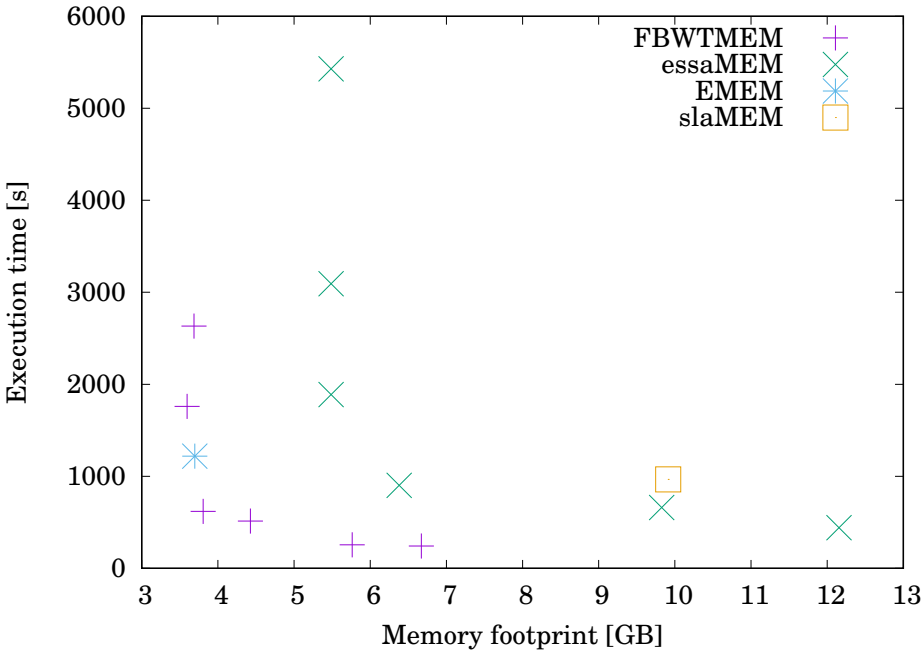

Figure 9: No. 8 dataset comparing time and memory footprint of MEMs computed between Mus musculus (mm10) and Human genome (hg19). Minimal MEM length is 100

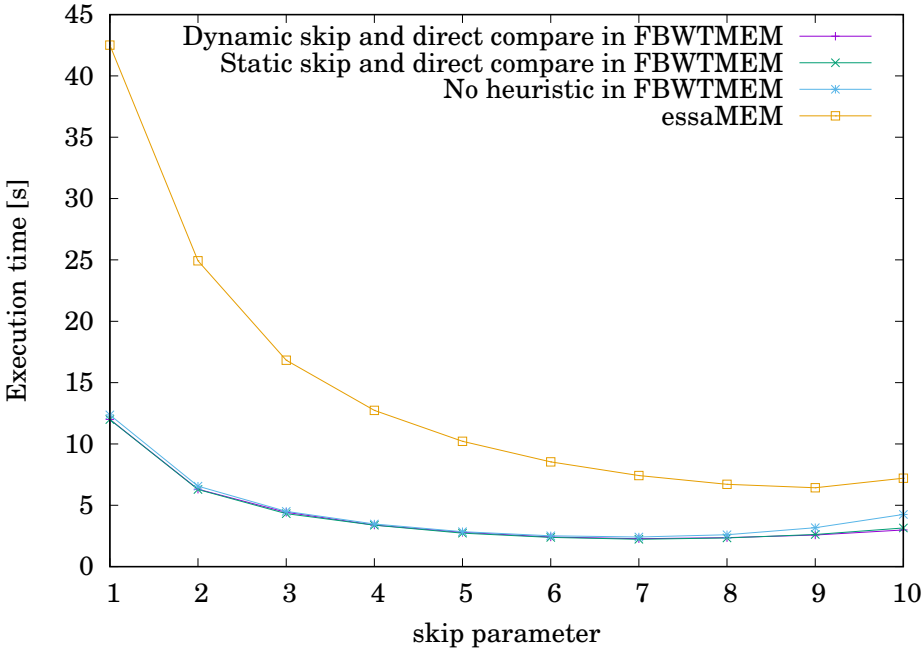

Figure 10: No. 1 dataset comparing time and skip parameter. essaMEM with child table not suffix link, FBWTMEM without any heuristics, FBWTMEM with direct compare heuristic, and FBWTMEM with direct compare heuristic and dynamic skip parameter heuristic. essaMEM and all FBWTMEM trials uses 1 as sparse parameter. The default parameter is the skip parameter in the dynamic one.

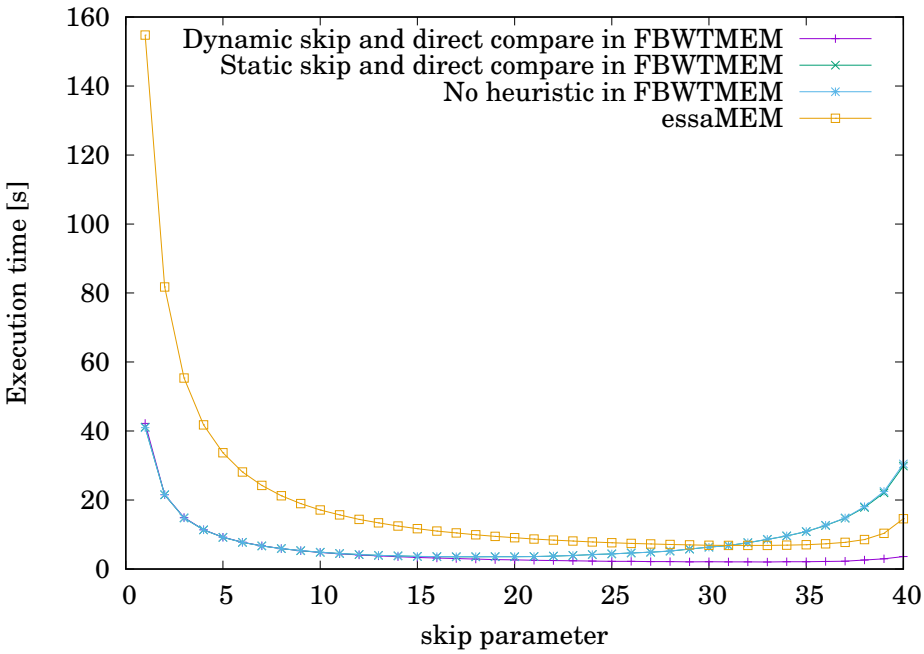

Figure 11: No. 2 dataset comparing time and skip parameter. essaMEM with child table not suffix link, FBWTMEM without any heuristics, FBWTMEM with direct compare heuristic, and FBWTMEM with direct compare heuristic and dynamic skip parameter heuristic. essaMEM and all FBWTMEM trials uses 1 as sparse parameter. The default parameter is the skip parameter in the dynamic one.

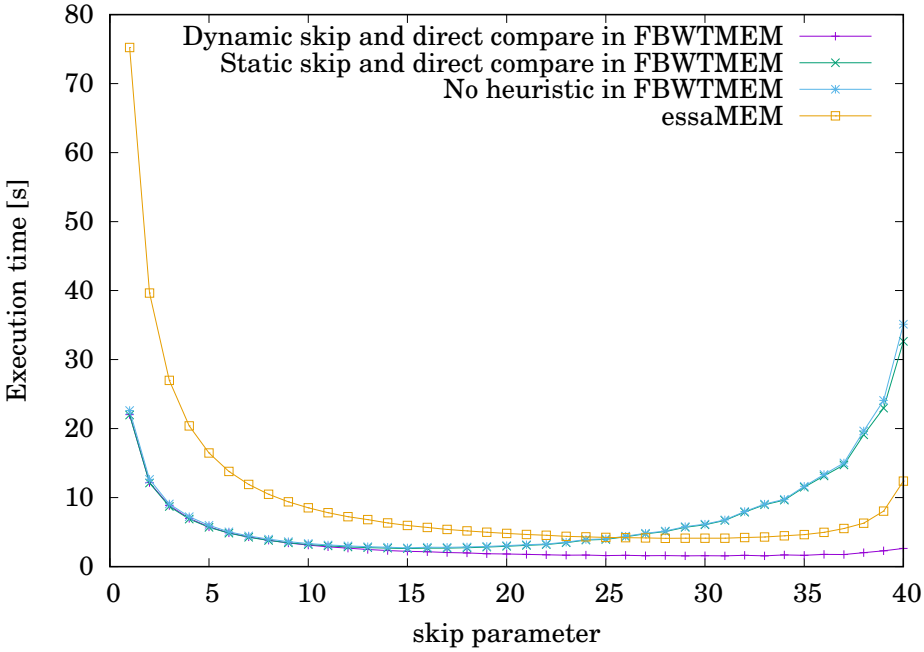

Figure 12: No. 3 dataset comparing time and skip parameter. essaMEM with child table not suffix link, FBWTMEM without any heuristics, FBWTMEM with direct compare heuristic, and FBWTMEM with direct compare heuristic and dynamic skip parameter heuristic. essaMEM and all FBWTMEM trials uses 1 as sparse parameter. The default parameter is the skip parameter in the dynamic one.

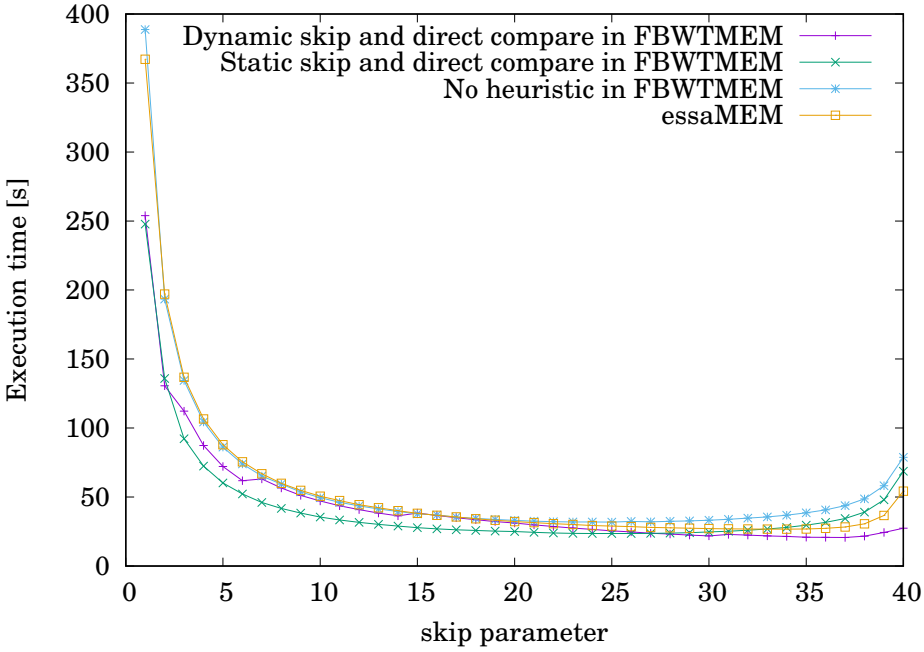

Figure 13: No. 4 dataset comparing time and skip parameter. essaMEM with child table not suffix link, FBWTMEM without any heuristics, FBWTMEM with direct compare heuristic, and FBWTMEM with direct compare heuristic and dynamic skip parameter heuristic. essaMEM and all FBWTMEM trials uses 1 as sparse parameter. The default parameter is the skip parameter in the dynamic one.

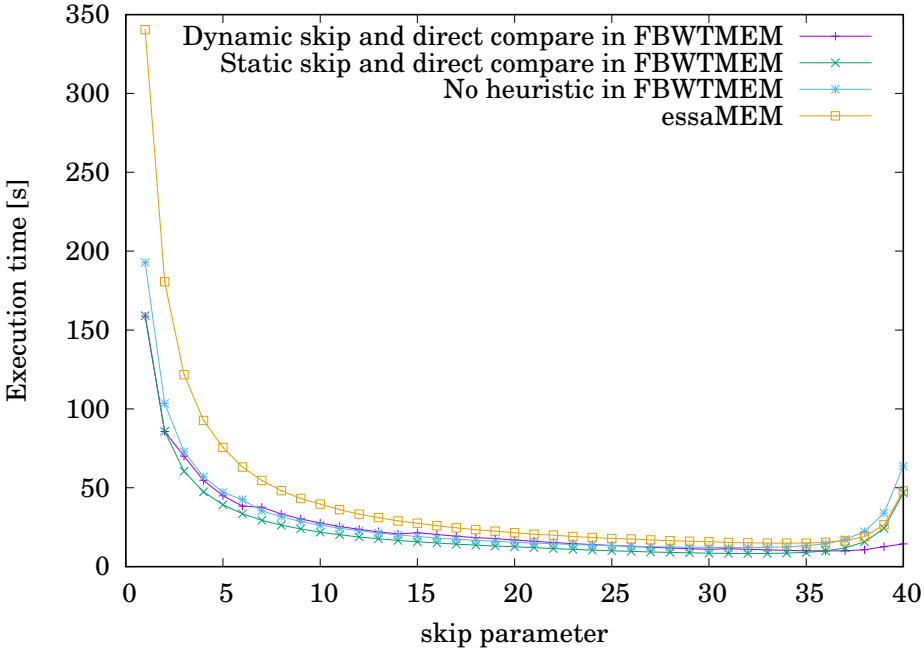

Figure 14: No. 5 dataset comparing time and skip parameter. essaMEM with child table not suffix link, FBWTMEM without any heuristics, FBWTMEM with direct compare heuristic, and FBWTMEM with direct compare heuristic and dynamic skip parameter heuristic. essaMEM and all FBWTMEM trials uses 1 as sparse parameter. The default parameter is the skip parameter in the dynamic one.

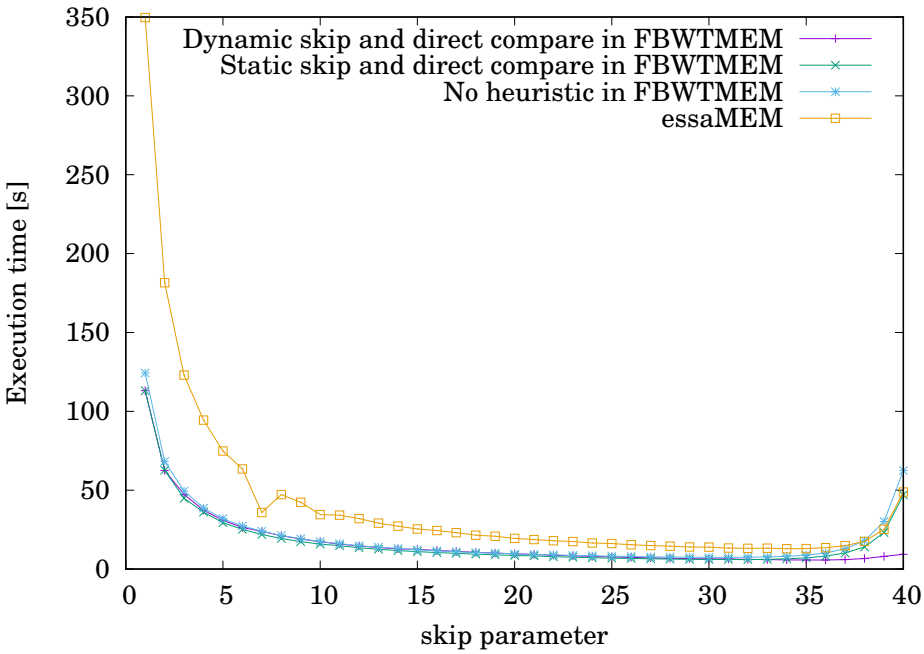

Figure 15: No. 6 dataset comparing time and skip parameter. essaMEM with child table not suffix link, FBWTMEM without any heuristics, FBWTMEM with direct compare heuristic, and FBWTMEM with direct compare heuristic and dynamic skip parameter heuristic. essaMEM and all FBWTMEM trials uses 1 as sparse parameter. The default parameter is the skip parameter in the dynamic one.

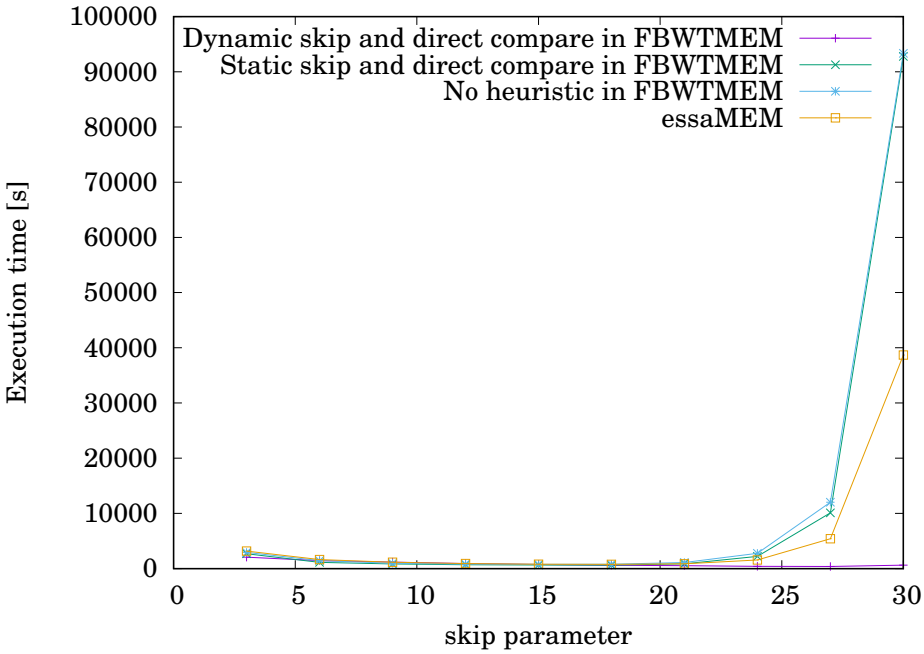

Figure 16: No. 7 dataset comparing time and skip parameter. essaMEM with child table not suffix link, FBWTMEM without any heuristics, FBWTMEM with direct compare heuristic, and FBWTMEM with direct compare heuristic and dynamic skip parameter heuristic. essaMEM and all FBWTMEM trials uses 1 as sparse parameter.

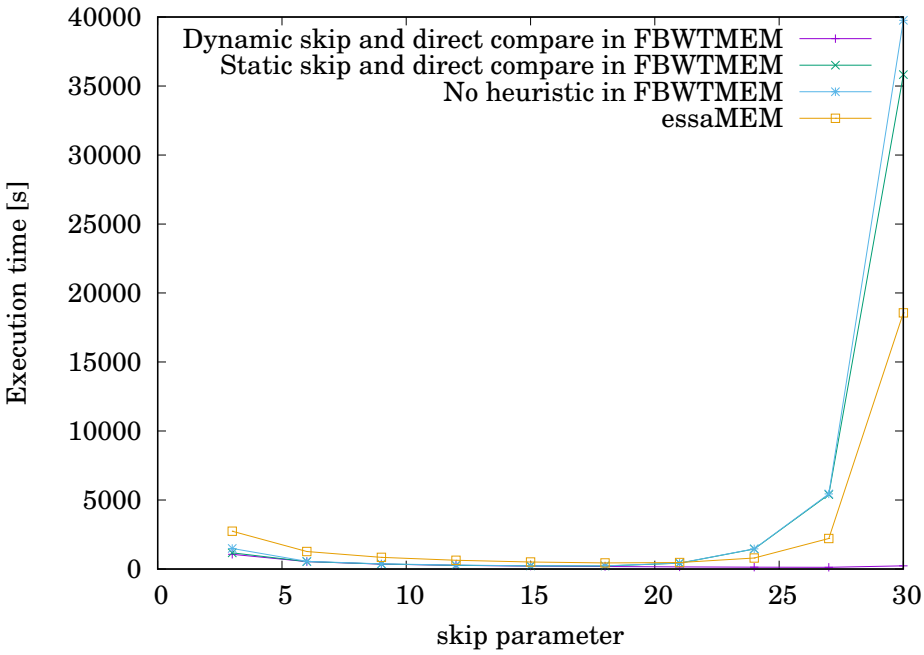

Figure 17: No. 8 dataset comparing time and skip parameter. essaMEM with child table not suffix link, FBWTMEM without any heuristics, FBWTMEM with direct compare heuristic, and FBWTMEM with direct compare heuristic and dynamic skip parameter heuristic. essaMEM and all FBWTMEM trials uses 1 as sparse parameter.

Table 2: Raw results. If there is no sparse option, the result is in the smallest sparse value row, id column corresponds to table 1, s column is sparse parameter, m means memory, t means time.

id	s	FBWTMEM		backwardMEM		essaMEM		slaMEM		EMEM	
		m[kB]	t[s]	m[kB]	t[s]	m[kB]	t[s]	m[kB]	t[s]	m[kB]	t[s]
1	1	160336	3.25718	185340	21.5531	311748	5.13078	103902	8.50493	1737204	9.66601
1	2	100196	4.92304	129064	24.3108	182004	10.2135	-	-	-	-
1	4	74544	6.05709	101308	21.9792	117148	17.6888	-	-	-	-
1	8	56316	11.5249	85624	23.0464	84824	27.0931	-	-	-	-
2	1	359688	3.91892	415732	69.9771	718804	6.82268	242768	26.4675	147756	13.2591
2	2	264808	6.00807	324512	72.0369	449296	7.83567	-	-	-	-
2	4	223116	7.69696	277404	67.3246	316504	16.4193	-	-	-	-
2	8	198608	12.2991	256432	68.5254	251168	25.8659	-	-	-	-
2	16	189740	22.035	243068	71.6184	218648	30.6376	-	-	-	-
2	32	179684	41.6963	239164	77.3257	202500	52.6254	-	-	-	-
3	1	561056	2.83641	642008	33.6146	1101932	3.03844	355218	11.8836	282292	10.8333
3	2	366492	3.81551	449796	33.1348	635272	4.59987	-	-	-	-
3	4	273448	6.62356	352536	34.105	402560	7.07815	-	-	-	-
3	8	220980	9.06654	305520	35.7553	286532	16.381	-	-	-	-
3	16	202744	16.4236	280856	44.4799	228668	24.9821	-	-	-	-
3	32	186052	24.447	269052	48.6102	199932	37.5089	-	-	-	-
4	1	713144	28.4086	883168	190.377	1553324	24.6267	445318	59.2582	420412	86.4835
4	2	440076	29.6161	612416	282.054	869928	31.5293	-	-	-	-
4	4	309216	33.8769	478080	356.74	530732	41.0642	-	-	-	-
4	8	238040	45.2837	410828	586.937	362000	98.5661	-	-	-	-
4	16	207196	61.8441	378568	1146.64	277932	120.897	-	-	-	-
4	32	187524	93.6093	361296	2347.93	271816	181.884	-	-	-	-
5	1	737988	15.1045	918620	154.758	1604228	13.1101	521658	44.3435	440868	27.2401
5	2	452864	19.6291	637580	148.15	886024	20.8738	-	-	-	-
5	4	316184	26.0372	498932	177.83	536612	33.4465	-	-	-	-
5	8	241948	37.7353	426820	248.145	366336	56.5269	-	-	-	-
5	16	211260	48.3297	392848	369.858	282512	96.5191	-	-	-	-
5	32	190148	77.7097	374352	673.786	282808	164.884	-	-	-	-
6	1	743516	10.0173	917364	138.215	1606248	11.4272	528110	46.8108	440260	27.2414
6	2	458028	15.4067	654984	141.288	887368	21.3334	-	-	-	-
6	4	322124	20.6658	496032	149.576	536628	36.4029	-	-	-	-
6	8	247772	37.1159	425420	165.848	365744	64.3609	-	-	-	-
6	16	215796	46.0367	390604	197.498	282156	109.382	-	-	-	-
6	32	194072	76.4607	371984	312.328	283360	173.165	-	-	-	-
7	3	7812356	743.703	-	-	13899224	763.369	16192772	1080.46	3986952	4227.33
7	4	6801056	688.238	-	-	11235108	983.305	-	-	-	-
7	8	5300456	880.633	-	-	7272276	1433.1	-	-	-	-
7	16	4598452	1019.19	-	-	6130708	3545.58	-	-	-	-
7	32	4346584	2220.56	-	-	6130576	3571.39	-	-	-	-
7	64	4420452	4078.09	-	-	6130168	6425.84	-	-	-	-
8	3	6671612	241.755	-	-	12154212	442.055	9911764	966.823	3697616	1218.62
8	4	5763852	255.093	-	-	9826568	661.148	-	-	-	-
8	8	4427284	513.935	-	-	6379948	901.597	-	-	-	-
8	16	3807516	619.401	-	-	5485792	1888.77	-	-	-	-
8	32	3595876	1759.55	-	-	5486632	3092.43	-	-	-	-
8	64	3687192	2632.85	-	-	5486496	5428.21	-	-	-	-