Box Project

Alexis Hamon, Benjamin Voisin
ENS Rennes
{alexis.hamon, benjamin.voisin}@ens-rennes.fr

I. Introduction

The goal of this project is to compute minimal absent words (MAWs) of sequences of DNA (fixed-size alphabet).

We present a linear algorithm using suffix-trees, as well as an efficient C implementation that is competitive (faster) than state of the-art[1].

Our open-source implementation is freely available here : framagit.org/AlexisHamon/box_project

Our implementation does not (yet) exactly match what was asked for the project. Precisely, we don't check for complements, and only calculate MAWs of individual sequences (instead of MAWs of the whole set of sequences). However, we describe in section II-G1 and II-G2 respectively how to address theese issue theorically.

In addition, it is important to say that we haven't looked at state-of-the arts articles (exception for the two in bibliography) as you said it was important not to. Hence, it is possible that we do not fit with conventions which this domain is used to, and they may be some trivial improvements we don't even know about.

II. Method

We build the trees using an implentation of the well-known ukonen algorithm[2].

A. Generalities

Let S be a sequence and consider only minimal absent words of size at least two.

a) Theorem: Let $w = c_1 w' c_2$. w is a minimal absent word of S iff w is not a factor of f but $c_1 w'$ and $w' c_2$ are.

b) Proof:

$\Rightarrow$ by definition of minimal absent words

$\Leftarrow$ let x be a factor of, w then it's either a factor of $c_1 w'$ or $w' c_2$ so it's also a factor of S

B. Generatities II

Let suppose $w = c_1 w' c_2$ is a minimal absent word of S.

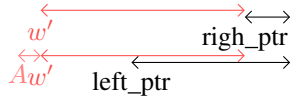
Since $c_1 w'$ and $w' c_2$ are two factors of S [Th 1] they are both prefixes of suffixes of S.

So they appear in the suffix tree of S.

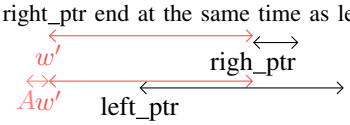
C. Is this word an absent word ?

So if we need to check whether $w = c_1 w' c_2$ is a minimal absent word of S. The only thing we have to do is to check is

1) $c_1 w'$ appear in the suffix tree



(a) If right_ptr end at the same time as left_ptr



(b) If right_ptr stops before

Fig. 1: Two cases of the main algorithm iteration scheme

2) $w' c_2$ appear in the suffix tree
3) $c_1 w' c_2$ do not appear in the suffix tree

That takes at most $\mathcal{O}(min(k, n))$ operation, where k is the length of the word to query and n the length of the sequence S.

That is way faster than the binary search in the sorted list of minimum absent words, which is in $\mathcal{O}(klog(\sigma n))$.

In conclusion, if your goal is to fulfill a huge list of request about whether some words are minimal absent words of, S it may be significantly faster to build the suffix tree and request by iterate through it on it instead of building the sorted absent word list and query whith a binary search on that list afterward.

D. Fetch all absent words ?

Let say $c_1 = A$, we will construct all w' possible such as Aw' is still in the suffix tree. In fact, if Aw' is in the suffix tree, so is w'. Hence, there are three possibilities to append w'.

We are defining two pointers during our tree traversal, `left_ptr` which contains the factor w_l which represent the node where Aw' stops in the suffix tree and `right_ptr` which contains the factor w_r which represent the node where w' stops in the suffix tree.

1) if `right_ptr` is a leaf so is `left_ptr`, we can end up there is no absent word
2) if `right_ptr` node has a length smaller, see 1b
   in that case, right_ptr as at least two children. One of them begins with the next character of `left_ptr` so we can append that character to w' and continues our tree traversal. However, for each other child of `left_ptr` which begins by character c, $Aw'w_r c$ is an absent word.
3) on the other case, see 1a we iterate on `righ_ptr` children. If `left_ptr` has the same child, we can continue to iterate, on the other, and we've found an absent word which is $Aw'w_r c$.

If we iterate that process for each starting point, eg the letter of the alphabet we can easeally fetch all absent words.

E. Correctness

⇒ If our algorithm returns a minimal absent word it looks like $Aw'w_rc$ and we know that $Aw'w_r$ and $w'w_rc$ are two prefixes of suffixes of S as they can be found by the suffix tree. Although, $Aw'w_rc$ cannot so, it is a minimal absent word.

⇐ According to Th1, if a word is a minimal absent word of S, it can be decomposed in two factor of S which are represented in the suffix tree. There is no chance for a traversal to not encounter them.

In conclusion, our algorithm only finds minimal absent words and it finds all of them.

F. Algorithm

For each node, we store the number of the suffix that created this node, as well as the start of the spacing.

But we followed the suffix that created the node in the radix tree, to recover the MAW. So we got a compressed version of our MAW.

We then need to find in the sequence the start of this suffix, to the end of the offset, and add the character c.

G. Extend

1) Extend to handle complements: We have some ideas to extend our approach in order to handle complements.
The first one is to create a merged suffix tree between the one of the sequence and its complement.
We thought that it can be done by using Ukkonnen's algorithm on the sequence and its reverse by respecting sequence's length priority. It may avoid creating two distinct trees and to merge them afterward.

However, we haven't got the time to spend more than some thought of it.

2) Extend to MAW of the set sequences: As for complements, the naïve idea is to create all trees and to merge them one after another. We also think about using a generalized Ukkonen in order only to create one tree.

In both extend cases, once we got the tree, we can use the same algorithm in order to ensure the tree traversal and gathering of all absent word. We have good hope some people have already found a way to create such trees, but we haven't looked at it yet.

H. Time Complexity

For a sequence of size n which contains θ absent words on a σ-sized alphabet.

Our algorithm splits in tho phases :

1) The suffix tree construction: The construction of the tree, which is linear according to Ukkonnen's algorithm $\mathcal{O}(n)$.

Algorithm 1: Algorithm to find MAWs

```
Data: A suffix tree
Result: All the MAW of the sequence represented by
        the suffix tree
if leaf(righ_ptr) then
    /* We are in the case of 1        */
    exit;
end
if left_ptr et righ_ptr end at the same time
 then
    /* We are in the case of 3        */
    for c, possible child of a node do
        if c child left_ptr then
            IterNode(left_ptr[c], 0, righ_ptr[c]);
        else
            if f child righ_ptr then
                Aw'w_rc is a minimal absent word;
            end
        end
    end
else
    /* We are in the case of 2        */
    u = the string containing the letters between
     left_ptr and right_ptr;
    IterNode(left_ptr, offset+length(righ_ptr),
     righ_ptr[u]);
    for c, possible child of a node do
        if u ≠ c c fils de righ_ptr then
            Aw'w_rc is a minimal absent word;
        end
    end
end
```

Algorithm 2: Algorithm to recover MAW from compressed version

```
Data: S, i_suffix, i_end, c
Result: the minimal absent word recovered
L = emptyList();
for i=i_suffix to i=i_end do
    L.append(S[i]);
end
L.append(c);
return L;
```

2) The suffix tree traversal: Firstly, the traversal of the tree. In the worst case, we can approximate this to $\mathcal{O}(\sigma n)$ with σ the size of the alphabet. In fact, if we see from afar we are doing at worst σ traversal of the tree which is in $\mathcal{O}(n)$.

So our algorithm has a complexity of $\mathcal{O}(n(\sigma+1))$ which is linear for fixed size alphabet. In practice, it takes almost the same time in order to construct the suffix-tree and to iterate through it.
That's very fine if we consider that the maximum number of minimal absent word is $\mathcal{O}(n\sigma)$.

I. Space complexity

Given n the length of the sequence (in numbers of chars).

- In each leaf of the tree, we store the index of the start of the node, so in $\mathcal{O}(\log n)$ (because to store n possible value, we need $\log_2 n$ bits).
- In each internal node of the tree, we store the index of the start and the index of the end of the node, as well as the suffix-links associated, and an array of 4 possible children, so in $\mathcal{O}(8 \log n)$
- Each tree has at max n leafs and n internal node. Thus, we have a space complexity of $\mathcal{O}(9n \log n)$. With 16 bits integers, we can have an upper bound of $\mathcal{O}(36n)$.

So, for the first assembled chromosome, taking 24.1 MiB, the suffix-tree will take 864 MiB (as seen in figure 4).
Hence, we still have some ideas to reduce that factor :

- **for internal nodes**
  1) avoid storing `i_suf` because we can recover that value during the tree traversal
  2) store `i_len` with a short integer instead of a long one
  3) free suffix links after the tree creation

In that case, we would have an upper bound of $\mathcal{O}(30n)$ during the tree creation and $\mathcal{O}(26n)$ during the tree traversal. That pretty fine result if we consider that computing the SA + LCP array cost approximately $\mathcal{O}(30n)$ according to `heaptrack` of maw.

III. Results

The results here are presented without taking in consideration the time to write the output, as this is just an implementation detail and blur the comparison between the core algorithms. Also, for the same reasons, all benchmarks are done without multi-threading (but our implementation appears to be much faster in multi-thread mode).

A. Linear in practice

As suggested in section II-H, the construction of the suffix tree, and our algorithm to find absent-words should be linear in the size of the sequence. This has been verified experimentally, as shown in figure 2a and 2b.

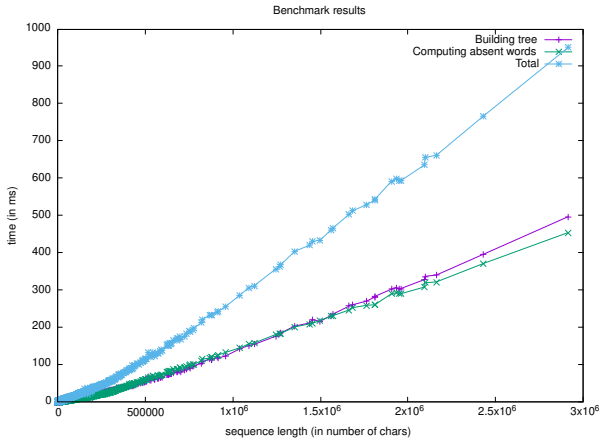
For both figures, the purple line show the time needed to build the suffix tree, and the green line show the time to run the actual algorithm. The cyan line is just the sum of the two other lines.

Figure 2a show details for a dataset of small sequences (*all_ebi_plasmids*), and figure 2b show details for much larger sequences, as found in the human genome.

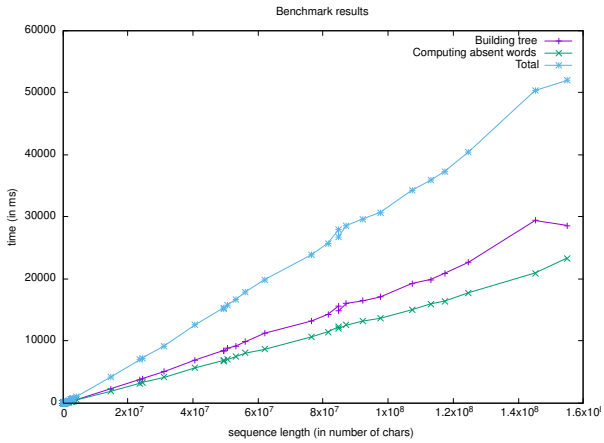
Figure 2a is obtained by running the command
`./main -f all_ebi_plasmids.fa -b`
and the figure 2b is obtained by running the command
`./main -f human_genome.fa -b`

B. Time per chromosomes

We compare our method and implementation against what we found to be the state of the art : MAW[1] (code can be found here : github.com/solonas13/maw).



(a) Sequence length versus elapsed time in the dataset *all_ebi_plasmids*



(b) Sequence length versus elapsed time in the human genome

Fig. 2: Benchmarks of two different order of magnitude

Table I and figure 3 show the same data in two differents manners. We can clearly see that our implementation takes much less times for each chromosomes

C. Memory consumption

Suffix trees are known to take a lot of memory. Our implementation remains reasonable as shown in figure 4. The space needed to build the suffix tree for the first assembled chromosome of the human genome does not exceed 1GB. When the tree is fully built, we can free some useless space, and thus the tree only takes around 700MB in RAM (see the drop at 5 seconds).

Figures 4 is the result of the command
`heaptrack ./main -f "ch1.fa" -t 1`
and the figure 5 is the result of the command
`heaptrack ./main -f "HumanGenome.fa" -t 1`.

The suffix-tree can be decomposed in two part : In orange, the space taken by the internal nodes, and in yellow the space taken by the leafs (note that the green and yellow are inversed in the figure 5).

The yellow space on the figure 5 is the one taken by the DNA sequences loaded in RAM. We can easily reduce this

Chromosome	Size(chars)	Ours(s)	MAW(s)
1	145,104,052	119	208
2	154,900,992	127	208
3	124,624,624	93	148
4	117,342,534	89	137
5	112,997,158	87	132
6	107,292,299	84	122
7	97,516,128	75	114
8	92,099,122	71	98
9	76,462,405	57	86
10	84,750,270	64	91
11	84,706,436	64	91
12	81,461,919	61	88
13	61,978,413	45	67
14	56,160,878	40	60
15	53,158,946	38	57
16	50,544,564	35	51
17	49,525,745	35	50
18	49,401,619	35	48
19	31,157,679	20	34
20	40,509,767	27	38
21	24,460,116	15	24
22	23,739,968	15	25
X	87,169,147	67	104
Y	14,664,429	8	22

TABLE I: Elapsed time comparison of our implementation and state-of-the-art MAW in the genome of *Homo sapiens*

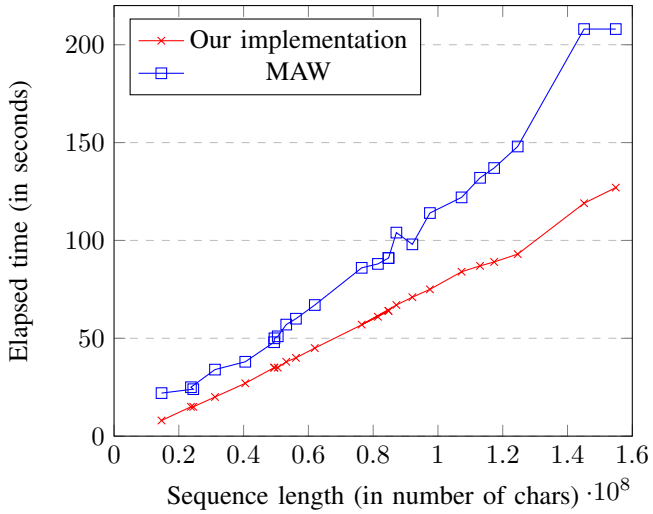


Fig. 3: Elapsed time versus sequence length

footprint by only loading the sequences when needed (with a small buffer to ensure no time wasted to wait the disk).

IV. Discussion

A. Why print it all in a file ?

The resulting file migth be very large, as the number of absent word is, we can represent MAWs much more efficiently, by just storing a pointer to the start of the absent word, the length of the absent word, and the one letter to add. Thus we might be able to significantly compress the output file.

B. The suffix automaton

We think that we could implement the same alike algorithm using a suffix automaton traversal with a double pointer, but we do not have testing yet either.

C. To improve

There are many possible improvments for our implementation, regarding RAM usage and time :

- Store nucleids on 2 bits (instead of 8 with a normal `char`)
- Only load sequences in memory when needed, instead of all at the beginning
- Use `mmap` to both read the input and write the output
- Improve our multi-threading scheme, to allow to parallelize the tree traversal
- Merge the suffix trees into generalized suffix trees to get MAWs for the whole set of sequences

V. Bibliography

References

[1] C. Barton, A. Heliou, L. Mouchard, and S. P. Pissis, *Linear-time Computation of Minimal Absent Words Using Suffix Array*. BMC Informatics, 2014, no. 15, ch. 188.

[2] E. Ukkonen, *On-line construction of suffix trees*. Algorithmica, 1995, no. 14.

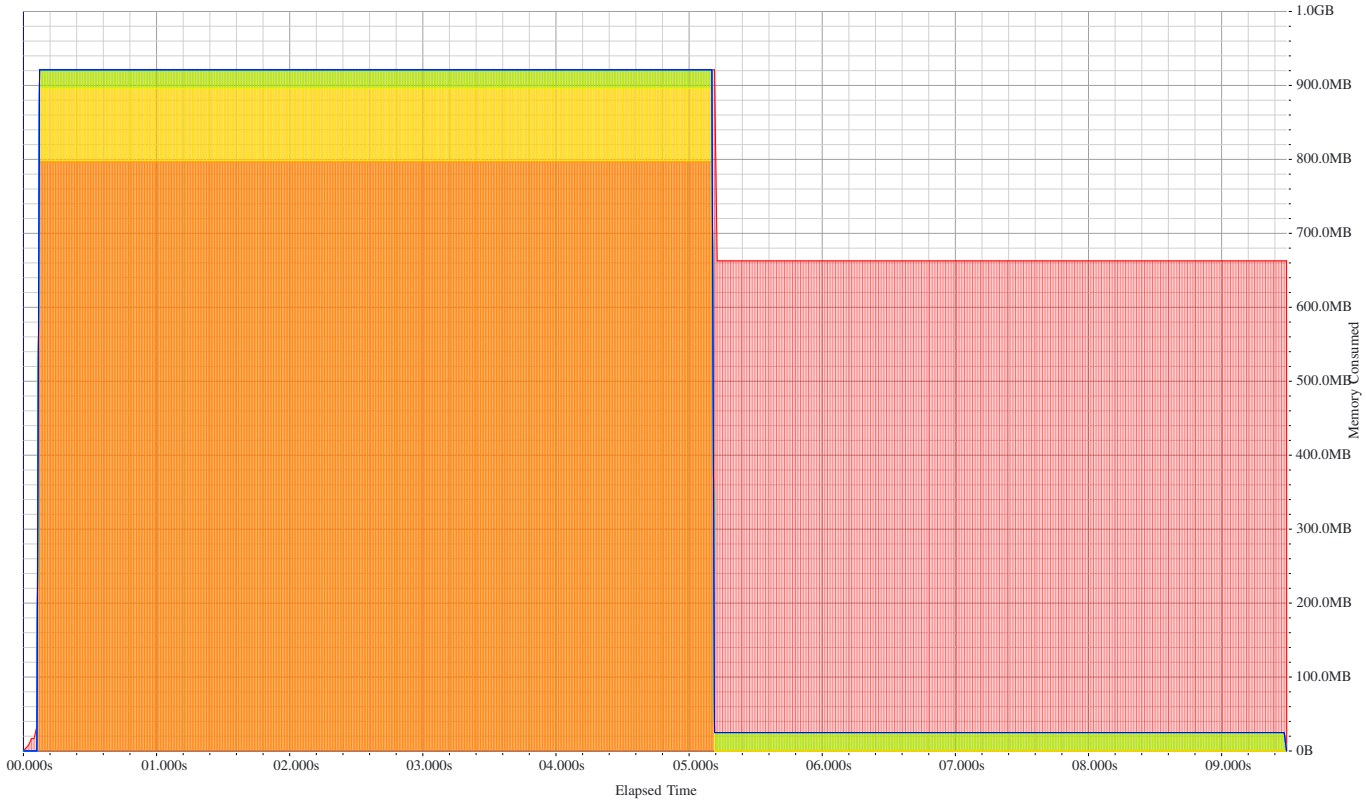


Fig. 4: RAM usage on the first assembled chromosome of the human genome

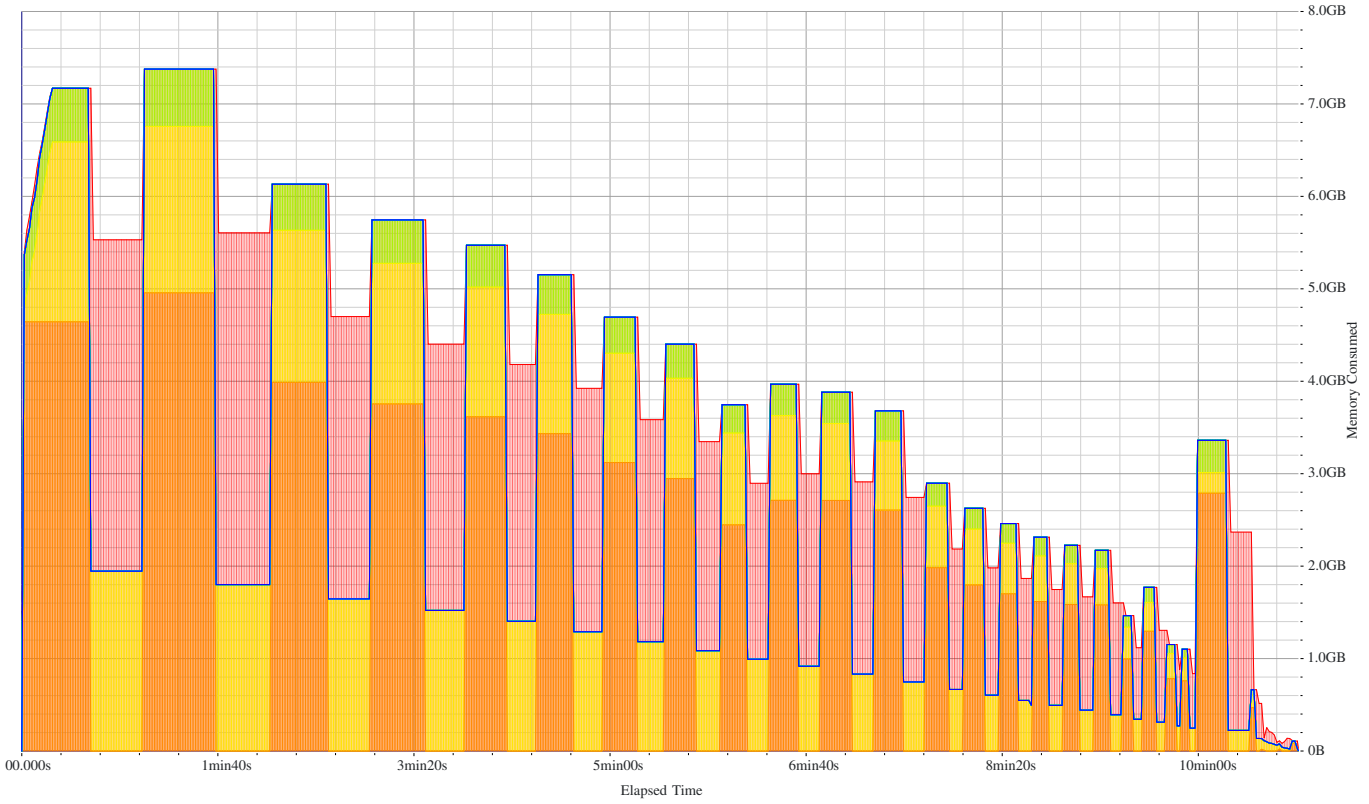


Fig. 5: RAM usage on the whole human genome