Faster Computation of Genome Mappability with One Mismatch

Sahar Hooshmand, Paniz Abedin, Daniel Gibney, Srinivas Aluru, Sharma V. Thankachan

1. University of Central Florida
2. Georgia Institute of Technology

October 18, 2018

Overview

Mappability - Definition

k-**Mappability problem:**

- **Input:** A sequence $S[1, n]$ of length n and two integers k and $m \leq n$
- **Output:** An integer array F_k s.t:

$$F_k[i] = |\ \{j \neq i \mid d_H(S[i, i+m-1], S[j, j+m-1]) \leq k\}\ |$$

- $d_H(\cdot, \cdot)$: Hamming Distance
- $S[i, i+m-1]$: The substrings of length m starting at position i

Mappability - Example

- **Input**: $S[1,8] = CCACAACA$ with $m = 3$, $k = 0$ or 1
- **Output:** Integer arrays F_0 and F_1:

Index	1	2	3	4	5	6	7	8
$S[1,8]$	C	C	A	C	A	A	C	A

Mappability - Example

- **Input**: $S[1,8] = CCACAACA$ with $m = 3$, $k = 0$ or 1
- **Output:** Integer arrays F_0 and F_1:

Index	1	2	3	4	5	6	7	8
$S[1,8]$	C	C	A	C	A	A	C	A

Position i	1	2	3	4	5	6
substring	CCA	CAC	ACA	CAA	AAC	ACA
$F_0[i]$	0	0	1	0	0	1
$F_1[i]$	3	2	2	2	1	2

Mappability - Example

- **Input**: $S[1,8] = CCACAACA$ with $m = 3$, $k = 0, 1$
- **Output:** Integer arrays F_0 and F_1:

Index	1	2	3	4	5	6	7	8
$S[1,8]$	C	C	A	C	A	A	C	A

Position i	1	2	**3**	4	5	6
substring	CCA	CAC	ACA	CAA	AAC	ACA
$F_0[i]$	0	0	1	0	0	1
$F_1[i]$	3	2	2	2	1	2

$F_0[3]$: ACA at index 6
$F_1[3]$: ACA at index 6, CCA at index 1

Mappability - Applications

- Derrien et al. : It is a measure of the approximate repeat structure of the genome with respect to substrings of specific length and a tolerance for mismatches.
- W. Li et al. : It can be used in Designing or interpreting high-throughput short read sequencing experiments
- A.Huda et al. : It can be used to quantify transcription counts in gene expression studies.

Mappability - Previous results

- 0-mappability problem: can be easily solved in linear time using the suffix tree data structure
- $k \geq 1$: Derrien *et al.* proposed a heuristic algorithm to approximate the solution.
- 1-mappability problem: Alzamel *et al.* proposed three linear space algorithms with run times as follows:
  1. An $O(n \log n \log \log n)$ algorithm.
  2. An $O(nm)$ time algorithm.
  3. An $O(n)$ average-case time algorithm for $m = \Omega(\log n)$.
- More recently Alzamel et al. provided a solution for k-mappability using $O(n \min(\log^{k+1} n, m^k))$ time and linear space
- Our result for 1-mappability problem:

  $O(n \log n)$ time and $O(n)$ space.

Our Algorithm Framework

- The algorithm consists of two phases:

1. In the first phase we construct data structures based on the suffix tree of the input string.
    - Side-tree (s-tree)
    - HeavyPath-tree (hp-tree)
2. In the second phase we traverse these data structures and gather the desired values for computing F_1 array.

Heavy Path Decomposition

- Start at the root, w, of the tree. We will consider w as a *light* node.
- Take w's child, v, which has the **largest subtree size** and add it to the heavy path. We will refer to the node v as w's *heavy* child.
- Continue adding nodes to the heavy path in this fashion until we reach a leaf.
- Recurse on each light node adjacent to the heavy path.

Heavy Path Decomposition - Example

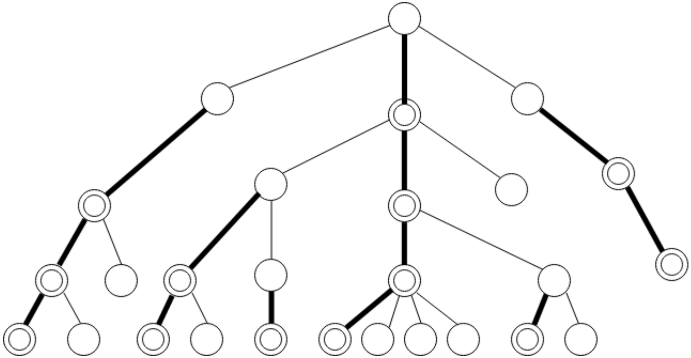


Figure: Nodes without double circles at the root of every heavy path are called light nodes. Double circles are called heavy nodes.

Heavy Path Decomposition - Key Observations

Observation

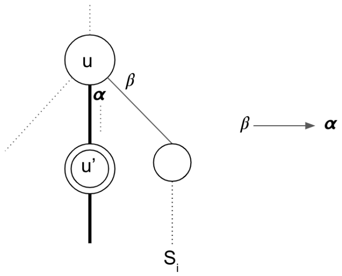
- *For a tree having n nodes, the path from the root to any leaf traverses at most $\lceil \log n \rceil$ light nodes.*
- *The sum of subtree sizes of all light nodes in a tree is $O(n \log n)$.*

Our Algorithm - Phase 1 Preliminaries

u: An internal node
u': u's heavy child
α: The leading character on the edge towards u'.
S_i: The suffix of S starting at position i

Definition

Modified Suffix S_i': is obtained from S_i under subtree of u after replacing its (strDepth(u) + 1)th character by α



Our Algorithm - Phase 1 Preliminaries

Suff(u): The set of suffixes corresponding to the leaves of subtree(u).
u: An internal node
u': u's heavy child

Definition

Side-Tree (s-Tree):
is a compact trie over all modified strings in

$$\text{Suff}'(u) = \{S'_i \mid S_i \in \text{Suff}(u) \setminus \text{Suff}(u')\}$$

s-Tree - Example

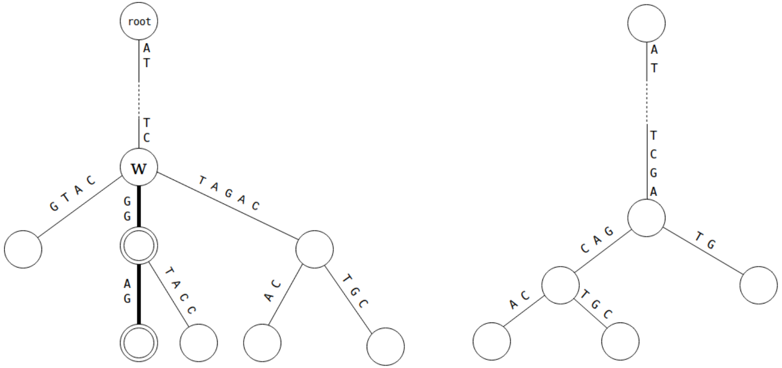
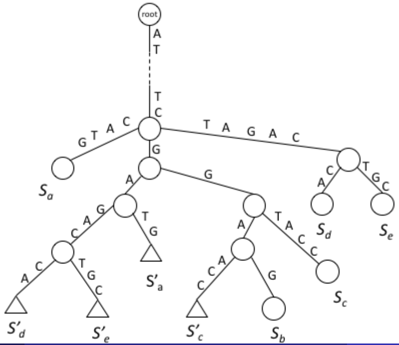


Figure: As an illustration, we show a portion of a suffix tree (on left) and the corresponding s-Tree(·) w.r.t. a light node w (on right).

hp-Tree

Definition

HeavyPath-tree (hp-Tree):
For each light node w , *hp-Tree(w)* is as a compact trie of *s-Trees* of all nodes on the heavy path rooted at w (modified suffixes) **and** Original suffixes corresponding to the leaves of subtree(w).



Our Algorithm - Phase 1

- **Input:** A sequence $S[1, n]$ of length n and two integers $k = 1$ and $m \leq n$

1. Perform a heavy path decomposition of the Suffix Tree of S.
2. Construct a s-Tree for every node u where $\text{strDepth}(u) < m$.
3. Construct a hp-Tree for every light node w where $\text{strDepth}(w) < m$.

Our Algorithm - Phase 2

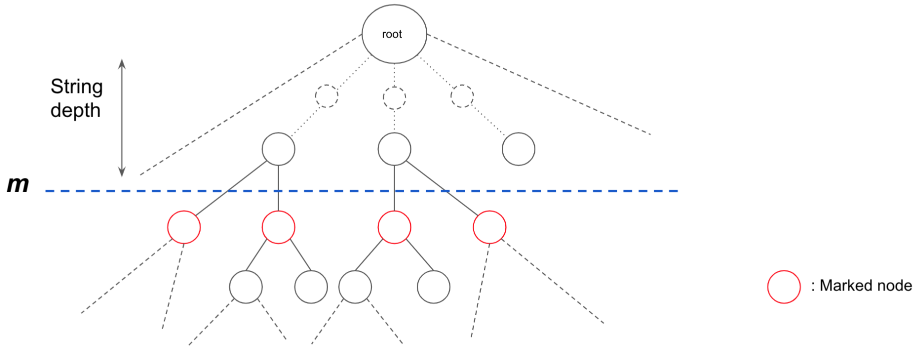
- There are 3 possibilities that can have an effect on the output array F_1 :
  - Two suffixes have already $|LCP| \geq m$
  - Two modified suffixes have $|LCP| \geq m$
  - One original suffix and one modified suffix have $|LCP| \geq m$

$|LCP|=$ Length of the longest common prefix

Phase 2: Step 1 - Processing s-Trees

Definition

A node v is marked iff $\text{strDepth}(\mathit{parent}(v)) < m \leq \text{strDepth}(v)$.



Phase 2: Step 1 - Processing s-Trees

- When $|LCP|$ of two modified suffixes $\geq m$: Scanning s-trees

```
Compute the array $F_0$
Initialize array $F_1$ to zero
for each node $u$ in ST with strDepth($u$) $< m$ do
    for every marked node $v$, in s-Tree($u$) do
        scan leaves of $v$'s subtree
        if leaf corresponds to a modified suffix $S'_i$ then
            $F_1[i]$ is incremented by (size($v$) - $F_0[i] - 1$).
        end if
    end for
end for
```

Phase 2: Step 2 - Processing hp-Trees

- When $|LCP|$ of one modified suffix and one original suffix $\geq m$: Scanning hp-trees

```
for each light node $w$ in ST with strDepth($w$) $< m$ do
    for every marked node $v$ in hp-Tree($w$) do
        Compute the number of modified suffixes $c'$ and the number of
        unmodified suffixes $c$.
        On a second scan of the same leaves:
        if a leaf corresponds to an unmodified suffix $S_i$ then
            increment $F_1[i]$ by $c'$
        else if a leaf corresponds to a modified suffix $S_i'$ then
            increment $F_1[i]$ by $c$.
        end if
    end for
end for
```

Phase 2: Step 3 - Remove double counted matches

- z_i : Marked node on the path towards the leaf corresponding to S_i.

```
for i from 1 to n-m+1 do
    if $z_i$ is a light node then
        increment $F_1[i]$ by $F_0[i]$,
    else
        decrement $F_1[i]$ by $F_0[i]$.
    end if
end for
```

Time and Space complexity

- Time complexity:
  - Phase 1, Constructing s-Trees and hp-Trees: Can be implemented in $O(n \log n)$ time.
  - Phase 2, Processing Trees: Runs in time proportional to the number of leaves in all of the hp-Trees and s-Trees combined. So, this phase also takes $O(n \log n)$.
- Space complexity:
  - Each phase can be maintained at $O(n)$.

Future Work: Solving K-mappability problem

We solved 1-mappability problem in $O(n \log n)$ time and $O(n)$ space.

Can we use the ideas presented here to get a $O(nlog^k n)$ solution for general $k \geq 1$?

References

[1] M. Alzamel, P. Charalampopoulos, C. S. Iliopoulos, S. P. Pissis, J. Radoszewski, and W.-K. Sung. Faster algorithms for 1-mappability of a sequence. In International Conference on Combinatorial Optimization and Applications, pages 109121. Springer, 2017.
[2] M. R. Brown and R. E. Tarjan. A fast merging algorithm. Journal of the ACM (JACM), 26(2):211226, 1979.
[3] R. Cole, L. Gottlieb, and M. Lewenstein. Dictionary matching and indexing with errors and dont cares. In Proceedings of the 36th Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, June 13-16, 2004, pages 91100, 2004.
[4] T. Derrien, J. Estelle , S. M. Sola, D. G. Knowles, E. Raineri, R. Guigo , and P. Ribeca. Fast computation and applications of genome mappability. PloS one, 7(1):e30377, 2012.
[5] M. Farach. Optimal suffix tree construction with large alphabets. In 38th Annual Symposium on Foundations of Computer Science, FOCS 97, Miami Beach, Florida, USA, October 19-22, 1997, pages 137143, 1997.

References

[6] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. SIAM J. Comput., 13(2):338355, 1984.
[7] A.Huda,L.Marin o-Ram rez,D.Landsman,andI.K.Jordan.Repetitive dna elements, nucleosome binding and human gene expression. Gene, 436(1):1222, 2009.
[8] W. Li and J. Freudenberg. Mappability and read length. Frontiers in genetics, 5:381, 2014.
[9] K. Sadakane. Compressed suffix trees with full functionality. Theory of Computing Systems, 41(4):589607, 2007.
[10] B. Schieber and U. Vishkin. On finding lowest common ancestors: Sim- plification and parallelization. SIAM Journal on Computing, 17(6):1253 1262, 1988.
[11] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. In Proceedings of the 13th Annual ACM Symposium on Theory of Computing, May 11-13, 1981, Milwaukee, Wisconsin, USA, pages 114 122, 1981.

References

[12] S. V. Thankachan, C. Aluru, S. P. Chockalingam, and S. Aluru. Algorithmic framework for approximate matching under bounded edits with applications to sequence analysis. In Research in Computational Molecular Biology - 22nd Annual International Conference, RECOMB 2018, Paris, France, April 21-24, 2018, Proceedings, pages 211224, 2018.
[13] S. V. Thankachan, A. Apostolico, and S. Aluru. A provably efficient algorithm for the k-mismatch average common substring problem. Journal of Computational Biology, 23(6):472482, 2016.
[14] E. Ukkonen. On-line construction of suffix trees. Algorithmica, 14(3):249260, 1995.
[15] P. Weiner. Linear Pattern Matching Algorithms. In SWAT, pages 111, 1973.

Thank you!