Efficient Construction of Hierarchical Overlap Graphs

Sung Park, Bastien Cazaux, Kunsoo Park, Eric Rivals

Efficient Construction of Hierarchical Overlap Graphs

Sung Gwan Park[1⋆], Bastien Cazaux[2,3], Kunsoo Park[1⋆(✉)], and Eric Rivals[3⋆⋆(✉)]

[1] Seoul National University, Korea
sgpark@theory.snu.ac.kr, kpark@theory.snu.ac.kr
[2] University of Helsinki, Finland
bastien.cazaux@lirmm.fr
[3] LIRMM, Univ Montpellier, CNRS, France
rivals@lirmm.fr

Abstract. The hierarchical overlap graph (HOG for short) is an overlap encoding graph that efficiently represents overlaps from a given set P of n strings. A previously known algorithm constructs the HOG in $O(||P|| + n^2)$ time and $O(||P|| + n \times \min(n, \max\{|s| : s \in P\}))$ space, where $||P||$ is the sum of lengths of the n strings in P. We present a new algorithm of $O(||P|| \log n)$ time and $O(||P||)$ space to compute the HOG, which exploits the segment tree data structure. We also propose an alternative algorithm using $O(||P|| \frac{\log n}{\log \log n})$ time and $O(||P||)$ space in the word RAM model of computation.

Keywords: hierarchical overlap graph, segment tree, word RAM model

1 Introduction

Genome sequencing is limited by sequencing technologies that yield sequencing reads which are orders of magnitude shorter than the entire genome. Hence, obtaining a whole genome sequence from sequencing reads resorts to *DNA assembly*. This problem consists in recovering the target sequence from the overlaps of reads by inferring their order and relative positions in the target sequence. It translates into seeking a maximal path in a graph that encodes suffix-prefix overlaps between pairs of reads [7,22,25,26]. The development of DNA sequencing goes along with several proposals of overlap encoding graphs, usually classified into two categories of digraphs:

– *Overlap Graph* [25] and its variants (like String Graph [22]), in which each input read is a node and an arc connecting a pair of reads represents the longest overlap between them, and

⋆ Supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government (MSIT) (No. 2018-0-00551, Framework of Practical Algorithms for NP-hard Graph Problems).
⋆⋆ ER thanks funding Labex NUMEV, GEM project (ANR 2011-LABX-076).

– "assembly" *de Bruijn Graph* [26], in which for a length k, each node represents a k-long substring (termed k-mer) and an arc connects two k-mers whenever the suffix of one matches the prefix of the other over length $k-1$.

The overlap relation is not symmetrical, which explains why directed, rather than undirected, graphs should be used in DNA assembly. Moreover, a pair of reads can have several overlaps (in the same direction), in which case a shorter overlap is necessarily nested into a longer one.

Recently, Cazaux and Rivals [9,8] proposed an alternative graph in which the input reads and substrings corresponding to suffix-prefix overlaps are nodes in the graph. This digraph, called *Extended Hierarchical Overlap Graph* (EHOG), encodes both the longest suffix relationship and the longest prefix relationship between nodes by using two kinds of arcs. To compact the EHOG even more, the *Hierarchical Overlap Graph* (HOG) which includes only maximal overlaps between reads was defined. A maximal overlap is a longest overlap for at least one pair of reads. By definition, therefore, the HOG is a subgraph of the EHOG. See Figure 1 for examples of EHOG and HOG. Even if the EHOG and the HOG can be identical for some instances, the ratio of the EHOG size over the HOG size (in the number of nodes) can tend to infinity for some families of instances [9]. Thus, efficient algorithms to build the HOG are important from both practical and theoretical viewpoints. The advantages of the HOG/EHOG for storing overlaps compared to other graphs are discussed in [9].

Given a set of strings, the *shortest superstring* problem is the problem of finding a shortest superstring of the given strings. The shortest superstring problem has applications in DNA assembly and data compression [6,29]. Since the problem is MAX SNP-hard, there has been extensive research to get better approximation ratios, e.g., 3 in [6], $2\frac{2}{3}$ in [3], $2\frac{1}{2}$ in [29], and more recently $2\frac{11}{23}$ [21] and $2\frac{11}{30}$ [23]. These approximation algorithms are based on the overlap graph (or equivalent *distance graph*). In the overlap graph (or the distance graph), many distinct arcs may encode the same overlap, but this fact is not specified in the graph. In the HOG, all identical overlaps are encoded into a unique node, i.e., this fact is specified. Hence, the HOG has structurally more information than the overlap graph, and thus it has a great potential in studying DNA assembly and the shortest superstring problem.

Suppose that an input instance P consists of n strings, where no string is a substring of another. The norm of P, denoted by $||P||$, is defined as the sum of lengths of the strings in P. Computing an overlap graph from P is equivalent to solving the all-pair suffix-prefix problem, which is studied extensively [12,14,20,27]. The best asymptotic bound for this problem is $O(||P|| + n^2)$ [14], which is optimal. Computing the EHOG from P takes linear time in the norm of P [9]. However, further limiting the set of overlap nodes to maximal overlaps, which enables us to build the HOG, is more challenging. A previously known algorithm achieves $O(||P|| + n^2)$ time with $O(||P|| + n \times \min(n, \max\{|s| : s \in P\}))$ space [9], which has the same time complexity as the all-pair suffix-prefix problem. The question of an optimal algorithm for computing the HOG remains open. In this paper we present an algorithm taking $O(||P|| \log n)$ time with $O(||P||)$

space in the standard RAM model, which exploits the segment tree data structure (Section 3). We also propose an alternative algorithm using $O(||P|| \frac{\log n}{\log \log n})$ time and $O(||P||)$ space in the word RAM model of computation [15] (Section 4). Throughout the paper, we assume that the size of the alphabet is constant.

2 Preliminaries

In this paper we consider strings over a finite alphabet Σ. Given a string s, the length of s is denoted by $|s|$. For any two integers $1 \leq i \leq j \leq |s|$, the substring of s which starts from i and ends at j is denoted by $s[i..j]$. Substring $s[i..j]$ is a prefix of s if $i = 1$, and a suffix of s if $j = |s|$. A prefix (suffix) of s is a proper prefix (suffix) of s if it is different from s. Given two strings s and t, string u is an overlap from s to t if u is a proper suffix of s and also a proper prefix of t. The longest overlap from s to t is denoted by $ov(s, t)$. Given a set $P = \{s_1, s_2, ..., s_n\}$ of strings, the sum of $|s_i|$'s is denoted by $||P||$.

2.1 Hierarchical overlap graph

We use definitions of *extended hierarchical overlap graph* and *hierarchical overlap graph* in [9].

Definition 1. Given a set $P = \{s_1, s_2, \ldots, s_n\}$ of strings, let $Ov^+(P)$ be the set of all overlaps from s_i to s_j for $1 \leq i, j \leq n$. The *Extended Hierarchical Overlap Graph* of P, denoted by EHOG(P), is a directed graph (V^+, E^+) where $V^+ = P \cup Ov^+(P) \cup \{\epsilon\}$ and $E^+ = E_1^+ \cup E_2^+$, where $E_1^+ = \{(x, y) \in V^+ \times V^+ \mid x$ is the longest proper prefix of $y\}$ and $E_2^+ = \{(x, y) \in V^+ \times V^+ \mid y$ is the longest proper suffix of $x\}$.

Definition 2. Given a set $P = \{s_1, s_2, \ldots, s_n\}$ of strings, let $Ov(P)$ be the set of the *longest* overlap from s_i to s_j for $1 \leq i, j \leq n$. The *Hierarchical Overlap Graph* of P, denoted by HOG(P), is a directed graph (V, E) where $V = P \cup Ov(P) \cup \{\epsilon\}$ and $E = E_1 \cup E_2$, where $E_1 = \{(x, y) \in V \times V \mid x$ is the longest proper prefix of $y\}$ and $E_2 = \{(x, y) \in V \times V \mid y$ is the longest proper suffix of $x\}$.

For example, Figure 1 from [9] shows an Aho-Corasick trie [1], EHOG, and HOG built with $P = \{aabaa, aacd, cdb\}$. Note that EHOG is a contracted form of the Aho-corasick trie and HOG is a contracted form of EHOG, as described in [9]. Consequently, both EHOG and HOG, without failure links, are trees.

By definitions of EHOG and HOG, each node $\mathtt{u}$ in a graph represents a string, which is the concatenation of labels on the path from the root to $\mathtt{u}$. If $(\mathtt{u}, \mathtt{v})$ is a tree arc (an edge in E_1^+ or E_1, solid line in Figure 1) in an EHOG (resp. HOG), the string represented by $\mathtt{u}$ is the longest proper prefix of the string represented by $\mathtt{v}$ in the EHOG (resp. HOG). If $(\mathtt{u}, \mathtt{v})$ is a failure link (an edge in E_2^+ or E_2, dotted line in Figure 1) in an EHOG (resp. HOG), the string represented by $\mathtt{v}$ is the longest proper suffix of the string represented by $\mathtt{u}$ in the EHOG

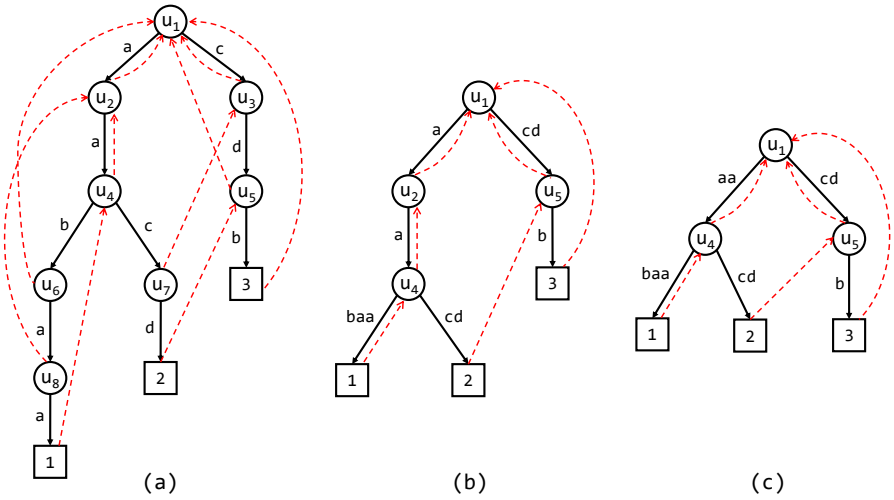

Fig. 1. Data structures built with $P = \{aabaa, aacd, cdb\}$. Dotted lines represent failure links of the nodes. (a) Aho-Corasick trie. (b) Extended hierarchical overlap graph. (c) Hierarchical overlap graph.

(resp. HOG). In this paper we use term 'node' to mean a node in EHOG or HOG, or a string represented by the node.

We can build an EHOG of $P = \{s_1, s_2, ..., s_n\}$ in $O(||P||)$ time and space [9]. Furthermore, if we know EHOG(P) and $Ov(P)$, we can compute HOG(P) in $O(||P||)$ time and space [9]. Therefore, the bottleneck of computing HOG(P) is to compute $Ov(P)$, which costs $O(||P||+n^2)$ time and $O(||P||+n\times\min(n, \max\{|s_i|\}))$ space in [9].

3 Main Algorithm

In this section we describe an algorithm to compute HOG from the given set $P = \{s_1, s_2, \ldots, s_n\}$ of strings in $O(||P|| \log n)$ time.

3.1 New approach to compute HOG

First, we build an Aho-Corasick trie of P and renumber the strings (i.e., leaves) in lexicographic order. This can be done in $O(||P||)$ time, assuming that the size of the alphabet is constant. Next, we build EHOG(P) in $O(||P||)$ time [9]. Furthermore, for each node `u` in EHOG(P), we define an interval `I(u)` that contains every leaf node that is in the subtree of `u` (i.e. `I(u)` $= \{i \in [1..n] \mid$ `u` is a prefix of $s_i\}$). Since P is renumbered in lexicographic order, we can see that `I(u)` forms one interval.

Algorithm 1 Computing HOG using interval encoding

```
1: procedure BUILD-HOG-INTERVAL-ENCODING(EHOG(P))
2:     for i ← 1 to n do
3:         Initialize B[1..n] to false
4:         u ← leaf corresponding to s_i in EHOG(P)
5:         Mark u as included in HOG(P)
6:         while u ≠ root do
7:             u ← failure link of u in EHOG(P)
8:             if ∃ j ∈ I(u) such that B[j] is false then
9:                 Mark u as included in HOG(P)
10:            for j ∈ I(u) do
11:                B[j] ← true
12:    Build HOG(P) with marked nodes
```

Given EHOG(P), we compute $Ov(P)$ by discarding nodes that are not longest overlaps. If a string s is included in $Ov(P)$, s is a proper suffix of s_i and a proper prefix of s_j for some i and j by definition of $Ov(P)$. To compute all longest overlaps from s_i, we start from the i-th leaf s_i, follow the failure links repeatedly up to the root, and check whether the node we are looking at is the longest prefix of s_j for some j. (Note that every overlap between two strings in P is represented as a node in EHOG(P), and thus we can iterate through all overlaps from s_i by following the failure links starting from s_i.) While traversing the nodes through failure links (namely $\mathtt{v_0}$ = i-th leaf $\rightarrow \mathtt{v_1} \rightarrow \cdots \rightarrow \mathtt{v_k} = \mathtt{root}$), $\mathtt{v_x}$ $(1 \leq x \leq k)$ is $ov(i, j)$ if and only if $\mathtt{v_x}$ is the first node that is a prefix of s_j during the traversal. More specifically, $\mathtt{v_x}$ should be the prefix of s_j and $\mathtt{v_y}$'s $(1 \leq y < x)$ should not be the prefixes of s_j. To check whether there exists such j efficiently, we maintain a bit vector B of length n defined as follows. At the end of the iteration with $\mathtt{v_x}$ $(1 \leq x \leq k)$, $B[j] = \mathtt{true}$ if and only if there exists $1 \leq y \leq x$ such that $\mathtt{v_y}$ is a prefix of s_j. We can maintain B as defined by marking $B[j]$ for every $j \in \mathtt{I}(\mathtt{v_x})$ as $\mathtt{true}$ during the iteration with $\mathtt{v_x}$. Note that $\mathtt{v_0}$ is always included in HOG(P) by definition and is not considered.

We can check whether $\mathtt{v_x}$ should be included in HOG(P) by using B. Suppose that there exists j such that $B[j] = \mathtt{false}$ and $j \in \mathtt{I}(\mathtt{v_x})$ at the beginning of the iteration with $\mathtt{v_x}$. By the definition of $B[j]$ and $\mathtt{I}(\mathtt{v_x})$, $\mathtt{v_x} = ov(i, j)$ and it should be included in HOG(P). On the other hand, if $B[j] = \mathtt{true}$ at the beginning of the iteration with $\mathtt{v_x}$, there exists a longer overlap from s_i to s_j than $\mathtt{v_x}$ and it should not be included in HOG(P). If we do this process for every leaf node, we can get the list of nodes that we should include in HOG(P). Algorithm 1 describes an algorithm to compute HOG(P).

For example, let's consider the example in Figure 1(b). First, we consider the case with $i = 1$ in line 2. After we mark leaf 1 to be included in HOG(P) in line 5, we begin the loop with $\mathtt{u} = \mathtt{u_4}$, which is the failure link of leaf 1. We consider $\mathtt{I}(\mathtt{u_4}) = \{1, 2\}$ in array B. Since $B[1]$ and $B[2]$ are $\mathtt{false}$, we mark $\mathtt{u_4}$ to be included in HOG(P) and set $B[1]$ and $B[2]$ as $\mathtt{true}$. We continue the loop with $\mathtt{u} = \mathtt{u_2}$ by following the failure link. Since there is no $j \in \mathtt{I}(\mathtt{u_2}) = \{1, 2\}$

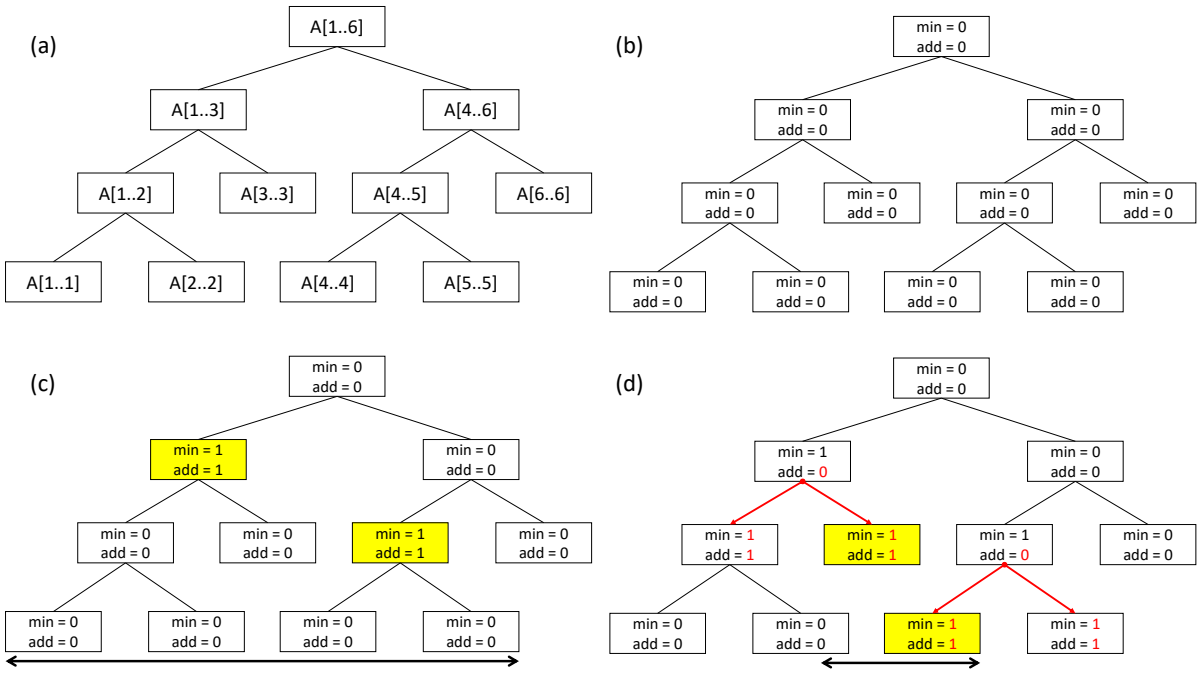

Fig. 2. Segment tree structure with $n = 6$. (a) The intervals that each node represents. (b) The values `min` and `add` that each node initially stores. (c) The values that each node stores after query 2 on $A[1..5]$. (d) The values that each node stores after query 1 on $A[3..4]$. Red arrows show that `add` values of the nodes are propagated to `min` and `add` values of their children.

such that $B[j]$ is `false`, we don't include $\mathtt{u}_2$ in HOG(P). We continue the loop with $\mathtt{u} = \mathtt{u}_1$. We consider $\mathtt{I}(\mathtt{u}_1) = \{1, 2, 3\}$ in array B. Since $B[3]$ is `false`, we mark $\mathtt{u}_1$ to be included in HOG(P) and set $B[3]$ as `true`. Since $\mathtt{u} = \mathtt{u}_1$ is the root, we finish the loop.

3.2 Improvement using segment tree

To speed up Algorithm 1, we have to process these two types of queries efficiently.

i) Given an interval $[a..b]$, check whether there is any index $j \in [a..b]$ such that $B[j] =$ `false` (Lines 8-9).
ii) Given an interval $[a..b]$, set $B[j]$ as `true` for every $j \in [a..b]$ (Lines 10-11).

In order to process these queries, let's consider the following two types of queries on an integer array A. For an index j, $A[j] > 0$ means that $B[j] =$ `true`, while $A[j] = 0$ means that $B[j] =$ `false`.

1. Given an interval $[a..b]$, compute the minimum value among $A[a..b]$ (and check whether it is zero or not).

Algorithm 2 Computing minimum of an interval using segment tree

```
1: procedure SegTree-Min(cnode, cinterval)
2:     if cnode.int ⊆ cinterval then
3:         return cnode.min
4:     if cnode.int ∩ cinterval = ∅ then
5:         return ∞
6:     left, right ← two children of cnode
7:     left.min += cnode.add, left.add += cnode.add
8:     right.min += cnode.add, right.add += cnode.add
9:     cnode.add = 0
10:    return min(SegTree-Min(left, cinterval), SegTree-Min(right,
   cinterval))
```

Algorithm 3 Add 1 to an interval using segment tree

```
1: procedure SegTree-Update(cnode, cinterval)
2:     if cnode.int ⊆ cinterval then
3:         cnode.min += 1, cnode.add += 1
4:         return
5:     if cnode.int ∩ cinterval = ∅ then
6:         return
7:     left, right ← two children of cnode
8:     left.min += cnode.add, left.add += cnode.add
9:     right.min += cnode.add, right.add += cnode.add
10:    cnode.add = 0
11:    SegTree-Update(left, cinterval)
12:    SegTree-Update(right, cinterval)
13:    cnode.min = min(left.min, right.min)
14:    return
```

2. Given an interval $[a..b]$, add 1 to each element of $A[a..b]$.

We can see that one could use queries 1 and 2 to solve queries i and ii, respectively.

Let A be an integer array of length n. We use the segment tree data structure [5] to process queries 1 and 2 on A. The segment tree is a binary tree, which has n leaf nodes (they are 1, 2, ..., n) and has $O(\log n)$ height. Each leaf node represents one element, and each internal node represents an interval of elements. Figure 2(a) shows a segment tree for $n = 6$. For each node `u` in the segment tree, we define `u.int` as the interval that `u` represents. In Figure 2(a), for instance, `u.int` for the root node is [1..6].

While processing the queries, each node `u` stores both the minimum value among the elements in `u.int` (denoted by `u.min`) and an added value to `u.int` (denoted by `u.add`). Since A should be initialized to zero, every value in the segment tree is also initialized to zero. Figure 2(b) shows an initial state of the segment tree.

Algorithms 2 and 3 show the algorithms to perform queries 1 and 2, respectively, in the segment tree, which use the *lazy propagation* technique in [19], though in [19] one computes the sum, while here we compute the minimum. If query 1 occurs, we follow the nodes recursively from top to down, starting from the root. Consider a node `u` during the recursion. If `u.int` is included in the query interval, we return `u.min`. If `u.int` is disjoint with the query interval, we return ∞ to indicate that there are no values to be considered in `u.int`. Otherwise, we propagate an added value to the child nodes, continue the process with the child nodes and return the minimum among them. Query 2 can be done in a similar way, but in this case we have to recompute the minimum value of a node after updating its child nodes, as shown in line 13 of Algorithm 3.

Figures 2(c) and 2(d) show an example of processing two queries, query 2 on $A[1..5]$ and query 1 on $A[3..4]$. In Figure 2(c), we can see that two nodes representing $A[1..3]$ and $A[4..5]$ are updated in the segment tree. Note that `min` and `add` values of the descendant nodes are not updated yet. In Figure 2(d), we access the two nodes representing $A[3..3]$ and $A[4..4]$ to compute the minimum value among $A[3..4]$. Note that `add` values in $A[1..3]$ and $A[4..5]$ are propagated to their children to ensure that appropriate `min` values are stored in $A[3..3]$ and $A[4..4]$.

We now prove the correctness of Algorithms 2 and 3. To the best of our knowledge, this is the first correctness proof for the folklore lazy propagation technique in [19]. The proof is non-trivial because Algorithms 2 and 3 work together, but their recursive structures differ. First, we need an invariant that holds for both algorithms, i.e., Invariant 1 below. Moreover, since Algorithm 2 makes recursive calls at the end, we need a top-down sub-invariant for Algorithm 2. In contrast, Algorithm 3 makes recursive calls in the middle, and thus we have to come up with a bottom-up sub-invariant for Algorithm 3.

Each node `u` in the segment tree maintains the following invariant while processing queries 1 and 2.

$$\min_{i \in \texttt{u.int}} A[i] = \texttt{u.min} + \sum_{\texttt{v}} \{\texttt{v.add} : \texttt{v} \text{ is an ancestor of } \texttt{u}\}, \tag{1}$$

where A is the conceptual array in the definitions of queries 1 and 2, and `u` is not an ancestor of itself.

Lemma 1. Invariant 1 holds after Algorithm 2 or 3 is called with `cnode = root` and `cinterval` $= [a..b]$ for query 1 or 2, respectively.

Proof. We prove the lemma by induction. Initially, Invariant 1 holds because $A[i] = 0$ for every index i, and `u.min` $= 0$ and `u.add` $= 0$ for every node `u` in the segment tree.

First we show that Invariant 1 holds after Algorithm 2 is called for query 1. The left-hand side (LHS) of Invariant 1 is unchanged since Algorithm 2 performs a query on A, but does not change it. However, the propagation of the `add` values in the segment tree may update the `min` and `add` values of other nodes in it. So we must prove that the right-hand side (RHS) of Invariant 1 remains the same

too. When Algorithm 2 is called with `cnode = root`, it recurses through nodes in the segment tree (i.e., it goes down) until it reaches the base cases of recursion (which are handled in lines 2 and 4), and then it goes up by computing minima (in line 10). When Algorithm 2 goes down, we will show inductively that the RHS of Invariant 1 remains the same for every node in the segment tree after each execution of lines 6-9 (i.e., top-down sub-invariant for Algorithm 2). Consider one execution of lines 6-9. Since `left`, `right`, and `cnode` have their `min` and `add` changed, we show that the RHS of Invariant 1 remains the same for every node `u` in the subtree rooted at `cnode`.

- If `u = cnode`, `cnode.min` is not changed, and so the RHS of Invariant 1 remains the same.
- If `u = left` (similarly for `u = right`), `left.min` is increased as much as `cnode.add` is decreased, so the RHS of Invariant 1 remains the same.
- If `u` is a descendant of `left` (similarly for a descendant of `right`), `left.add` is increased as much as `cnode.add` is decreased. Since both `left` and `cnode` are `u`'s ancestors, the RHS of Invariant 1 remains the same.

Therefore, the RHS of Invariant 1 remains the same for every node `u` in the segment tree when Algorithm 2 goes down.

When Algorithm 2 goes up (including the base cases of recursion), the RHS of Invariant 1 does not change for any node in the segment tree. Therefore, Invariant 1 holds after Algorithm 2 is called for query 1.

Now we show that Invariant 1 holds after Algorithm 3 is called for query 2. When Algorithm 3 is called with `cnode = root`, it goes down by recursion and then it goes up, like Algorithm 2. When Algorithm 3 goes down, one can show inductively that the RHS of Invariant 1 does not change after each execution of lines 7-10, in a way similar to Algorithm 2.

When Algorithm 3 goes up, we will show inductively that Invariant 1 holds for every node in the subtree rooted at `cnode` at the moment when SegTree-Update (`cnode`, `cinterval`) returns (i.e., bottom-up sub-invariant for Algorithm 3). We first consider two base cases which are handled in lines 2 and 5.

- If $\texttt{cnode.int} \subseteq \texttt{cinterval}$, SegTree-Update(`cnode`, `cinterval`) performs line 3 and returns in line 4. After line 3 is done, the RHS of Invariant 1 for `cnode` and its descendants increases by 1. Since every $A[i]$ for $i \in \texttt{cnode.int}$ increases by 1, the LHS of Invariant 1 for them also increases by 1 and Invariant 1 holds.
- If $\texttt{cnode.int} \cap \texttt{cinterval} = \emptyset$, SegTree-Update(`cnode`, `cinterval`) does nothing and returns in line 6, and thus the RHS of Invariant 1 remains the same for `cnode` and its descendants. Since every $A[i]$ for $i \in \texttt{cnode.int}$ remains the same, Invariant 1 holds.

Next, we consider the induction step, where we assume that Invariant 1 holds for `left`, `right` and their descendants by the bottom-up sub-invariant. Now we need to show that Invariant 1 holds for `cnode` when SegTree-Update(`cnode`, `cinterval`) executes line 13 and returns. Suppose that $\texttt{left.min} \leq \texttt{right.min}$

Algorithm 4 Algorithm to compute HOG in $O(||P|| \log n)$ time

```
1: procedure BUILD-HOG(EHOG(P))
2:     for i ← 1 to n do
3:         Initialize the segment tree
4:         u ← leaf corresponding to s_i in EHOG(P)
5:         Mark u as included in HOG(P)
6:         while u ≠ root do
7:             u ← failure link of u in EHOG(P)
8:             if SEGTREE-MIN(root, I(u)) = 0 then
9:                 Mark u as included in HOG(P)
10:            SEGTREE-UPDATE(root, I(u))
11:    Build HOG(P) with marked nodes
```

(similarly for the case `left.min` > `right.min`). Consider Invariant 1 for `left` and `right`. Since `left` and `right` share the same ancestors, the summation parts of Invariant 1 for `left` and `right` are the same. So if `left.min` ≤ `right.min`, $\min_{i \in \texttt{left.int}} A[i] \leq \min_{i \in \texttt{right.int}} A[i]$ holds. Since `cnode.int` = `left.int` ∪ `right.int`, the LHS of Invariant 1 for `cnode` is the same as that of `left`. The RHS of Invariant 1 for `cnode` is also the same as that of `left` because `cnode.min` = `left.min` by line 13 and `cnode.add` = 0 by line 10.

Therefore, Invariant 1 holds for every node in the segment tree after Algorithm 3 is called with `cnode = root`.

Using Lemma 1, we can show the correctness of Algorithms 2 and 3 to solve queries 1 and 2.

Theorem 1. For any sequences of Algorithms 2 and 3 called with `cnode = root` and `cinterval` = $[a..b]$, Algorithm 2 (i.e., SEGTREE-MIN(`root`, `cinterval`)) returns a correct answer for query 1 with the given interval $[a..b]$.

Proof. By Lemma 1 Invariant 1 holds after every call on Algorithm 2 or 3. Furthermore, if we access node `u` by recursion in Algorithm 2, `v.add` = 0 for every ancestor `v` of `u` due to line 9 in Algorithm 2. Therefore, at the moment we access `u`, $\min_{i \in \texttt{u.int}} A[i] = \texttt{u.min}$ always holds from Invariant 1.

Since Algorithm 2 computes the minimum of `u.min` for every `u` whose interval is included in the given interval $[a..b]$, it is equal to the minimum value among $A[a..b]$. Therefore, Algorithm 2 returns a correct answer for query 1.

Given the EHOG, Algorithm 4 describes how to compute the HOG using queries on the segment tree data structure. Algorithm 4 is almost identical to Algorithm 1. First, the condition ($\exists j \in$ `I(u)` such that $B[j]$ is `false`) on line 8 of Algorithm 1 is now performed by (SEGTREE-MIN(`root`, `I(u)`) = 0) on line 8 of Algorithm 4. Second, the update **for** loop of lines 10-11 in Algorithm 1 is performed using a single query on line 10 of Algorithm 4: SEGTREE-UPDATE(`root`, `I(u)`).

Since any interval $[a..b]$ can be represented by $O(\log n)$ nodes with a segment tree [5], Algorithms 2 and 3 can be done in $O(\log n)$ time. By using them, we can get an $O(||P|| \log n)$ time algorithm to compute $\mathrm{HOG}(P)$, as shown in Algorithm 4. Since $\mathrm{HOG}(P)$ and the segment tree take $O(||P||)$ and $O(n)$ space, respectively, the space complexity of building the HOG is $O(||P||)$.

4 Improvement using the word RAM model

By using the word RAM model of computation [15] with w-bit machine words, where $w \geq \log n$, we show that we can compute the HOG from the given set P of n strings in $O(\|P\| \frac{\log n}{\log \log n})$ time.

Indeed, by using bitwise operations, we can improve queries 1 and 2 from $O(\log n)$ to $O(\log_w n) = O(\log_{\log n} n) = O(\frac{\log n}{\log \log n})$. To do so, we introduce the w-segment tree, which is the w-ary version of the segment tree as in [2,11].

4.1 Algorithms with bitwise operations

Unlike the original segment tree which is a binary tree, we define the w-segment tree as a tree with n leaves, a height of $O(\log_w n)$, and each node having at most w children. As in the segment tree, each internal node represents an interval of elements of P (i.e., $1, 2, \ldots, n$), and each leaf contains a single element (the interval of a node `u` is denoted by `u.int`). But, instead of storing for a node `u` the minimum value `u.min` and the added value `u.add`, we store two bit vectors of length w (`v.Vmin` and `v.Vadd`) for every internal node `v`. If a node `u` is the j-th child of its parent `p`, the j-th value of `p.Vmin` is `true` if `u.min` ≥ 1; `false` if `u.min` $= 0$ (same for `Vadd`).

To compute query 1 for a node `u` and an interval $[a, b]$, we begin by comparing the interval $[L, R] =$ `u.int` with $[a, b]$:

- If $[L, R] \subseteq [a, b]$, we return the j-th bit of `p.Vmin`, where `u` is the j-th child of its parent `p`.
- If $[L, R] \cap [a, b] = \emptyset$, we return `true`.
- Otherwise, we compute the positions i_a and i_b corresponding to a and b in $[0, w-1]$:

  $$i_a = \lfloor \tfrac{(a-L)w}{R-L+1} \rfloor \text{ and } i_b = \lfloor \tfrac{(b-L)w}{R-L+1} \rfloor.$$

  If the j-th position of `p.Vadd` is equal to 1, all the values of `u.Vmin` and `u.Vadd` become 1, and the j-th position of `p.Vadd` becomes 0. At the end, we recursively call the function on $\texttt{Child}_{i_a}$ and $\texttt{Child}_{i_b}$ and return the minimum of two recursive calls and the values of `u.Vmin` between positions $i_a + 1$ and $i_b - 1$, where the minimum of the corresponding values of `u.Vmin` is computed as the following Boolean value:

  $$(\texttt{u.Vmin AND } (2^{i_b} - 2^{i_a+1})) = (2^{i_b} - 2^{i_a+1}).$$

In a similar way, we can compute query 2 by using bitwise operations.

4.2 Using a table for a compressed space

Instead of a tree structure, we can use two tables to simulate the segment tree. Let

$$h = \frac{w^{\lceil \log_w n \rceil - 1} - 1}{w - 1} + \left\lceil \frac{n}{w} \right\rceil$$

denote the size of these tables, and let `Tmin`$[0..h-1]$ and `Tadd`$[0..h-1]$ be two tables of w-bit words initialized to $[\mathbf{0}, \ldots, \mathbf{0}]$. We store `Vmin`'s and `Vadd`'s of Section 4.1 into `Tmin` and `Tadd`, respectively, in the BFS order of the w-segment tree (i.e., top to bottom, left to right) and run the algorithm described in Section 4.1 (see Algorithm 5). In the same way, we can build the algorithm corresponding to query 2 with bitwise operations.

Algorithm 5 Computing minimum of an interval using w-segment tree

procedure SegTreeMinRam($k, [a, b]$)
  $d \leftarrow \lfloor \log_w((w-1)k + 1) \rfloor$ ▷ Depth of node k
  $x \leftarrow k - \frac{w^d - 1}{w-1}$ ▷ Node k is the x-th node with depth d
  $Y \leftarrow w^{\lceil \log_w n \rceil - d}$ ▷ Node k represents an interval of length Y
  $L \leftarrow xY + 1$
  $R \leftarrow (x+1)Y$
  $p \leftarrow \lfloor \frac{k-1}{w} \rfloor$ ▷ p is parent of node k
  $j \leftarrow (k-1) \bmod w$ ▷ Node k is the j-th child of p
  $i_a \leftarrow \max(\lfloor \frac{(a-L)w}{Y} \rfloor, 0)$
  $i_b \leftarrow \min(\lfloor \frac{(b-L)w}{Y} \rfloor, w-1)$
  if $(a \le L) \wedge (R \le b)$ **then**
    return (`Tmin[p]` AND $2^j) = 2^j$
  if $(R < a) \vee (b < L)$ **then**
    return `true`
  if (`Tadd[p]` AND $2^j) = 2^j$ **then**
    `Tmin[k]` $\leftarrow 2^w - 1$
    `Tadd[k]` $\leftarrow 2^w - 1$
    `Tadd[p]` $\leftarrow$ `Tadd[p]` AND $(2^w - 1 - 2^j)$
  return SegTreeMinRam($wk + 1 + i_a, [a, b]$)
    $\wedge$ SegTreeMinRam($wk + 1 + i_b, [a, b]$)
    $\wedge$ (`Tmin[k]` AND $(2^{i_b} - 2^{i_a+1}) = (2^{i_b} - 2^{i_a+1})$)

By using a table to simulate the tree, we do not need to store the interval of each node and we can store the segment tree by using $O(n)$ bits. Indeed, the tables `Tmin` and `Tadd` are of size h. As $\lceil \frac{n}{w} \rceil \le \frac{2n}{w}$ and $\frac{w^{\lceil \log_w n \rceil - 1} - 1}{w-1} \le 2 \times w^{\lceil \log_w n \rceil - 2} \le \frac{2n}{w}$, we need at most $w \times 4 \times \frac{n}{w} = 4n$ bits to store each table.

That is, the space for the segment tree is reduced to $O(n)$ bits (i.e., $O(\frac{n}{\log n})$ words) by using the two tables, but the space complexity of building the HOG remains $O(||P||)$ due to the size of the HOG itself.

5 Conclusion

We have presented a new algorithm to compute the HOG in $O(||P|| \log n)$ time and linear space, which improves upon an earlier solution, and a version of our algorithm using bitwise operations in the word RAM model of computation.

Several interesting questions concerning the HOG and EHOG deserve future work. The *reverse engineering* of indexing data structures, also termed *inference* or *recognition* problem, has attracted a lot of interest. The question is, for instance, given a tree, can one decide whether it is the suffix tree of some string or not? The reverse engineering problem has been studied, e.g., for the suffix tree [16] or the longest-common-prefix array [18]. In 2014, Gevezes and Pitsoulis investigated the reverse engineering of overlap graphs [10]: given a weighted directed graph G, find an instance P such that the overlap graph of P equals G. Clearly this question can be applied to the EHOG and HOG, where the weight on an arc (which is the length of the label on the arc) may or may not be given.

The sizes of the EHOG and HOG (in the number of nodes) can be equal, but they may differ considerably [9]. An average case analysis of their sizes could help understand their differences, and predict the memory required for storing them. Some results connected to this question exist in the literature, e.g., [24] for tries. The notion of *clusters* of word occurrences [4,13,17,28] can be helpful in investigating the number of nodes of the EHOG and HOG for a random set of words.

References

1. Aho, A.V., Corasick, M.J.: Efficient string matching: An aid to bibliographic search. Communications of the ACM **18**(6), 333–340 (1975). https://doi.org/10.1145/360825.360855
2. Arge, L., Brodal, G.S., Georgiadis, L.: Improved dynamic planar point location. In: 47th Proc. of FOCS. pp. 305–314 (2006). https://doi.org/10.1109/FOCS.2006.40
3. Armen, C., Stein, C.: A $2\frac{2}{3}$-approximation algorithm for the shortest superstring problem. In: CPM. pp. 87–101 (1996). https://doi.org/10.1007/3-540-61258-0_8
4. Bassino, F., Clement, J., Nicodeme, P.: Counting occurrences for a finite set of words: Combinatorial methods. ACM Transactions on Algorithms **8**(3), 31:1–31:28 (2012). https://doi.org/10.1145/2229163.2229175
5. Berg, M., Kreveld, M., Overmars, M., Schwarzkopf, O.: Computational geometry: Algorithms and applications, 3rd edition. Springer-Verlag, Berlin (2008). https://doi.org/10.1007/978-3-540-77974-2
6. Blum, A., Jiang, T., Li, M., Tromp, J., Yannakakis, M.: Linear approximation of shortest superstrings. Journal of the ACM **41**(4), 630–647 (1994). https://doi.org/10.1145/179812.179818
7. Cazaux, B., Juhel, S., Rivals, E.: Practical lower and upper bounds for the shortest linear superstring. In: SEA. pp. 18:1–18:14 (2018). https://doi.org/10.4230/LIPIcs.SEA.2018.18
8. Cazaux, B., Rivals, E.: A linear time algorithm for shortest cyclic cover of strings. Journal of Discrete Algorithms **37**, 56–67 (2016). https://doi.org/10.1016/j.jda.2016.05.001

9. Cazaux, B., Rivals, E.: Hierarchical overlap graph. Information Processing Letters **155**, 105862 (2020). https://doi.org/10.1016/j.ipl.2019.105862
10. Gevezes, T.P., Pitsoulis, L.S.: Recognition of overlap graphs. Journal of Combinatorial Optimization **28**(1), 25–37 (2014). https://doi.org/10.1007/s10878-013-9663-3
11. Giora, Y., Kaplan, H.: Optimal dynamic vertical ray shooting in rectilinear planar subdivisions. ACM Transactions on Algorithms **5**(3) (2009). https://doi.org/10.1145/1541885.1541889
12. Gonnella, G., Kurtz, S.: Readjoiner: A fast and memory efficient string graph-based sequence assembler. BMC Bioinformatics **13**(1), 82 (2012). https://doi.org/10.1186/1471-2105-13-82
13. Guibas, L.J., Odlyzko, A.M.: Periods in strings. Journal of Combinatorial Theory, Series A **30**(1), 19–42 (1981). https://doi.org/10.1016/0097-3165(81)90038-8
14. Gusfield, D., Landau, G.M., Schieber, B.: An efficient algorithm for the all pairs suffix-prefix problem. Information Processing Letters **41**(4), 181–185 (1992). https://doi.org/10.1016/0020-0190(92)90176-V
15. Hagerup, T.: Sorting and searching on the word RAM. In: STACS. pp. 366–398 (1998). https://doi.org/10.1007/BFb0028575
16. I, T., Inenaga, S., Bannai, H., Takeda, M.: Inferring strings from suffix trees and links on a binary alphabet. Discrete Applied Mathematics **163**, 316–325 (2014). https://doi.org/10.1016/j.dam.2013.02.033
17. Jacquet, P., Szpankowski, W.: Autocorrelation on words and its applications: Analysis of suffix trees by string-ruler approach. Journal of Combinatorial Theory, Series A **66**(2), 237–269 (1994). https://doi.org/10.1016/0097-3165(94)90065-5
18. Karkkainen, J., Piatkowski, M., Puglisi, S.J.: String inference from longest-common-prefix array. In: ICALP. LIPIcs, vol. 80, pp. 62:1–62:14 (2017). https://doi.org/10.4230/LIPIcs.ICALP.2017.62
19. Laaksonen, A.: Guide to competitive programming, pp. 246–248. Springer (2017). https://doi.org/10.1007/978-3-319-72547-5
20. Lim, J., Park, K.: A fast algorithm for the all-pairs suffix–prefix problem. Theoretical Computer Science **698**, 14–24 (2017). https://doi.org/10.1016/j.tcs.2017.07.013
21. Mucha, M.: Lyndon words and short superstrings. In: SODA. pp. 958–972. SIAM (2013). https://doi.org/10.1137/1.9781611973105.69
22. Myers, E.W.: The fragment assembly string graph. Bioinformatics **21 Suppl 2**, ii79–ii85 (2005). https://doi.org/10.1093/bioinformatics/bti1114
23. Paluch, K.: Better approximation algorithms for maximum asymmetric traveling salesman and shortest superstring (2014), `https://arxiv.org/abs/1401.3670`
24. Park, G., Hwang, H., Nicodeme, P., Szpankowski, W.: Profiles of tries. SIAM Journal of Computing **38**(5), 1821–1880 (2009). https://doi.org/10.1137/070685531
25. Peltola, H., Soderlund, H., Tarhio, J., Ukkonen, E.: Algorithms for some string matching problems arising in molecular genetics. In: IFIP Congress. pp. 53–64 (1983)
26. Pevzner, P.A., Tang, H., Waterman, M.S.: An eulerian path approach to DNA fragment assembly. Proceedings of the National Academy of Sciences **98**(17), 9748–9753 (2001). https://doi.org/10.1073/pnas.171285098
27. Rachid, M.H., Malluhi, Q.: A practical and scalable tool to find overlaps between sequences. BioMed Research International **2015** (2015). https://doi.org/10.1155/2015/905261
28. Robin, S., Rodolphe, F., Schbath, S.: DNA, Words and Models. Cambridge Univ. Press (2005)
29. Sweedyk, Z.: A $2\frac{1}{2}$-approximation algorithm for shortest superstring. SIAM Journal on Computing **29**(3), 954–986 (2000). https://doi.org/10.1137/S0097539796324661