A Linear Time Algorithm for Constructing Hierarchical Overlap Graphs

Sangsoo Park
Samsung Electronics, Seoul, Korea

Sung Gwan Park
Samsung Electronics, Seoul, Korea

Bastien Cazaux
LIRMM, Université Montpellier, CNRS, Montpellier, France

Kunsoo Park
Seoul National University, Seoul, Korea

Eric Rivals
LIRMM, Université Montpellier, CNRS, Montpellier, France

Abstract

The hierarchical overlap graph (HOG) is a graph that encodes overlaps from a given set P of n strings, as the overlap graph does. A best known algorithm constructs HOG in $O(||P|| \log n)$ time and $O(||P||)$ space, where $||P||$ is the sum of lengths of the strings in P. In this paper we present a new algorithm to construct HOG in $O(||P||)$ time and space. Hence, the construction time and space of HOG are better than those of the overlap graph, which are $O(||P|| + n^2)$.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases overlap graph, hierarchical overlap graph, shortest superstring problem, border array

Digital Object Identifier 10.4230/LIPIcs.CPM.2021.22

Funding S. Park, S.G. Park and K. Park were 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). B. Cazaux and E. Rivals acknowledge the funding from Labex NumeV (GEM flagship project, ANR 2011-LABX-076), and from the Marie Skłodowska-Curie Innovative Training Networks ALPACA (grant 956229).

1 Introduction

For a set of strings, a *superstring* of the set is a string that has all strings in the set as substrings. The *shortest superstring* problem is to find a shortest superstring of a set of strings. This problem is known to play an important role in *DNA assembly*, which is the problem to restore the entire genome from short sequencing reads. Despite its importance, the shortest superstring problem is known to be NP-hard [6]. As a result, extensive research has been done to find good approximation algorithms for the shortest superstring problem [2, 18, 11, 13, 19, 20].

The shortest superstring problem is reduced to finding a shortest hamiltonian path in a graph that encodes overlaps between the strings [2, 12, 16], which is the *distance graph* or equivalent *overlap graph*. The overlap graph [15] of a set of strings is a graph in which each string constitutes a node and an edge connecting two nodes shows the longest overlap between them. Many approaches for approximating the shortest superstring problem focus on the overlap graph, and try to find good approximations of its hamiltonian path [11, 13].

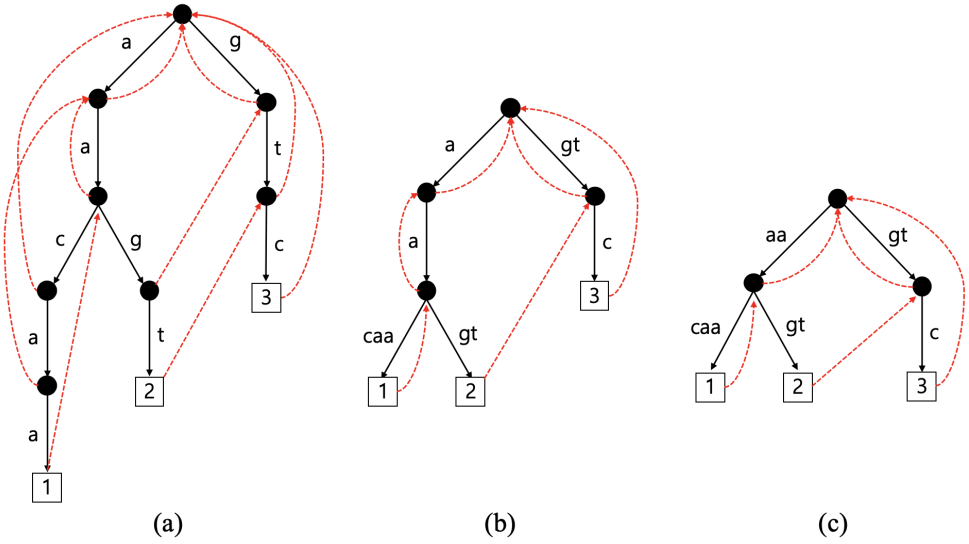

Figure 1 Data structures built with $P = \{aacaa, aagt, gtc\}$. (a) Aho–Corasick trie. (b) Extended Hierarchical Overlap Graph. (c) Hierarchical Overlap Graph.

Given a set of strings $P = \{s_1, s_2, ..., s_n\}$, computing the overlap graph of P is equivalent to solving the *all-pair suffix-prefix problem*, which is to find the longest overlap for every pair of strings in P. The best theoretical bound for this problem is $O(||P|| + n^2)$ [8], where $||P||$ is the sum of lengths of the strings in P. Since the input size of the problem is $O(||P||)$ and the output size is $O(n^2)$, this bound is optimal. There has also been extensive research on the all-pair suffix-prefix problem in the practical point of view [7, 10, 17] because it is the first step in DNA assembly.

Recently, Cazaux and Rivals [4, 5] proposed a new graph which stores the overlap information, called the *hierarchical overlap graph* (HOG). HOG is a graph with two types of edges (which will be defined in Section 2) in which a node represents either a string or the longest overlap between a pair of strings. The *extended hierarchical overlap graph* (EHOG) is also a graph with two types of edges in which a node represents either a string or an overlap between a pair of strings (which may be not the longest one). For example, Figure 1 shows EHOG and HOG built with $P = \{aacaa, aagt, gtc\}$. Even though HOG and EHOG may be the same for some input instances, there is a series of instances where the ratio of EHOG size over the HOG size tends to infinity with respect to the number of nodes. Therefore, HOG has an advantage over EHOG in both practical and theoretical points of view.

HOG also has a couple of advantages compared to the overlap graph [5]. First, HOG uses only $O(||P||)$ space, while the overlap graph needs $O(||P|| + n^2)$ space in total. For input instances with many short strings, HOG uses a considerably smaller amount of space than the overlap graph. Second, HOG contains the relationship between the overlaps themselves, since the overlaps appear as nodes in HOG. In contrast, the overlap graph stores only the lengths of the longest overlaps, and thus we cannot find the relationship between two overlaps easily. Therefore, HOG stores more information than the overlap graph, while using less space.

There have been many works to compute HOG and EHOG efficiently. Computing the EHOG from P costs $O(||P||)$ time, which is optimal [3]. For computing the HOG, Cazaux and Rivals proposed an $O(||P|| + n^2)$ time algorithm using $O(||P|| + n \times \min(n, \max\{|s| : s \in P\}))$ space [5]. Recently, Park et al. [14] gave an $O(||P|| \log n)$ time algorithm using $O(||P||)$ space by using the segment tree data structure.

In this paper we present a new algorithm to compute HOG, which uses $O(||P||)$ time and space, which are both optimal. The algorithm is based on the Aho–Corasick trie [1] and the border array [9]. Therefore, the construction time and space of HOG are better than those of the overlap graph, which are $O(||P|| + n^2)$, and this fact may lead to many applications of HOG. For example, consider the problem of finding *optimal cycle cover* in the overlap graph built with a set $P = \{s_1, s_2, ..., s_n\}$ of strings. Typically this problem needs to be solved in finding good approximations of shortest superstrings. A greedy algorithm to solve the optimal cycle cover problem on the overlap graph was given in [2], which takes $O(||P|| + n^2)$ time. Recently, Cazaux and Rivals proposed an $O(||P||)$ time algorithm to solve the optimal cycle cover problem given the HOG or EHOG of P [4]. By using our result in this paper, the optimal cycle cover problem can be solved in $O(||P||)$ time and space by using HOG instead of the overlap graph.

The rest of the paper is organized as follows. In Section 2 we give preliminary information for HOG and formalize the problem. In Section 3 we present an $O(||P||)$ time and space algorithm for computing HOG. In Section 4 we conclude and discuss a future work.

2 Preliminaries

2.1 Basic notation

We consider strings over a constant-size alphabet Σ. The length of a string s is denoted by $|s|$. Given two integers $1 \leq l \leq r \leq |s|$, the substring of s which starts from l and ends at r is denoted by $s[l..r]$. Note that $s[l..r]$ is a prefix of s when $l = 1$, and a suffix of s when $r = |s|$. If a prefix (suffix) of s is different from s, we call it a proper prefix (suffix) of s. Given two strings s and t, an *overlap* from s to t is a string which is both a proper suffix of s and a proper prefix of t. Given a set $P = \{s_1, s_2, ..., s_n\}$ of strings, the sum of $|s_i|$'s is denoted by $||P||$.

2.2 Hierarchical Overlap Graph

We define *hierarchical overlap graph* and *extended hierarchical overlap graph* as in [5].

▶ **Definition 1** (Hierarchical Overlap Graph). Given a set $P = \{s_1, s_2, \ldots, s_n\}$, we define $Ov(P)$ as 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 with a vertex set $V = P \cup Ov(P) \cup \{\epsilon\}$ and an edge set $E = E_1 \cup E_2$, 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** (Extended Hierarchical Overlap Graph). Given a set $P = \{s_1, s_2, \ldots, s_n\}$, we define $Ov^+(P)$ as 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 with a vertex set $V^+ = P \cup Ov^+(P) \cup \{\epsilon\}$ and an edge set $E^+ = E_1^+ \cup E_2^+$, 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 $x\}$.

Figure 1 shows the Aho–Corasick trie [1], EHOG, and HOG built with $P = \{aacaa, aagt, gtc\}$. It is shown in [5] that EHOG is a contracted form of the Aho–Corasick trie and HOG is a contracted form of EHOG.

As in the Aho–Corasick trie, each node u in HOG or EHOG corresponds to a string (denoted by $S(u)$), which is a concatenation of all labels on the path from the root (node representing ϵ) to u.

There are two types of edges in EHOG and HOG as in the Aho–Corasick trie: a tree edge and a failure link. An edge (u, v) is a tree edge (an edge in E_1^+ or E_1, solid line in Figure 1) in an EHOG (HOG), if $S(u)$ is the longest proper prefix of $S(v)$ in the EHOG (HOG). It is a failure link (an edge in E_2^+ or E_2, dotted line in Figure 1) in an EHOG (HOG), if $S(v)$ is the longest proper suffix of $S(u)$ in the EHOG (HOG).

Given a set $P = \{s_1, s_2, \ldots, s_n\}$ of strings, we can build an EHOG of P in $O(||P||)$ time and space [5]. Furthermore, given EHOG(P) and $Ov(P)$, we can compute HOG(P) in $O(||P||)$ time and space [5]. Therefore, the bottleneck of computing HOG(P) is computing $Ov(P)$ efficiently.

3 Computing HOG in linear time

In this section we introduce an algorithm to build the HOG of $P = \{s_1, s_2, \ldots, s_n\}$ in $O(||P||)$ time. We assume that there are no two different strings $s_i, s_j \in P$ such that s_i is a substring of s_j for simplicity of presentation. Our algorithm directly computes HOG(P) (and $Ov(P)$) from the Aho–Corasick trie of P in $O(||P||)$ time.

Let us assume we have the Aho–Corasick trie of P including the failure links. We define $R(u)$ for each node u of the trie, as follows:

$$R(u) = \{i \in \{1, \ldots, n\} \mid S(u) \text{ is a proper prefix of } s_i\}. \tag{1}$$

That is, $R(u)$ is a set of string indices in the subtree rooted at u if u is an internal node, or an empty set if u is a leaf node.

For each input string s_i, we will do the following operation separately, which is to find the longest overlap from s_i to any string in P. Consider a path $(v_0, v_1, \ldots, v_l)$ which starts from the leaf representing s_i and follows the failure links until it reaches the root, i.e., $S(v_0) = s_i$ and v_l is the root of the tree. By definition of the failure link, the strings corresponding to nodes appearing in the path are suffixes of s_i. If there are an index j and a node v_k on the path such that $j \in R(v_k)$, $S(v_k)$ is both a suffix of s_i and a proper prefix of s_j, so $S(v_k)$ is an overlap from s_i to s_j.

$S(v_k)$ for $0 < k \leq l$ is the longest overlap from s_i to s_j if and only if $j \in R(v_k)$ and there is no m such that $0 \leq m < k$ and $j \in R(v_m)$. If there exists such m, then $S(v_m)$ is a longer overlap from s_i to s_j than $S(v_k)$, so $S(v_k)$ is not the longest overlap. Therefore, we get the following lemma.

▶ **Lemma 3.** $S(v_k)$ is the longest overlap from s_i to s_j if and only if $j \in R(v_k) - R(v_{k-1}) - \ldots - R(v_0)$.

Therefore, if $|R(v_k) - R(v_{k-1}) - \ldots - R(v_0)| > 0$, $S(v_k)$ is the longest overlap from s_i to s_j for $j \in R(v_k) - R(v_{k-1}) - \ldots - R(v_0)$, and thus $v_k \in Ov(P)$. Therefore, we aim to compute $|R(v_k) - R(v_{k-1}) - \ldots - R(v_0)|$ for every $0 < k \leq l$.

Given an index k, we define $k + 1$ auxiliary sets of indices $I_k(k), I_k(k-1), \ldots, I_k(0)$ in a recursive manner as follows.

- $I_k(k) = R(v_k)$
- $I_k(m) = I_k(m+1) - R(v_m)$ for $m = k-1, k-2, \ldots, 0$

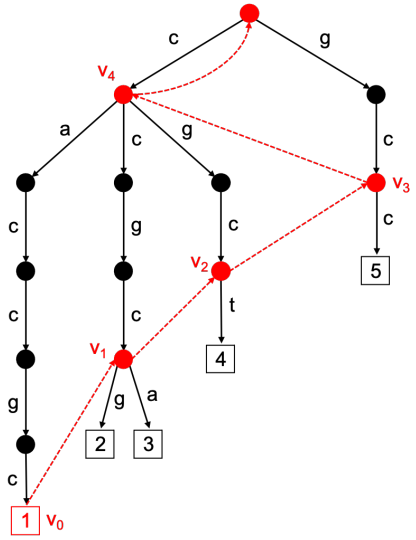

Figure 2 Aho–Corasick trie with $P = \{caccgc, ccgcg, ccgca, cgct, gcc\}$.

By definition, $I_k(0)$ is $R(v_k) - R(v_{k-1}) - \ldots - R(v_0)$ in Lemma 3 and we want to compute $|I_k(0)|$. For every $0 \leq m < k$, $I_k(m) = I_k(m+1) - R(v_m) \subseteq I_k(m+1)$ and thus $|I_k(m)| = |I_k(m+1)| - |I_k(m+1) - I_k(m)|$ holds. By summing up all these equations for $0 \leq m < k$, we get $|I_k(0)| = |I_k(k)| - \sum_{m=0}^{k-1} |I_k(m+1) - I_k(m)|$. Since $I_k(k) = R(v_k)$ and $I_k(m+1) - I_k(m) = I_k(m+1) - (I_k(m+1) - R(v_m)) = I_k(m+1) \cap R(v_m)$, we have

$$|I_k(0)| = |R(v_k)| - \sum_{m=0}^{k-1} |I_k(m+1) \cap R(v_m)|. \tag{2}$$

We also define a new function $up(u)$ for a node u as follows.

▶ **Definition 4.** Given a node u in the Aho–Corasick trie, $up(u)$ is defined as the first ancestor of u (except u itself) in the path that starts at u and follows the failure links until it reaches the root node. We define an ancestor on the tree which consists of tree edges in the Aho–Corasick trie.

Note that $up(u)$ is well defined when u is not the root node, since the root node is always an ancestor of u. When u is the root node, $up(u)$ is empty.

Now we analyze the value of $|I_k(m+1) \cap R(v_m)|$ in Equation (2) for each $0 \leq m < k$ as follows. We use a path $(v_0, v_1, ..., v_5)$ in Figure 2 as a running example, i.e., $l = 5$ and $0 < k \leq 5$.

▶ **Lemma 5.** $|I_k(m+1) \cap R(v_m)|$ is $|R(v_m)|$ if $up(v_m) = v_k$; it is 0 otherwise.

Proof. We divide the relationship between v_m and v_k into cases.

1. v_m is outside the subtree rooted at v_k
   Let us assume that $I_k(m+1) \cap R(v_m)$ is not empty and there exists $j \in I_k(m+1) \cap R(v_m)$. Then $j \in R(v_k) \cap R(v_m)$ should hold since $I_k(m+1) \subseteq I_k(k) = R(v_k)$. Therefore, both v_m and v_k should be ancestors of the leaf corresponding to s_j. Because $|S(v_m)| > |S(v_k)|$, v_k should be an ancestor of v_m. Since v_m is outside the subtree rooted at v_k, v_k cannot be an ancestor of v_m, which is a contradiction. Therefore such j does not exist, which shows that $I_k(m+1) \cap R(v_m) = \emptyset$ and $|I_k(m+1) \cap R(v_m)| = 0$.

For example, consider the case with $k = 4$ and $m = 3$ in Figure 2. Since $I_4(4) = R(v_4) = \{1, 2, 3, 4\}$ and $R(v_3) = \{5\}$, we can see that $I_4(4) \cap R(v_3) = \emptyset$.

2. v_m is inside the subtree rooted at v_k
In this case, v_k is an ancestor of v_m and we further divide it into cases.

a. There exists q such that $m < q < k$ and v_q is an ancestor of v_m.
We get $R(v_m) \subseteq R(v_q)$ because v_q is an ancestor of v_m. Since $I_k(m+1) = R(v_k) - R(v_{k-1}) - ... - R(v_{m+1})$ and $m < q < k$, we have $I_k(m+1) \subseteq R(v_k) - R(v_q)$. Therefore, $I_k(m+1) \cap R(v_m) \subseteq (R(v_k) - R(v_q)) \cap R(v_q) = \emptyset$. That is, $I_k(m+1) \cap R(v_m) = \emptyset$ and $|I_k(m+1) \cap R(v_m)| = 0$.

b. For any q such that $m < q < k$, v_q is not an ancestor of v_m.
Here we show that $R(v_m) \subseteq I_k(m+1)$. Let us consider an index $x \in R(v_m)$. Since v_k is an ancestor of v_m, we have $x \in R(v_k)$. Moreover, for any q such that $m < q < k$, neither v_q is an ancestor of v_m nor v_m is an ancestor of v_q. That is, $R(v_q) \cap R(v_m) = \emptyset$ and thus $x \notin R(v_q)$. Therefore, we have $x \in I_k(m+1) = R(v_k) - R(v_{k-1}) - \ldots - R(v_{m+1})$. In conclusion, $R(v_m) \subseteq I_k(m+1)$ and thus $|I_k(m+1) \cap R(v_m)| = |R(v_m)|$.
For example, consider the case with $k = 4$ and $m = 1$ in Figure 2. Since $I_4(2) = R(v_4) - R(v_3) - R(v_2) = \{1, 2, 3\}$ and $R(v_1) = \{2, 3\}$, we can see that $R(v_1) \subseteq I_4(2)$ and $I_4(2) \cap R(v_1) = R(v_1)$.

In summary, $|I_k(m+1) \cap R(v_m)| = |R(v_m)|$ in case 2(b), and 0 otherwise. In case 2(b), v_k is an ancestor of v_m and there is no q such that $m < q < k$ and v_q is an ancestor of v_m. In other words, v_k is the first ancestor of v_m in the path starting from v_m and following the failure links repeatedly, which means that $up(v_m) = v_k$. ◀

▶ **Theorem 6.** For every $0 < k \leq l$, $|I_k(0)| = |R(v_k)| - \sum_{v_m} |R(v_m)|$, where $0 \leq m < k$ and $up(v_m) = v_k$.

Proof. From Equation (2), we have $|I_k(0)| = |R(v_k)| - \sum_{m=0}^{k-1} |I_k(m+1) \cap R(v_m)|$. By Lemma 5, we have $\sum_{m=0}^{k-1} |I_k(m+1) \cap R(v_m)| = \sum_{v_m : up(v_m) = v_k} |R(v_m)|$. By merging the two equations, we have the theorem. ◀

Now let us consider the relationship between u and $up(u)$. $S(up(u))$ is a proper suffix of $S(u)$ because $up(u)$ can be reached from u through failure links. Furthermore, $S(up(u))$ is a proper prefix of $S(u)$ because $up(u)$ is an ancestor of u. That is, $S(up(u))$ is a *border* [9] of $S(u)$. Moreover, we visit every suffix of $S(u)$ in the trie in the decreasing order of lengths and $S(up(u))$ is the first border we visit, so $S(up(u))$ is the *longest border* of $S(x)$. Since each node in the Aho–Corasick trie corresponds to a prefix of some s_i, we can compute $up(u)$ for all nodes u by computing the border array of every s_i as follows. Let $pnode_i(l)$ be the node which corresponds to $s_i[1..l]$, and $border_i(l)$ be the length of the longest border of $s_i[1..l]$. Then we have the following equation for every s_i and $1 \leq l \leq |s_i|$:

$$up(pnode_i(l)) = pnode_i(border_i(l)). \tag{3}$$

If we store $pnode_i$ and $border_i$ using arrays, we can compute $pnode_i, border_i$, and $up(u)$ in $O(||P||)$ time and space, because $border_i$ can be computed in $O(||P||)$ time using an algorithm in [9].

▶ **Example 7.** Let us consider Figure 2, which is an Aho–Corasick trie built with a set $P = \{s_1 = caccgc, s_2 = ccgcg, s_3 = ccgca, s_4 = cgct, s_5 = gcc\}$ of strings. For each string, we compute its corresponding border array, and get $border_1 = (0, 0, 1, 1, 0, 1)$, $border_2 = (0, 1, 0, 1, 0)$, $border_3 = (0, 1, 0, 1, 0)$, $border_4 = (0, 0, 1, 0)$, and $border_5 = (0, 0, 0)$. We also

Algorithm 1 Build HOG in linear time.

```
1:  procedure BUILD-HOG(P)
2:      Build the Aho–Corasick trie with P
3:      Compute border arrays border_i for 1 ≤ i ≤ n
4:      Compute up(u) for each node u
5:      Compute |R(u)| for each node u
6:      Mark root as included in HOG(P)
7:      For each node u, initialize Child(u) with an empty set
8:      for i ← 1 to n do
9:          u ← leaf corresponding to s_i in Aho–Corasick trie
10:         Mark u as included in HOG(P)
11:         while u ≠ root do
12:             I(u) ← |R(u)|
13:             for all u' ∈ Child(u) do
14:                 I(u) ← I(u) − |R(u')|
15:             if I(u) > 0 then
16:                 Mark u as included in HOG(P)
17:             Child(u) ← an empty set
18:             Add u to Child(up(u))
19:             u ← failure link of u
20:     Build HOG(P) with marked nodes
```

store $pnode_i$'s by traversing the Aho–Corasick trie. Now we can compute up by using $pnode_i$ and $border_i$. For example, let us consider $v_1 = pnode_2(4)$, which represents $ccgc$. Since the longest border of $ccgc$ is c, which has length 1, we have $border_2(4) = 1$. As a result, we have $up(v_1) = up(pnode_2(4)) = pnode_2(border_2(4)) = pnode_2(1) = v_4$ by Equation (3). Note that v_4 represents c, which is the longest border of $ccgc$.

We are ready to describe an algorithm to compute HOG of P in $O(||P||)$ time and space. First, we build the Aho–Corasick trie with P and a border array for each s_i. By using the border arrays, we compute $up(u)$ for every node u except the root. Next, we compute $|R(u)|$ for each node u by the post-order traversal of the Aho–Corasick trie. For each string s_i, we start from the leaf node corresponding to s_i and follow the failure links until we reach the root. For each node v_k that we visit, we compute its corresponding $|I_k(0)| = |R(v_k)| - \sum_{v_m : up(v_m) = v_k} |R(v_m)|$. If $|I_k(0)| > 0$, we mark v_k to be included in HOG. Algorithm 1 shows an algorithm to compute HOG. Lines 2–5 compute the preliminaries for the algorithm, while lines 6–19 compute the nodes to be included in HOG. Note that the loop of lines 8–19 works separately for each input string s_i. We consider v_k in the order of increasing k, and thus if $up(v_m) = v_k$, then $m < k$. Hence, Child(v_k) in line 13 stores every v_m such that $up(v_m) = v_k$ by line 18 of previous iterations. For each node $u = v_k$ in lines 11–19, $I(u)$ correctly computes $|I_k(0)|$ since we get $|R(v_k)|$ in line 12 and subtract every $|R(v_m)|$ where $v_k = up(v_m)$ in lines 13–14. According to Theorem 6, lines 12–14 correctly computes $|I_k(0)|$. We build HOG(P) in line 20 by removing the unmarked nodes and contracting the edges while traversing the Aho–Corasick trie once, as in [5].

▶ **Example 8.** Consider again the Aho–Corasick trie built with $P = \{s_1 = caccgc, s_2 = ccgcg, s_3 = ccgca, s_4 = cgct, s_5 = gcc\}$ in Figure 2. Let us consider a path starting from a node representing s_1 and following the failure links until the root node. The path

$(v_0, v_1, v_2, v_3, v_4, v_5)$ is marked with dotted lines in Figure 2. By definition of up, we get $up(v_0) = up(v_1) = up(v_2) = v_4$ and $up(v_3) = up(v_4) = v_5$. Therefore, we can compute $|I_k(0)|$'s as follows.

- $|I_0(0)| = |R(v_0)| = 0$
- $|I_1(0)| = |R(v_1)| = 2$
- $|I_2(0)| = |R(v_2)| = 1$
- $|I_3(0)| = |R(v_3)| = 1$
- $|I_4(0)| = |R(v_4)| - |R(v_0)| - |R(v_1)| - |R(v_2)| = 4 - 0 - 2 - 1 = 1$

Note that $|R(v_0)| = 0$ by definition of $R(u)$. Since v_1, v_2, v_3, and v_4 have positive $|I_k(0)|$'s, we mark them to be included in HOG. We do this process for every s_i.

Now we show that Algorithm 1 runs in $O(||P||)$ time and space. Computing an Aho–Corasick trie, a border array for each string, and $up(u)$ and $|R(u)|$ for each node u costs $O(||P||)$ time and space. Furthermore, for a given index i, lines 13–14 are executed at most $|s_i|$ times since line 18 is executed at most $|s_i|$ times, and thus the sum of $|\mathrm{Child}(u)|$ is at most $|s_i|$. Therefore, lines 9–19 run in $O(|s_i|)$ time for given i, and thus lines 8–19 run in $O(||P||)$ time in total. Also they use $O(|s_i|)$ additional space to store the Child list. Lastly, we can build HOG(P) with marked nodes in $O(||P||)$ space and time [5]. Therefore, Algorithm 1 runs in $O(||P||)$ time and space. We remark that Algorithm 1 can be modified so that it builds the HOG from an EHOG instead of an Aho–Corasick trie, while it still costs $O(||P||)$ time and space.

▶ **Theorem 9.** Given a set P of strings, HOG(P) can be built in $O(||P||)$ time and space.

4 Conclusion

We have presented an $O(||P||)$ time and space algorithm to build the HOG, which improves upon an earlier $O(||P|| \log n)$ time solution. Since the input size of the problem is $O(||P||)$, the algorithm is optimal.

There are some interesting topics about HOG and EHOG which deserve the future work. As mentioned in the introduction, the *shortest superstring problem* gained a lot of interest [2, 18, 11, 13]. Since many algorithms dealing with the shortest superstring problem are based on the overlap graph, HOG may give better approximation algorithms for the shortest superstring problem by using the additional information that HOG has when compared to the overlap graph.

References

1. A. V. Aho and M. J. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975. doi:10.1145/360825.360855.
2. A. Blum, T. Jiang, M. Li, J. Tromp, and M. Yannakakis. Linear approximation of shortest superstrings. *Journal of the ACM*, 41(4):630–647, 1994. doi:10.1145/179812.179818.
3. B. Cazaux, R. Cánovas, and E. Rivals. Shortest DNA cyclic cover in compressed space. In *DCC*, pages 536–545, 2016. doi:10.1109/DCC.2016.79.
4. B. Cazaux and E. Rivals. A linear time algorithm for shortest cyclic cover of strings. *Journal of Discrete Algorithms*, 37:56–67, 2016. doi:10.1016/j.jda.2016.05.001.
5. B. Cazaux and E. Rivals. Hierarchical overlap graph. *Information Processing Letters*, 155:105862, 2020. doi:10.1016/j.ipl.2019.105862.
6. J. Gallant, D. Maier, and J. Astorer. On finding minimal length superstrings. *Journal of Computer and System Sciences*, 20(1):50–58, 1980. doi:10.1016/0022-0000(80)90004-5.

7 G. Gonnella and S. Kurtz. Readjoiner: A fast and memory efficient string graph-based sequence assembler. *BMC Bioinformatics*, 13(1):82, 2012. doi:10.1186/1471-2105-13-82.

8 D. Gusfield, G. M. Landau, and B. Schieber. An efficient algorithm for the all pairs suffix-prefix problem. *Information Processing Letters*, 41(4):181–185, 1992. doi:10.1016/0020-0190(92)90176-V.

9 D. E. Knuth, J. H. Morris, Jr., and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977. doi:10.1137/0206024.

10 J. Lim and K. Park. A fast algorithm for the all-pairs suffix–prefix problem. *Theoretical Computer Science*, 698:14–24, 2017. doi:10.1016/j.tcs.2017.07.013.

11 M. Mucha. Lyndon words and short superstrings. In *SODA*, pages 958–972. SIAM, 2013. doi:10.1137/1.9781611973105.69.

12 E. W. Myers. The fragment assembly string graph. *Bioinformatics*, 21 Suppl 2:ii79–ii85, 2005. doi:10.1093/bioinformatics/bti1114.

13 K. Paluch. Better approximation algorithms for maximum asymmetric traveling salesman and shortest superstring, 2014. arXiv:1401.3670.

14 S. G. Park, B. Cazaux, K. Park, and E. Rivals. Efficient construction of hierarchical overlap graphs. In *SPIRE*, pages 277–290, 2020. doi:10.1007/978-3-030-59212-7_20.

15 H. Peltola. Algorithms for some string matching problems arising in molecular genetics. In *IFIP Congress*, pages 53–64, 1983.

16 P. A. Pevzner, H. Tang, and M. S. Waterman. An Eulerian path approach to DNA fragment assembly. *Proceedings of the National Academy of Sciences*, 98(17):9748–9753, 2001. doi:10.1073/pnas.171285098.

17 M. H. Rachid and Q. Malluhi. A practical and scalable tool to find overlaps between sequences. *BioMed Research International*, 2015, 2015. doi:10.1155/2015/905261.

18 Z. Sweedyk. A $2\frac{1}{2}$-approximation algorithm for shortest superstring. *SIAM Journal on Computing*, 29(3):954–986, 2000. doi:10.1137/S0097539796324661.

19 J. Tarhio and E. Ukkonen. A greedy approximation algorithm for constructing shortest common superstrings. *Theoretical Computer Science*, 57(1):131–145, 1988. doi:10.1016/0304-3975(88)90167-3.

20 E. Ukkonen. A linear-time algorithm for finding approximate shortest common superstrings. *Algorithmica*, 5(1):313–323, 1990. doi:10.1007/BF01840391.