Hierarchical Overlap Graph

Bastien Cazaux, Eric Rivals

Hierarchical Overlap Graph

Bastien Cazaux, Eric Rivals
L.I.R.M.M. & Institut Biologie Computationnelle,
Université Montpellier, CNRS U.M.R. 5506
161 rue Ada, F-34392 Montpellier Cedex 5, France
cazaux@lirmm.fr,rivals@lirmm.fr

January 19, 2018

Abstract

Given a set of finite words, the Overlap Graph (OG) is a complete weighted digraph where each word is a node and where the weight of an arc equals the length of the longest overlap of one word onto the other (Overlap is an asymmetric notion). The OG serves to assemble DNA fragments or to compute shortest superstrings which are a compressed representation of the input. The OG requires a space is quadratic in the number of words, which limits its scalability. The Hierarchical Overlap Graph (HOG) is an alternative graph that also encodes all maximal overlaps, but uses a space that is linear in the sum of the lengths of the input words. We propose the first algorithm to build the HOG in linear space for words of equal length.

1 Introduction

DNA assembly problem arises in bioinformatics because DNA sequencing is unable to read out complete molecules, but instead yields partial sequences of the target molecule, called reads. Hence, recovering the whole DNA sequence requires to assemble those reads, that is to merge reads according to their longest overlaps (because of the high redundancy of sequencing strategies). DNA assembly comes down to building a digraph where sequences are nodes and arcs represent possible overlaps, and second to choosing a path that dictates in which order to merge the reads. The first proposed graph was the Overlap Graph [9, 10]. Throughout this article, the input is $P := \{s_1, \ldots, s_n\}$ a set of words. Let us denote by $||P|| := \sum_1^n |s_i|$.

Definition 1 (Overlap Graph) *The* Overlap Graph *(OG) of P is a complete, directed graph, weighted on its arcs, whose nodes are the words of P, and in which the weight of an arc (u,v) equals the length of the maximal overlap from string u to string v.*

In the OG (and in its variants such as the String Graph), an optimal assembly path can be computed by finding a Maximum Weighted Hamiltonian Path (which is NP-hard and difficult to

approximate [2]). To build the graph, one has to compute the weights of the arcs by solving the so-called All Pairs Suffix Prefix overlaps problem (APSP) on P. Although, Gusfield has given an optimal time algorithm for APSP in 1992, APSP has recently regained attention due to innovation in sequencing technologies that allow sequencing longer reads. Indeed, solving APSP remains difficult in practice for large datasets. Several other optimal time algorithms that improve on practical running times have been described recently, e.g. [8, 11].

The OG has several drawbacks. First, it is not possible to know whether two distinct arcs represent the same overlap. Second, the OG has an inherently quadratic size since it contains an arc for each possible (directed) pairs of words. Here, we present an alternative graph, called HOG, which represents all maximal overlaps and their relationships in term of suffix and of prefix. Since the HOG takes a space linear in cumulated lengths of the words, it can advantageously replace the OG. Note that we already gave a definition of the HOG in [4]. Here, we proposed the first algorithm to build the HOG in linear space.

In DNA assembly, the de Bruijn Graph (DBG) is also used, especially for large datasets of short reads. For a chosen integer k, reads are split into all their k-long substrings (termed k-mers), which make the nodes of the DBG, and the arcs store only $(k-1)$-long overlaps. Moreover, the relationship between reads and k-mers is not recorded. DBGs achieve linear space, but disregard many overlaps whose lengths differ from k. Hence, HOGs also represent an interesting alternative to DBGs.

1.1 Notation and Definition of the Hierarchical Overlap Graph

We consider finite, linear of strings over a finite alphabet Σ and denote the empty string with ε. Let s be a string over Σ. We denote the length of s by $|s|$. For any two integers $i \leq j$ in $[1,|s|]$, $s[i,j]$ denotes the linear substring of s beginning at the position i and ending at the position j. Then we say that $s[i,j]$ is a *prefix* of s iff $i = 1$, a *suffix* iff $j = |s|$. A prefix (or suffix) s' of s is said *proper* if s' differs from s. For another linear string t, an overlap from s to t is a proper suffix of s that is also a proper prefix of t. We denote the longest such overlaps by $ov(s,t)$. For A, B any two boolean arrays of the same size, we denote by $A \wedge B$ the boolean operation `and` between A and B.
Let $Ov^+(P)$ be the set of all overlaps between words of P. Let $Ov(P)$ be the set of **maximum** overlaps from a string of P to another string or the same string of P.

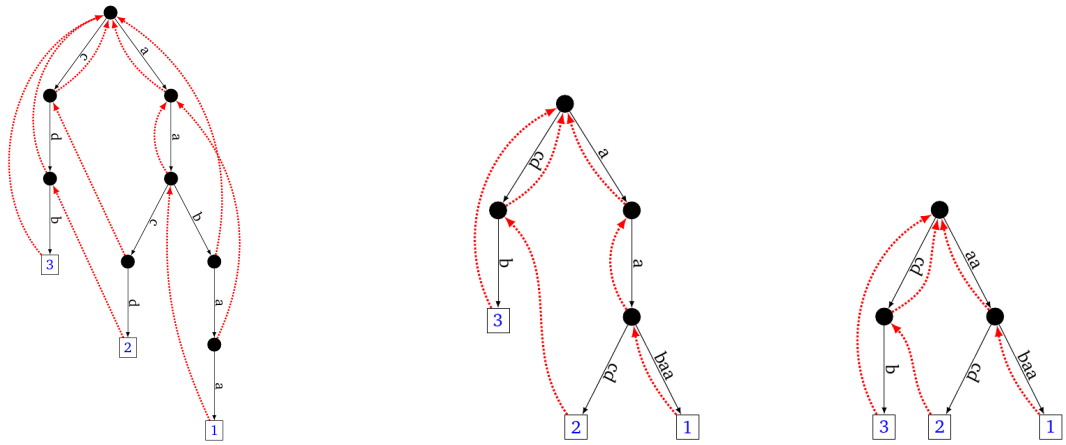
Let us define the *Extended Hierarchical Overlap Graph* and the *Hierarchical Overlap Graph* as follows.

Definition 2 *The* Extended Hierarchical Overlap Graph *of P, denoted by $EHOG(P)$, is the directed graph (V^+, E^+) where $V^+ = P \cup Ov^+(P)$ and E^+ is the set:*

$$\{(x,y) \in (P \cup Ov(P))^2 \mid y \text{ is the longest proper suffix of } x \text{ or } x \text{ is the longest proper prefix of } y\}.$$

The Hierarchical Overlap Graph *of P, denoted by $HOG(P)$, is the directed graph (V,E) where $V := P \cup Ov(P)$ and E is the set:*

$$\{(x,y) \in (P \cup Ov(P))^2 \mid y \text{ is the longest proper suffix of } x \text{ or } x \text{ is the longest proper prefix of } y\}.$$



(a) Aho Corasik tree of P (b) EHOG of P (c) HOG of P

Figure 1: Consider instance $P := \{aabaa, aacd, cdb\}$. (a) Aho Corasik tree of P with `Failure Links` in dotted lines. (b) EHOG of P - compared to (a) all nodes that were not overlaps were removed, and the arcs contracted. (c) HOG of P; compared to (b) the node a has been removed, and some arcs contracted.

Remark that $Ov(P)$ is a subset of $Ov^+(P)$. Both definitions are identical except that each occurrence of $Ov^+(P)$ in the $EHOG(P)$ is replaced by $Ov(P)$ in the $HOG(P)$. Hence, $HOG(P)$ is somehow included in $EHOG(P)$. Examples of EHOG and HOG are shown in Figures 1b, 1c.

1.2 Related works

In 1990, Ukkonen gave a linear time implementation of the greedy algorithm for finding a linear superstring of a set P [12]. This algorithm is based on the Aho-Corasick (AC) automaton [1]. In AC automaton, the input words are spelled out along a tree structure and each node represents a unique prefix of these words ; hence, the leaves represent the input words (Example in Figure 1a: the tree structure, which are is in black). The AC automaton contains additional arcs called, `Failure links`, which link two nodes say (x, y) if y is the longest suffix of x that is a node in the tree. Ukkonen characterised which nodes of the tree correspond to overlaps between words of P [12, Lemma 3]. In the EHOG, beyond the leaves, we keep only those nodes that are overlaps (Figure 1b). From this lemma, it follows that $EHOG(P)$ is embedded in the Aho-Corasick automaton: the nodes of $EHOG(P)$ are a subset of those of the automaton, its arcs are contracted `goto` transitions or contracted `Failure Links`.

Algorithm to build the EHOG. Given the Aho-Corasick automaton of P (whose underlying structure is a tree spelling out the words of P from the root – this tree is commonly called the *trie*), for each leaf, follow the chain of `Failure links` up to the root and mark all visited nodes. Thanks to [12, Lemma 3] all such nodes are overlaps. Another traversal of the AC automaton suffices to remove unmarked nodes and to contract the appropriate arcs. This algorithm takes $O(||P||)$ time. An involved algorithm to build a memory compact version of the EHOG is detailed in [3]. In the sequel, whenever we refer to the *tree* structure, it is the tree structure of the EHOG, we mean the

tree defined by the `goto` arcs in the EHOG (which appears in black in Figure 1b).

An algorithm for computing a shortest cyclic cover of a set of DNA sequences needs to build either the EHOG or HOG [4]. There, we stated mistakingly in Theorem 3 that the HOG could be built in $O(||P||)$ time, although we meant the EHOG. The construction algorithm uses the Generalised Suffix Tree of P to detect nodes representing overlaps as explained in [7].

The HOG is reminiscent of the "Hierarchical Graph" of [6], where the arcs also encodes inclusion between substrings of the input words. However, in the Hierarchical Graph, each arc extends or shortens the string by a single symbol, making it larger than the HOG.

2 Construction algorithm for the HOG

All internal nodes of $EHOG(P)$ are overlaps between words of P, while those of $HOG(P)$ are *maximal* overlaps between words of P. Given the EHOG of P, to build $HOG(P)$ we need to discard nodes that are not maximal overlaps and to merge the arcs entering and going out of such nodes. This processing can be performed in linear time on the size of $EHOG(P)$ provided that the nodes of $HOG(P)$ are known. We present Algorithm 1, which recapitulates this construction procedure. Once $EHOG(P)$ is built, each of its internal node u is equipped with a list $R_l(u)$, whose meaning is explained below. Then, Algorithm 1 calls the key procedure MarkHOG(r) on line 4 to mark the nodes of $HOG(P)$ in a global boolean array denoted bHog, which we store in a *bit vector*. The procedure for MarkHOG is given in Algorithm 2. Once done, it contracts $EHOG(P)$ in place to obtain $HOG(P)$.

Algorithm 1: *HOG* construction

Input : P a substring free set of words
Output: $HOG(P)$; **Variable**: bHog a bit vector of size $|EHOG(P)|$

1 build $EHOG(P)$
2 set all values of bHog to `False`
3 traverse $EHOG(P)$ to build $R_l(u)$ for each internal node u of $EHOG(P)$
4 run MarkHOG(r) where r is the root of $EHOG(P)$
5 Contract($EHOG(P)$, bHog)
`// Procedure Contract traverses` $EHOG(P)$ `to discard nodes that are not marked in bHog and contract the appropriate arcs`

Meaning of $R_l(u)$ For any internal node u, $R_l(u)$ lists the words of P that admit u as a suffix. Formally stated: $R_l(u) := \{i \in \{1,\dots,|P|\} : u \text{ is suffix of } s_i\}$. As we need to iterate over $R_l(u)$, it is convenient to store it in a list of integers. A traversal of $EHOG(P)$ allows to build a list $R_l(u)$ for each internal node u as stated in [12]. Remark that, while our algorithm processes $EHOG(P)$, Ukkonen's algorithm processes the full trie or Aho Corasik automaton, in which $EHOG(P)$ is embedded. The cumulated sizes of all R_l is linear in $||P||$ (indeed, internal nodes represent different prefixes of words of P and have thus different begin/end positions in those words).

Node of the EHOG and overlaps The following proposition states an important property of EHOG nodes in terms of overlaps. It will allow us not to consider all possible pairs in $P \times P$, when determining maximum overlaps.

Proposition 1 *Let u be an internal node of $EHOG(P)$ and let $i \in R_l(u)$. Then, for any word s_j whose leaf is in the subtree of u, we have $ov(s_i, s_j) \geq |u|$.*

Proof 1 *Indeed, as u belongs to $EHOG(P)$, we get that u is an overlap from s_i to s_j. Thus, their maximal overlap has length at least $|u|$.* **QED**

Description of Algorithm 2 We propose Algorithm 2, a recursive algorithm that determines which nodes belong to $Ov(P)$ while traversing $EHOG(P)$ in a depth first manner, and marks them in bHog. At the end of the algorithm, for any node w of $EHOG(P)$, the entry bHog$[w]$ is `True` if and only if w belongs to $HOG(P)$, and `False` otherwise.

By the definition of a HOG, the leaves of the $EHOG(P)$, which represent the words of P, also belong to $HOG(P)$ (hence, line 3).

The goal is to distinguish among internal nodes those representing maximal overlaps (*i.e.*, nodes of $Ov(P)$) of at least one pair of words. We process internal nodes in order of decreasing word depth. By Proposition 1, we know that for any i in $R_l(u)$, s_i overlaps any s_j whose leaf is in the subtree of u. However, u may not be the maximal overlap for some pairs, since longer overlaps may have been found in each subtree. Indeed, u can be a maximal overlaps from s_i onto some s_j, if and only if for any child v of u, s_i has not already a maximal overlaps with the leaves in the child's subtree. Hence, to check this, we compute the C vector for each child by recursively calling MarkHOG for each child, and merge them with a boolean `and` (line 7). We get invariant #1:

$C[w]$ is `True` iff for any leaf l in the subtree of u the pair $ov(w,l) > |u|$.

Then, we scan $R_l(u)$, for each word w if $C[w]$ is `False`, then for at least one word s_j, the maximum overlap of w onto s_j has not been found in the subtree. By Proposition 1, u is an overlap from w onto s_j, and thus we set both $C[w]$ and bHog$[u]$ to `True` (lines 9-11). Otherwise, if $C[w]$ is `True`, then it remains so, but then bHog$[u]$ remains unchanged. This yields Invariant #2:

$C[w]$ is `True` iff for any leaf l in the subtree of u the pair $ov(w,l) \geq |u|$.

This ensures the correctness of MarkHOG.

The complexity of Algorithm 1 is dominated by the call of the recursive algorithm MarkHOG(r) in line 4, since computing $EHOG(P)$, building the lists $R_l(.)$ and contracting the EHOG into the HOG (line 5) all take linear time in $||P||$.

How many simultaneous calls of MarkHOG can there be? Each call uses a bit vector C of length $|P|$, which impacts the space complexity. The following proposition helps answering this question.

Proposition 2 *Let u, v be two nodes of $EHOG(P)$. Then if u and v belong to distinct subtrees, then MarkHOG(u) terminates before MarkHOG(v) begins, or vice versa.*

Hence, the maximum number of simultaneously running MarkHOG procedures is bounded by the maximum (node) depth of $EHOG(P)$, which is itself bounded by the length of the longest word in P. Now, consider the amortised time complexity of Algorithm 2 over all calls. For each node,

Algorithm 2: MarkHOG(u);

Input : u a node of $EHOG(P)$
Output: C: a boolean array of size $|P|$

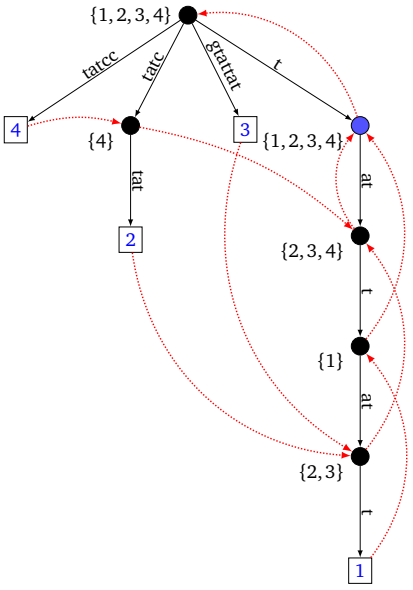
if *u is a leaf* **then**
    set all values of C to `False`
    bHog[u] := `True`
    return C
// Cumulate the information for all children of u
$C :=$ MarkHOG(v) where v is the first child of u
foreach *v among the other children of u* **do**
    $C := C \wedge$ MarkHOG(v)
// Invariant **1**: $C[w]$ is True iff for any leaf s_j in the subtree of u the pair $ov(w, s_j) > |u|$
// Process overlaps arising at node u: Traverse the list $R_l(u)$
for *node x in the list $R_l(u)$* **do**
    if $C[x] =$ `False` **then**
        bHog[u] := `True`
    $C[x]$:= `True`
// Invariant **2**: $C[w]$ is True iff for any leaf s_j in the subtree of u the pair $ov(w, s_j) \geq |u|$
return C

the corresponding C vector is computed once and merged once during the processing of his parent. Moreover, if amortised over all calls of MarkHOG, the processing all the lists R_l for all nodes take linear time in $||P||$. Altogether all calls of Algorithm 2 require $O(||P|| + |P|^2)$ time. The space complexity sums the space occupied by $EHOG(P)$, *i.e.* $O(||P||)$, and that of the C arrays. Now, the maximum number of simultaneously running calls of MarkHOG is the maximum number of simultaneously stored C vectors. This number is bounded by the maximum number of branching nodes on a path from the root to a leaf of $EHOG(P)$, which is smaller than the minimum among $\max\{|s| : s \in P\}$ and $|P|$. Hence, the space complexity is $O(||P|| + |P| \times \min(|P|, \max\{|s| : s \in P\})$.

Theorem 3 *Let P be a set of words. Then Algorithm 1 computes $HOG(P)$ using $O(||P|| + |P|^2)$ time and $O(||P|| + |P| \times \min(|P|, \max\{|s| : s \in P\})$ space. If all words of P have the same length, then the space complexity is $O(||P||)$.*

An example of HOG construction Consider the instance $P := \{tattatt, ctattat, gtattat, cctat\}$; the graph $EHOG(P)$ is shown Figure 2a. Table 2b traces the execution of Algorithm 2 for instance P. It describes for each internal node (leaves not included), the list R_l, the computation of C vector, the final value of bHog and the list of specific pairs of words for which the corresponding node is a maximum overlap. We write $(x/y, z)$ as a shortcut for pairs (x, z) and (y, z). Another example is

given in the web appendix.



(a) EHOG of P

(x,y)	1	2	3	4
1	4	0	0	0
2	6	0	0	0
3	6	0	0	0
4	3	4	0	0

(b) Weights of OG of P in a matrix

node	R_ℓ	C(before)	C(after)	Specific pairs	bHog
ctat	{4}	0000	0001	(4,2)	1
tattat	{2,3}	0000	0110	(2,1) (3,1)	1
tatt	{1}	0110	1110	(1,1)	1
tat	{2,3,4}	1110	1111	(4,1)	1
t	{1,2,3,4}	1111	1111	empty	0
root	{1,2,3,4}	0000 ^ 0001	0000		
root	{1,2,3,4}	0000 ^ 0000	0000	(2/3,2)	
root	{1,2,3,4}	0000 ^ 1111	0000	(1/2/3/4,4)	
root	{1,2,3,4}	0000	1111	(2/3/4,3)	1

(c)

Figure 2: (a) EHOG for instance $P := \{tattatt, ctattat, gtattat, cctat\}$. `goto` transitions appear in black arcs, `Failure Links` in dotted red arcs. For each internal node, the list R_L is given between brackets. (b) Overlap Graph of P given in matrix form. (c) Trace of Algorithm 2. For each internal node are shown: the word it represents, R_l, the bit vector C when before and after the node is processed, bHog, and the pairs for which it is a maximum overlap. The node t is the only internal node which is not a maximal overlap for some pair, and indeed bHog is set to 0. The computation for the root node shows each `and` between C vectors from the children on four lines.

3 Conclusion

The Hierarchical Overlap Graph (HOG) is a compact alternative to the Overlap Graph (OG) since it also encodes all maximal overlaps as the OG, but in a space that is linear in the norm of P. In addition, the HOG records the suffix- and the prefix-relationship between the overlaps, while the OG lacks this information, which is useful for computing greedy superstrings [12, 5]. Because the norm of P can be large in practice, it is thus important to build the HOG also in linear space, which our algorithm achieves if all words have the same length.

For constructing the HOG, Algorithm 1 takes $O(||P|| + |P|^2)$ time. Whether one can compute the HOG in a time linear in $||P|| + |P|$ remains open. An argument against this complexity is that all the information needed to build the Overlap Graph is encoded in the HOG.

The EHOG and HOG differs by definition, and the nodes of the HOG are a subset of the nodes of the EHOG. In practice or in average, is this difference in number of nodes substantial? There

exist instances such that in the limit the ratio between the number of nodes of EHOG versus HOG tends to infinity when $||P||$ tends to infinity while the number of input words remains bounded (see our appendix at http://www.lirmm.fr/~rivals/res/superstring/hog-art-appendix.pdf). For instance, a small alphabet (e.g. DNA) favors multiple overlaps and tend to increase the number of nodes that are proper to the EHOG, while for natural languages HOG and EHOG tend to be equal.

For some applications like DNA assembly, it is valuable to compute approximate rather than exact overlaps. The approach proposed here does not easily extend to approximate overlaps. Some algorithms have been proposed to compute OG with arcs representing approximate overlaps, where approximation is measured by the Hamming or the edit distance [13].

Acknowledgements: This work is supported by the Institut de Biologie Computationnelle ANR (ANR-11-BINF-0002), and Défi MASTODONS from CNRS.

References

[1] A Aho and M Corasick. Efficient string matching: an aid to bibliographic search. *Comm ACM*, 18:333–340, 1975.

[2] Avrim Blum, Tao Jiang, Ming Li, John Tromp, and Mihalis Yannakakis. Linear approximation of shortest superstrings. In *ACM Symposium on the Theory of Computing*, pages 328–336, 1991.

[3] Rodrigo Cánovas, Bastien Cazaux, and Eric Rivals. The Compressed Overlap Index. *CoRR*, abs/1707.05613, 2017.

[4] Bastien Cazaux, Rodrigo Cánovas, and Eric Rivals. Shortest DNA cyclic cover in compressed space. In *Data Compression Conf.*, pages 536–545. IEEE Computer Society Press, 2016.

[5] Bastien Cazaux and Eric Rivals. A linear time algorithm for Shortest Cyclic Cover of Strings. *J. of Discrete Algorithms*, 37:56–67, 2016. http://dx.doi.org/10.1016/j.jda.2016.05.001.

[6] Alexander Golovnev, Alexander S. Kulikov, and Ivan Mihajlin. Solving SCS for bounded length strings in fewer than 2^n steps. *Inf. Proc. Letters*, 114(8):421–425, 2014.

[7] Dan Gusfield, Gad M. Landau, and Baruch Schieber. An efficient algorithm for the All Pairs Suffix-Prefix Problem. *Information Processing Letters*, 41(4):181 – 185, 1992.

[8] Jihyuk Lim and Kunsoo Park. A fast algorithm for the all-pairs suffix–prefix problem. *Theor. Comp. Sci.*, 2017.

[9] Hannu Peltola, Hans Söderlund, Jorma Tarhio, and Esko Ukkonen. Algorithms for some string matching problems arising in molecular genetics. In *IFIP Congress*, pages 59–64, 1983.

[10] Jorma Tarhio and Esko Ukkonen. A greedy approximation algorithm for constructing shortest common superstrings. *Theor. Comp. Sci.*, 57:131–145, 1988.

[11] William H. A. Tustumi, Simon Gog, Guilherme P. Telles, and Felipe A. Louza. An improved algorithm for the all-pairs suffix–prefix problem. *J. of Discrete Algorithms*, 37:34–43, 2016.

[12] Esko Ukkonen. A Linear-Time Algorithm for Finding Approximate Shortest Common Superstrings. *Algorithmica*, 5:313–323, 1990.

[13] Niko Välimäki, Susana Ladra, and Veli Mäkinen. Approximate All-Pairs Suffix/Prefix overlaps. *Inf. Comput.*, 213:49–58, 2012.