Overlap Graphs for Assembling and Scaffolding Algorithms: Paradigm Review and Implementation Proposals

Victor Epain, Rumen Andonov

▶ **To cite this version:**

Victor Epain, Rumen Andonov. Overlap Graphs for Assembling and Scaffolding Algorithms: Paradigm Review and Implementation Proposals. 2022. hal-03878293

HAL Id: hal-03878293
https://inria.hal.science/hal-03878293v1

Preprint submitted on 29 Nov 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Overlap Graphs for Assembling and Scaffolding Algorithms: Paradigm Review and Implementation Proposals

Victor Epain and Rumen Andonov

Univ. Rennes, Inria, IRISA, F-35000 Rennes, France
{victor.epain, rumen.andonov}@irisa.fr

Abstract. Assembling DNA fragments based on their overlaps remains the main assembly paradigm with long DNA fragments sequencing technologies, independently of the aim to resolve only one or several haplotypes. Since an overlap can be seen as a succession relationship between two oriented fragments, the directed graph structure has emerged as an appropriate data structure for handling overlaps. However, this graph paradigm does not appear to take benefit of the reverse symmetry of the orientated fragments and their overlaps, which is a result of blind DNA double-strand sequencing. Thus, the bi-directed graph paradigm was introduced in 1995 towards reducing the graph size by handling the reverse symmetry, and becomes since then the main graph paradigm used in assembly/scaffolding methods. Nevertheless, the available graph paradigms have never been contrasted before, and no implementations have been described. Here we make a complete review on the existing overlap graph paradigms. Furthermore, we present suitable data structures that are theoretically compared in terms of time and memory consumption in the context of the design of some basic graph algorithms. We also show that each one of the paradigms can be switched to another by slightly modifying their data structures.

Keywords: Graph · Reverse symmetry · Overlap-Layout-Consensus

1 Introduction

Double-stranded DNA molecules still cannot be entirely sequenced. In fact, every sequencing technology (sequencer) generates a tremendous amount of overlapping genomic fragments. Each fragment, known as a *read*, is sequenced from one of the two complementary strands. Yet, the sequencers do not provide the one from which a read has been sequenced. Furthermore, only keeping the original reads' sequences can result in a loss of strand pieces during the assembly stage that aims to reconstruct the longest true fragment of one strand. Thus, it may be necessary to reverse-complement the sequence of some reads: this implies considering the reads on the complementary strand and may allow assembling longer strand pieces. However, as both strands are sequenced in blind, previous remark involves considering two *orientations* for each read. Arbitrarily, a

read is defined being in *forward orientation* (denoted by subscribe f) when its sequence is unchanged comparing to the one in the input data, while it is defined being in *reverse orientation* (denoted by subscribe r) when its sequence is reverse-complemented.

Assembly paradigms evolve according to the available data. One of the most famous paradigms is Overlap-Layout-Consensus (OLC). It is well adapted for long fragments and small dataset (order of thousand fragments), as in the case of Sanger sequencing (input data for e.g. Celera [14], ARACHNE [2] and Minimus [18] assembly methods), and more recently such as Oxford Nanopore (ONT) or Pacific Biosciences (PacBio) fragments (input data for e.g. FALCON [4], Canu [10] and hifiasm [3]). OLC first computes pairwise alignment for all the reads to find *overlaps* between them (sufix-prefix alignments). Both orientations (forward/reverse) for each read are considered. Let $\mathcal{R}$ be the set of oriented reads and let $r \in \mathcal{R}$. Denote by $\overline{r}$ its reverse complement. If r overlaps a read $q \in \mathcal{R}$, then $\overline{q}$ overlaps $\overline{r}$ (i.e. each overlap has a reverse too). Finally, the assembly stage aims to order the oriented reads based on their overlaps.

A graph structure is well suited to handle the overlap information. The literature outlines three overlap graph paradigms, reviewed in Section 2. For each one, we propose in Section 3 one or several data structures particularly convenient to manage the reverse symmetry. In Section 4, we analyse their memory consumption and their impact on the design of some fundamental graph algorithms, as well as the corresponding time consumption. Although our three data structures belong to the same time and memory complexity class, we observe particular cases where one of them outperforms the others.

2 Graph Paradigms

In the following we use the notations and definitions summarised in Table 1.

Table 1. List of symbols and their description.

$\mathcal{R}_{aw}$	Raw reads set (reads with their original sequences)
$\mathcal{R}_{ev} = \{\overline{r} \mid r \in \mathcal{R}_{aw}\}$	Reverse reads set (reads with their sequence reverse-complemented)
$\mathcal{R} = \mathcal{R}_{aw} \cup \mathcal{R}_{ev}$	The entire set of oriented reads (forward and reverse reads)
$\mathcal{O} \subset \mathcal{R} \times \mathcal{R}$	Overlaps set (pairs of oriented reads)
$G = (V, E)$	Graph symbol where V is the set of vertices and E the set of edges
N, N^-, N^+	Set of neighbours, predecessors and successors

Table 2 outlines the notations we use and the main properties of each paradigm. Figure 1 summarises overlaps visualisation according to each graph paradigm.

Table 2. Overview of graph paradigms.

Year	Graph paradigm	Graph type	$\|V\|$	$\|E\|$
1991 [9]	Oriented fragments based (DG)	directed	$2 \times \|\mathcal{R}_{aw}\|$	$\|\mathcal{O}\|$
1995 [13]	Oriented walk based (BG)	bi-directed	$\|\mathcal{R}_{aw}\|$	$\frac{1}{2} \times \|\mathcal{O}\|$
2005 [15]	Tail-head fragments based (UG)	undirected	$2 \times \|\mathcal{R}_{aw}\|$	$\|\mathcal{R}_{aw}\| + \frac{1}{2} \times \|\mathcal{O}\|$

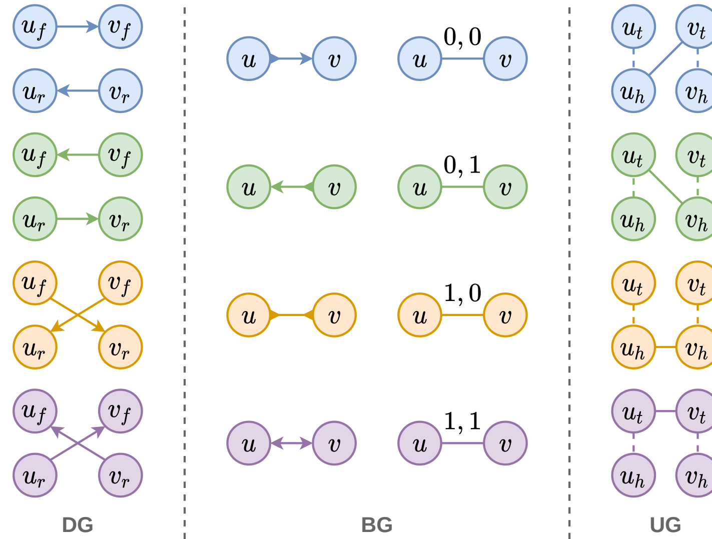

Fig. 1. Overlap cases visualisation. u and v denote the identifiers of two reads. Each vertical dashed line separates two graph paradigms. Each colour is associated with an overlap case (and its reverse). The blue colour corresponds to the overlap (u_f, v_f) (and its reverse (v_r, u_r)) — the green to the overlap (v_f, u_f) (and (u_r, v_r)) — the orange to the overlap (u_f, v_r) (and (v_f, u_r)) — the violet to the overlap (v_r, u_f) (and (u_r, v_f)). The first column represents the overlap cases for the DG paradigm (c.f. Section 2.1). The BG paradigm is represented by the two columns in the middle. The first one gives the bi-directed view while the second one gives the undirected graph view, where the edge's attributes correspond to *or* and *rel* boolean values (c.f. Section 2.2). The last column depicts the overlaps in the UG paradigm: the plain edges are overlap-edges while the dotted edges are read-edges (c.f. Section 2.3).

2.1 Directed Graph (DG): Oriented Fragments Based

In this graph paradigm there are two vertices for each read, one for the forward orientation, and one for the reverse orientation. Each overlap (r, q) in the overlaps set $\mathcal{O}$ is associated with an oriented edge from r to q. Moreover, $(r, q) \in \mathcal{O} \iff (\overline{q}, \overline{r}) \in \mathcal{O}$ (and hence there exists an edge from $\overline{q}$ to $\overline{r}$). As a consequence, this paradigm is defined as an *oriented fragments based* one, because each vertex represents a read for a fixed orientation. The first column in Figure 1 shows how overlaps are visualised in this paradigm.

First mentioned in 1991 by Kececioglu [9], the directed graph structure raises as the more natural one to handle the oriented reads and their overlaps. Later, it was also used by Chin et al. [4], Kamath et al. [8], Andonov et al. [1], Shafin et al. [17] and Cheng et al. [3] in their respective assembly methods.

2.2 Bi-directed Graph (BG): Oriented Walk Based

In order to avoid creating two vertices for each read (and thus creating two edges for each overlap), the bi-directed graph structure for storing overlaps was first employed by E. W. Myers in 1995 [13]. The key idea is to represent each read with only one vertex and to keep the strict necessary overlap information between two reads on the edge connecting two vertices. This structure was also used by Sommer et al. [18], Hernandez et al. [7] and Salmela et al. [16].

In memory, the bi-directed graph is undirected. Let $(u, v) \in E$ be an edge, and without loss in generality suppose that u identifier is smaller than this one of v. This edge has two attributes:

$$or_{uv} = \begin{cases} 0 & \text{if the orientations of } u \text{ and } v \text{ are the same in the overlap} \\ 1 & \text{otherwise} \end{cases}$$

and

$$rel_{uv} = \begin{cases} 0 & \text{if } u_f \text{ overlaps } v_f \text{ or } u_f \text{ overlaps } v_r \\ 1 & \text{otherwise} \end{cases}$$

This paradigm is defined as an *oriented walk graph based*. Indeed, given a vertex enriched by an orientation value, it is necessary to verify for each of its oriented neighbours if they are the successors or predecessors (determined by rel_{uv}). Also, the rel_{uv} attribute depends on the vertex with the smaller identifier.

The second column of Figure 1 shows how each overlap case is visualised in this graph, both for bi-directed and undirected views.

2.3 Undirected Graph (UG): Tail-Head Fragments Based

A new undirected graph structure was presented by E. W. Myers in 2005 [15]: one read is represented here by its tail and its head. This representation was also described in the Mäkinen et al.'s book [12] and in Li [11].

Both the tail and the head are vertices, and there is one edge from the tail to the head. Passing through the tail first and then the head corresponds to choose the read in forward orientation, while traversing the head first and then the tail corresponds to choose it in its reverse orientation. This new type of edges are called read-edges, at the opposite of overlap-edges that correspond to overlaps. A valid walk in this graph must alternate between read-edges and overlap-edges, starting from and finishing by a read-edge.

The last column of Figure 1 shows overlaps visualisation in this graph.

3 Graph Implementations

For each graph paradigm in Section 2 we propose at least one implementation in Sections 3.1 to 3.3.

In the sequel, let consider that all overlaps in $\mathcal{O}$ are kept, and that any of the reads in $\mathcal{R}_{aw}$ is involved in at least one overlap. Assume also that $\min_{r \in \mathcal{R}_{aw}} r_{rid} = 0$ and $\max_{r \in \mathcal{R}_{aw}} r_{rid} = |\mathcal{R}_{aw}| - 1$ hold, where r_{rid} denotes the identifier of read r. These assumptions permit to easily build an index set with read identifiers and an index set for the overlaps in the interval $[\![0, |\mathcal{O}| - 1]\!]$.

The aim here is to find implementations that respect the requirements below:

1. Querying requirements
   - given an oriented read, getting all oriented reads overlapping it
   - given two oriented reads, answering true if and only if they overlap
2. Dynamic requirements
   - adding a read/overlap
   - deleting a read/overlap

Given all reads, all the overlaps can be represented in a squared sparse matrix (of size $\mathcal{R}^2$). Sparse matrix compression by row or by column is known for efficiently storing the matrix and enabling fast overlap existence querying. Finally, they enable fast edges iteration, but does not satisfy efficiently requirements 2 and can not be immediately adapted for handling overlaps reverse symmetry.

Thus, we decide to implement the paradigms with adjacency list structures as they appear to be better suited to dynamically adding vertices and edges, even if they use a large amount of RAM due to the neighbour lists' pointers.

3.1 Directed Graph (DG): Oriented Fragments Based

For each read $r \in \mathcal{R}_{aw}$, there are two vertices $v_f, v_r \in V$ in the directed graph. Remind that each read identifier r_{rid} corresponds to a unique integer identifier. Therefore, the index of v_f equals to $2 \times r_{rid}$ and this of v_r equals to $2 \times r_{rid} + 1$.

The first idea is to directly build the two adjacency lists: one for the predecessors, another one for the successors. They contain the index of each predecessor, respectively of each successor vertices — as in the standard directed graph implementation. They also contain the edges' indices.

However, the above strategy does not take benefits from the reverse symmetry. The following proposals demonstrate that the needed memory can be approximately divided by two.

Only Oriented Fragments' Successors Directed Graph (DGS) This implementation is defined as above, except the predecessor lists are omitted. Indeed, the reverse symmetry allows retrieving the predecessors at a low algorithmic cost. Let $v \in V$ be a vertex. Then for each $(\overline{v}, \overline{u}) \in E$, $\overline{u}$ is in the successor list of $\overline{v}$. By reverse symmetry, $(\overline{v}, \overline{u}) \in E \iff (u, v) \in E$: so the reverse of the successors of the reverse vertex are the predecessors of the vertex.

Forward Fragments Directed Graph (DGF) DGF implementation requires both the predecessor and the successor lists, but they are needed for the forward reads only. In fact, the lists are not required for the reverse reads. Thus, it is sufficient to build a vertices index in the range 0 to $|\mathcal{R}_{aw}| - 1$. For each vertex, neighbour lists give the neighbours' indices and their orientation (a boolean value). They also provide edges' indices, as for DGS.

Figure 2 reports how overlaps are represented. To save space, the edges' indices are omitted.

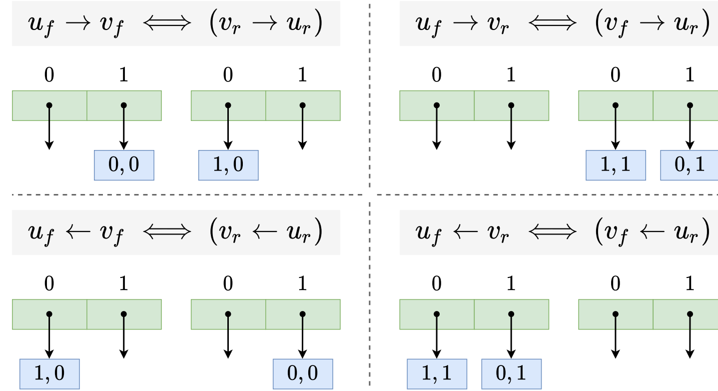

Fig. 2. Forward fragments directed graph implementation. Each mathematical formula provides the overlap case and its reverse symmetric under parenthesis. The predecessor (left) and the successor (right) lists for the forward orientation are visualised below. They contain the couples index-orientation of the predecessors/successors. Index of u equals 0, index of v equals 1. Forward/reverse orientation is represented by 0/1.

The reverse symmetry allows retrieving the predecessors and the successors of the reverse reads at a low algorithmic cost (c.f. Table 4). Let $v \in V$ be a vertex such that its orientation is reverse. Then for each $(\overline{v}, \overline{u}) \in E$, $\overline{u}$ is in the successor list of $\overline{v}$ (which is in forward orientation). By reverse symmetry,

$(\overline{v}, \overline{u}) \in E \iff (u, v) \in E$: so the reverse of the successors of the reverse vertex are the predecessors of the vertex. Retrieving the successors follows the same logic: the reverse of the predecessors of the reverse vertex are the successors of the vertex.

To conclude, DGS consumes one pointer less than DGF. Furthermore, the algorithmic cost because of the memory reduction related to the reverse symmetry is subtly different: for DGS an additional cost is necessary when the predecessors have to be returned for an oriented vertex (forward or reverse) while for DGF it is necessary when either the predecessors or the successors have to be returned for the vertex with reverse orientation. In practice, DGS should be preferred as there is no additional cost until the successors of oriented vertices have to be returned.

3.2 Bi-directed Graph (BG): Oriented Walk Based

Undirected Bi-directed Graph (BGU) For each read there is one vertex (that represents an unoriented read or a not-yet oriented read as it depends on the walk, see Section 2.2). The two boolean edge's attributes *or* and *rel* allow to differentiate an overlap from its reverse, while they are associated to only one edge.

Figure 3 reports how overlaps are represented. To save space, the edges' indices are omitted.

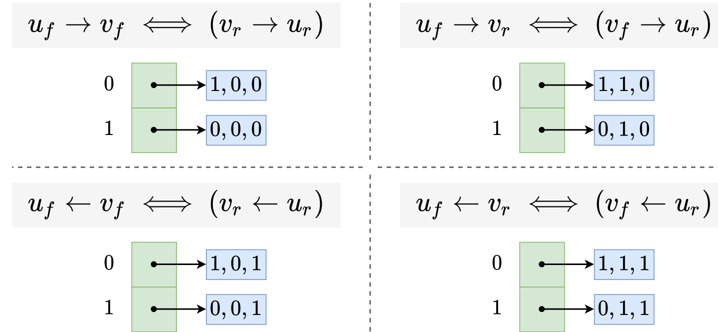

Fig. 3. Bi-directed graph implementation. Each mathematical formula provides the overlap case and its reverse symmetric under parenthesis. The neighbour lists for the unoriented vertices are visualised below. Each tuple in the lists contains: first the neighbour index (an integer), then the *or* and finally the *rel* edge attributes (booleans). Index of u equals 0, index of v equals 1: so for each edge (u, v), its attribute rel_{uv} indicates whether u_f overlaps v_f or u_f overlaps v_r.

The four combinations of edges' attributes *or* and *rel* enable the representation of all the overlap cases and their reverse without redundancy. To verify the

reverse symmetry, it is sufficient (and easier) to choose an orientation of u and to calculate the overlap case with the edges' attributes.

It is possible to take benefits from the undirected graph structure and reduce by two the memory consumption of the edges indices. In fact, the index of the edge e can be written $e_{ind} = 2 \times e'_{ind} + r$, where $r \in \{0; 1\}$. In the neighbour lists, e'_{ind} replaces e_{ind}. Therefore, the orientation of the smaller vertex index (say u) is used to retrieve the original edge index during a walk in the graph. If the considered overlap uses u_f then $e_{ind} = 2 \times e'_{ind}$, otherwise $e_{ind} = 2 \times e'_{ind} + 1$.

Bi-directed Graph to Directed Graph (From BGU to DGF) Each neighbour list can be split into two parts (that requires to allocate fix memory size before): the first part corresponds to the predecessors, while the second one to the successors, and both are described only for the forward reads. Thus, *or* and *rel* edges attributes disappear, and each tuple in the neighbour lists contains first the index of the neighbour and its orientation as in DGF. Finally, the edges indices must be calculated to retrieve the definition in DGF.

3.3 Undirected Graph (UG): Tail-Head Fragments Based

For each read $r \in \mathcal{R}_{aw}$, there are two vertices $v_t, v_h \in V$ in the undirected graph. Assume that the index of v_h equals to $2 \times r_{rid}$ and this one of v_t equals to $2 \times r_{rid} + 1$. The undirected graph may follow the standard implementation.

However, a valid walk in UG alternates between read-edges and overlap-edges (see Figure 1 and Section 2.3). Thus, it is possible to break the undirected graph paradigm by replacing the index of each vertices' neighbour by its extremities ($u_t \rightleftharpoons u_h$). Interestingly, the transformation of the neighbour lists results in DGS implementation (u_t is equivalent to u_r, u_h is equivalent to u_f).

4 Memory & Algorithmic Costs

Here we report and compare the memory and the algorithmic cost of implementations DGS, DGF (Section 3.1) and BGU (Section 3.2). The complexity analysis and the complete algorithms can be found in our working paper [6].

Table 3 shows that BGU is better in terms of memory because it consumes half as many pointers as DGS and DGF (the latters differ by one pointer).

In the sequel, $o_v = o_v^- + o_v^+$, where o_v^- and o_v^+ correspond to the number of reads that overlap, respectively are overlapped by, v. Moreover, z denotes the read with the largest index.

Iterating over the overlaps of an oriented read (i.e. iterating over the neighbours of a vertex — enriched by an orientation for DGF and BGU) is the most fundamental algorithm. According to Table 4, DGS and DGF are much better than BGU for this task. The weakness of DGS comparing to DGF appears when the predecessors are required. But DGS outperforms DGF for the successors.

Concerning the algorithms for structure dynamics, the cost of adding a vertex in the graph is equivalent for all of them. In contrast, DGS is the best for adding

Table 3. Memory consumption (in octets) of the implementations. P is the memory size of a memory address.

DGS	$(2 \times \lvert\mathcal{R}_{aw}\rvert + 1) \times P + \lvert\mathcal{O}\rvert \times \left(\left\lceil \frac{1 + \log_2\lvert\mathcal{R}_{aw}\rvert}{8} \right\rceil + \left\lceil \frac{\log_2\lvert\mathcal{O}\rvert}{8} \right\rceil\right)$
DGF	$2 \times (\lvert\mathcal{R}_{aw}\rvert + 1) \times P + \lvert\mathcal{O}\rvert \times \left(\left\lceil \frac{1 + \log_2\lvert\mathcal{R}_{aw}\rvert}{8} \right\rceil + \left\lceil \frac{\log_2\lvert\mathcal{O}\rvert}{8} \right\rceil\right)$
BGU	$(\lvert\mathcal{R}_{aw}\rvert + 2) \times P + \lvert\mathcal{O}\rvert \times \left(\left\lceil \frac{\log_2\lvert\mathcal{R}_{aw}\rvert}{8} \right\rceil + \left\lceil \frac{\log_2\lvert\mathcal{O}\rvert - 1}{8} \right\rceil\right) + \left\lceil \frac{\lvert\mathcal{O}\rvert}{8} \right\rceil$

Table 4. Algorithmic costs of iterating over the neighbours for DGS, DGF and BGU.

	Iterate over the predecessors			**Iterate over the successors**		
	Best	**Worst**	**Average**	**Best**	**Worst**	**Average**
DGS		$5 \times o_v^- + 2$			o_v^+	
DGF	$o_v^- + 1$	$4 \times o_v^- + 3$	$2.5 \times o_v^- + 2$	$o_v^+ + 1$	$4 \times o_v^+ + 3$	$2.5 \times o_v^+ + 2$
BGU	$3 \times o_v$	$6 \times o_v$	$4.75 \times o_v$	$3 \times o_v$	$6 \times o_v$	$4.75 \times o_v$

Table 5. Algorithmic costs of adding a vertex or an edge for DGS, DGF and BGU.

	Add a vertex			**Add an edge**		
	Best	**Worst**	**Average**	**Best**	**Worst**	**Average**
DGS		5			8	
DGF		5		6	10	8
BGU		3			9	

Table 6. Average algorithmic costs of deleting an edge for DGS, DGF and BGU.

DGS	$3 \times \left(\left\lceil \frac{o_u^+}{2} \right\rceil + \left\lceil \frac{o_v^-}{2} \right\rceil\right) + 13 - \frac{1}{o_u^+} - \frac{1}{o_v^-}$
DGF	$\frac{3}{2} \times \left(\left\lceil \frac{o_u^-}{2} \right\rceil + \left\lceil \frac{o_u^+}{2} \right\rceil + \left\lceil \frac{o_v^-}{2} \right\rceil + \left\lceil \frac{o_v^+}{2} \right\rceil\right) + 13 - 2 \times \left(\frac{1}{o_u} + \frac{1}{o_v}\right)$
BGU	$3 \times \left(\left\lceil \frac{o_u}{2} \right\rceil + \left\lceil \frac{o_v}{2} \right\rceil\right) + 11 - \frac{1}{o_u} - \frac{1}{o_v}$

an edge (see Table 5). Finally, DGS is better than DGF which is better than BGU for deleting an edge (Table 6) while BGU is much better than DGF and DGS for deleting a vertex (Table 7). BGU is the best for this task thanks to the undirected graph structure.

Table 7. Approximated average cost for deleting a vertex for DGS, DGF and BGU.

$10 \times o_v + \frac{10 \times \mathcal{R}_{aw} - 5}{2 \times \mathcal{R}_{aw}} \times o_z + 3 \times \left(\sum_{u \in N_v^-} \left\lceil \frac{o_u^+}{2} \right\rceil + \sum_{w \in N_v^+} \left\lceil \frac{o_w^-}{2} \right\rceil \right) + \frac{6 \times \mathcal{R}_{aw} - 3}{2 \times \mathcal{R}_{aw}} \times \left(\sum_{x \in N_z^-} \left\lceil \frac{o_x^+}{2} \right\rceil + \sum_{y \in N_z^+} \left\lceil \frac{o_y^-}{2} \right\rceil \right)$
$\frac{15}{2} \times o_v + \frac{6 \times \mathcal{R}_{aw} - 3}{2 \times \mathcal{R}_{aw}} \times o_z + \frac{3}{2} \times \sum_{u \in N_v^- \cup N_v^+} \left(\left\lceil \frac{o_u^-}{2} \right\rceil + \left\lceil \frac{o_u^+}{2} \right\rceil \right) + \frac{6 \times \mathcal{R}_{aw} - 3}{4 \times \mathcal{R}_{aw}} \times \sum_{x \in N_z^- \cup N_z^+} \left(\left\lceil \frac{o_x^-}{2} \right\rceil + \left\lceil \frac{o_x^+}{2} \right\rceil \right)$
$5 \times o_v + \frac{3 \times \mathcal{R}_{aw} - 3}{\mathcal{R}_{aw}} \times o_z + 3 \times \sum_{w \in N_v} \left\lceil \frac{o_w}{2} \right\rceil + \frac{3 \times \mathcal{R}_{aw} + 3}{2 \times \mathcal{R}_{aw}} \times \sum_{y \in N_z} \left\lceil \frac{o_y}{2} \right\rceil$

5 Conclusion & Discussion

The Overlap-Layout-Consensus paradigm to assemble the genomes is based on pairwise comparisons of all the reads to find suffix-prefix alignments that are denoted by overlaps. As these overlaps correspond to succession relationships between the reads, a graph structure should be adapted to store them, by taking benefits from the fact that a read is associated with its reverse. In this paper, our will is to clarify and to formalise the graph structures that have been described and suggested in the fragment assembly's literature.

Three overlap graph paradigms have emerged. While the directed graph highlights visually the double strand sequencing, its weakness is that it requires two vertices for each read [9]. The bi-directed graph has been introduced in 1970 [5] but has been firstly employed to store overlaps by E. W. Myers [13]. The key idea is to aggregate the two orientations of a read into only one vertex, and two overlaps into only one edge. Finally, the undirected graph paradigm associates one vertex for each of the two extremities of a read in order to simplify the graph traversal [15]. The edges set is partitioned into two parts: the set of read-edges (there is an edge between the two vertices associated to the extremities of a read) and the set of overlap-edges.

Since just counting the number of vertices and the number of edges of the paradigms' visualisations is not sufficient, here we propose and compare several implementations. DGS, DGF and BGU data structures are based on adjacency lists, as they allow adding and deleting elements more efficiently than for compressed sparse matrix.

Furthermore, we describe how the bi-directed graph paradigm can be transformed to the directed paradigm by simple adaptations of BGU to DGF. We also show that jumping the read-edges in the undirected graph is equivalent to implement DGS.

Finally, if memory is the major issue, then the BGU implementation should be preferred. Otherwise, DGS is recommended for iterating over the neighbours, for adding and deleting edges, while BGU is preferable for deleting vertices.

References

1. Andonov, R., Djidjev, H., François, S., Lavenier, D.: Complete assembly of circular and chloroplast genomes based on global optimization. Journal of Bioinformatics and Computational Biology **17**(3), 1950014 (Jun 2019). https://doi.org/10.1142/S0219720019500148
2. Batzoglou, S., Jaffe, D.B., Stanley, K., Butler, J., Gnerre, S., Mauceli, E., Berger, B., Mesirov, J.P., Lander, E.S.: ARACHNE: A whole-genome shotgun assembler. Genome Research **12**(1), 177–189 (Jan 2002). https://doi.org/10.1101/gr.208902
3. Cheng, H., Concepcion, G.T., Feng, X., Zhang, H., Li, H.: Haplotype-resolved de novo assembly using phased assembly graphs with hifiasm. Nature Methods **18**(2), 170–175 (Feb 2021). https://doi.org/10.1038/s41592-020-01056-5
4. Chin, C.S., Peluso, P., Sedlazeck, F.J., Nattestad, M., Concepcion, G.T., Clum, A., Dunn, C., O'Malley, R., Figueroa-Balderas, R., Morales-Cruz, A., Cramer, G.R., Delledonne, M., Luo, C., Ecker, J.R., Cantu, D., Rank, D.R., Schatz, M.C.: Phased diploid genome assembly with single-molecule real-time sequencing. Nature Methods **13**(12), 1050–1054 (Dec 2016). https://doi.org/10.1038/nmeth.4035
5. Edmonds, J., Johnson, E.L.: Matching: A well-solved class of integer linear programs. In: In: Combinatorial Structures and Their Applications (Gordon and Breach. pp. 89–92 (1970)
6. Epain, V., Andonov, R.: Overlap graph for assembling and scaffolding algorithms: Paradigm review and implementation proposals (Oct 2022)
7. Hernandez, D., François, P., Farinelli, L., Østerås, M., Schrenzel, J.: De novo bacterial genome sequencing: Millions of very short reads assembled on a desktop computer. Genome Research **18**(5), 802–809 (Jan 2008). https://doi.org/10.1101/gr.072033.107
8. Kamath, G.M., Shomorony, I., Xia, F., Courtade, T.A., Tse, D.N.: HINGE: Long-read assembly achieves optimal repeat resolution. Genome Research **27**(5), 747–756 (Jan 2017). https://doi.org/10.1101/gr.216465.116
9. Kececioglu, J.D.: Exact and Approximation Algorithms for DNA Sequence Reconstruction. The University of Arizona (1991)
10. Koren, S., Walenz, B.P., Berlin, K., Miller, J.R., Bergman, N.H., Phillippy, A.M.: Canu: Scalable and accurate long-read assembly via adaptive k-mer weighting and repeat separation. Genome Research **27**(5), 722–736 (Jan 2017). https://doi.org/10.1101/gr.215087.116
11. Li, H.: Minimap and miniasm: Fast mapping and de novo assembly for noisy long sequences. Bioinformatics **32**(14), 2103–2110 (Jul 2016). https://doi.org/10.1093/bioinformatics/btw152
12. Mäkinen, V., Belazzougui, D., Cunial, F., Tomescu, A.I.: Genome-Scale Algorithm Design. Cambridge University Press (May 2015)
13. Myers, E.W.: Toward simplifying and accurately formulating fragment assembly. Journal of Computational Biology: A Journal of Computational Molecular Cell Biology **2**(2), 275–290 (1995). https://doi.org/10.1089/cmb.1995.2.275

14. Myers, E.W., Sutton, G.G., Delcher, A.L., Dew, I.M., Fasulo, D.P., Flanigan, M.J., Kravitz, S.A., Mobarry, C.M., Reinert, K.H., Remington, K.A., Anson, E.L., Bolanos, R.A., Chou, H.H., Jordan, C.M., Halpern, A.L., Lonardi, S., Beasley, E.M., Brandon, R.C., Chen, L., Dunn, P.J., Lai, Z., Liang, Y., Nusskern, D.R., Zhan, M., Zhang, Q., Zheng, X., Rubin, G.M., Adams, M.D., Venter, J.C.: A whole-genome assembly of Drosophila. Science (New York, N.Y.) **287**(5461), 2196–2204 (Mar 2000). https://doi.org/10.1126/science.287.5461.2196
15. Myers, E.W.: The fragment assembly string graph. Bioinformatics **21**(suppl_2), ii79–ii85 (Jan 2005). https://doi.org/10.1093/bioinformatics/bti1114
16. Salmela, L., Mäkinen, V., Välimäki, N., Ylinen, J., Ukkonen, E.: Fast scaffolding with small independent mixed integer programs. Bioinformatics **27**(23), 3259–3265 (Dec 2011). https://doi.org/10.1093/bioinformatics/btr562
17. Shafin, K., Pesout, T., Lorig-Roach, R., Haukness, M., Olsen, H.E., Bosworth, C., Armstrong, J., Tigyi, K., Maurer, N., Koren, S., Sedlazeck, F.J., Marschall, T., Mayes, S., Costa, V., Zook, J.M., Liu, K.J., Kilburn, D., Sorensen, M., Munson, K.M., Vollger, M.R., Monlong, J., Garrison, E., Eichler, E.E., Salama, S., Haussler, D., Green, R.E., Akeson, M., Phillippy, A., Miga, K.H., Carnevali, P., Jain, M., Paten, B.: Nanopore sequencing and the Shasta toolkit enable efficient de novo assembly of eleven human genomes. Nature Biotechnology **38**(9), 1044–1053 (Sep 2020). https://doi.org/10.1038/s41587-020-0503-6
18. Sommer, D.D., Delcher, A.L., Salzberg, S.L., Pop, M.: Minimus: A fast, lightweight genome assembler. BMC bioinformatics **8**, 64 (Feb 2007). https://doi.org/10.1186/1471-2105-8-64