



**HAL**  
open science

# Overlap Graph for Assembling and Scaffolding Algorithms: Paradigm Review and Implementation Proposals

Victor Epain

► **To cite this version:**

Victor Epain. Overlap Graph for Assembling and Scaffolding Algorithms: Paradigm Review and Implementation Proposals. 2022. hal-03815190v2

**HAL Id: hal-03815190**

**<https://inria.hal.science/hal-03815190v2>**

Preprint submitted on 22 Oct 2022 (v2), last revised 26 Oct 2022 (v3)

**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 Graph for Assembling and Scaffolding Algorithms: Paradigm Review and Implementation Proposals

Victor Epain

Univ. Rennes, IRISA/CNRS, Inria, F-35000 Rennes, France

22nd October 2022

## Abstract

---

Assembling *Deoxyribonucleic Acid (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 the more appropriate data structure for handling *overlaps*. However, this *graph* paradigm did 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 to be the one that reduces the *graph* size by handling the *reverse* symmetry, and since becomes the mainly used *graph* paradigm. Nevertheless, *graph* paradigms have never been contrasted before, and no implementations were described. Here we make a complete review on the existing *overlap graph* paradigms. Furthermore, we present different implementations that are theoretically compared in terms of memory, and their impact on the design and on the time of some basic *graph* algorithms. We also show that by adapting close logic implementations, a *graph* paradigm can be switched to another.

## Keywords

---

*Directed graph · Overlaps*

---



# Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
1.1	Reads . . . . .	5
1.2	Overlaps . . . . .	7
<b>2</b>	<b>Storing Overlaps in a Graph Structure</b>	<b>9</b>
2.1	Directed Graph: Oriented Fragments Based . . . . .	11
2.2	Bi-directed Graph: Oriented Walk Based . . . . .	14
2.3	Undirected Graph: Tail-Head Fragments Based . . . . .	18
<b>3</b>	<b>Overlap Graph Implementations</b>	<b>22</b>
3.1	Directed Graph: Oriented Fragments Based . . . . .	22
3.1.1	All Oriented Fragments Directed Graph (DGA) . . . . .	22
3.1.2	Only Oriented Fragments' Successors Directed Graph (DGS) . . . . .	25
3.1.3	Forward Fragments Directed Graph (DGF) . . . . .	27
3.2	Bi-directed Graph: Oriented Walk Based . . . . .	31
3.2.1	Forward Fragments Bi-directed Graph (BG) . . . . .	31
3.2.2	Forward Fragments Directed Graph (DGF) . . . . .	33
3.3	Undirected Graph: Tail-Head Fragments Based . . . . .	33
3.3.1	All Oriented Fragments Undirected Graph (UGA) . . . . .	33
3.3.2	Read-edges Jump Directed Graph (DGS) . . . . .	35
<b>4</b>	<b>Memory and Time Costs</b>	<b>35</b>
4.1	Algorithms . . . . .	36
4.1.1	Iterating Over the Predecessors . . . . .	36
4.1.2	Iterating Over the Successors . . . . .	37
4.1.3	Adding a Vertex . . . . .	39
4.1.4	Adding an Edge . . . . .	40
4.1.5	Deleting a Vertex . . . . .	41
4.1.6	Deleting an Edge . . . . .	45
4.2	Time Complexities . . . . .	48
	References	<b>53</b>
	Acronyms	<b>i</b>
	Symbols	<b>i</b>
	Glossary	<b>ii</b>



# 1 Introduction

## What is complementary strands?

Double-stranded [Deoxyribonucleic Acid \(DNA\)](#) molecules still cannot be entirely sequenced. In fact, every *sequencing* technology (*sequencer*) generates numerous overlapping genomic *fragments*.

## 1.1 Reads

A [DNA fragment](#) obtained from a *sequencing* stage is denoted by *read*.

### ► Definition 1.1: Raw reads

Let  $\mathcal{R}_{aw}$  be the set of raw [reads](#).  $\forall r \in \mathcal{R}_{aw}$ , let define the following attributes:

- $r_{rid} \in \mathbb{N}$ , the [read](#)'s integer identifier (index)
- $r_{seq} \in \Sigma^+$  the [read](#)'s sequence, where  $\Sigma = \{A, T, G, C\}$  is *nucleotide* alphabet

A [read](#) can be sequenced from one *strand* or from its complement.

### ► Axiom 1.1: Double-stranded DNA sequencing

- i. The two [DNA strands](#) are both sequenced.
- ii. A [strand](#) is read in reverse order to the order of its complementary [strand](#).
- iii. Given two [reads](#), it is not possible to know *a priori* it they have been sequenced from the same [strand](#).

The Definition 1.2 formalises Axiom 1.1 point iii.:

### ► Definition 1.2: Strand identifier

Let  $\text{strand\_iid}: \mathcal{R}_{aw} \rightarrow \{0;1\}$  be the function which returns the [DNA strand](#) identifier for a [read](#).

Then Axiom 1.1 point iii. can be mathematically formalised:

$$\exists (q, r) \in \mathcal{R}_{aw}^2, q \neq r \mid \text{strand\_iid}(q) \neq \text{strand\_iid}(r)$$

Axiom 1.1 suggests that combining two reads may result in a chimaera mix of the two strands because all the reads have not been sequenced from the same strand. Furthermore, only keeping the original sequence of the reads can result in the loss of strand parts during the *assembly* stage that aims to reconstruct the longest true parts of one strand. To solve these issues, it is sometimes necessary to consider the complementary sequence of some reads (i.e. the sequences on the complementary strand). For this, the sequence of the concerned reads are reversed (the end becomes the beginning, and vice versa) and each of its nucleotides is complemented ( $A \rightleftharpoons T, G \rightleftharpoons C$ ). Arbitrarily, the original sequence of a read is denoted as the *forward*, while the *reverse* denotes the reverse-complement. In the following, determining what sequence is taken for a read is defined as *orienting* it. Finally, it is necessary to consider both the forward and the reverse orientation for all the reads. Figure 1 explains the meaning of forward and reverse orientations in the context of sequencing technology.

► **Definition 1.3: Reversed reads**

Let  $\mathcal{R}_{ev} = \{u_r \mid u_f \in \mathcal{R}_{aw}\}$  denote the set of reversed reads. Thus  $\mathcal{R} = \mathcal{R}_{aw} \cup \mathcal{R}_{ev}$ . Let  $u \in \mathcal{R}_{aw}$  be a read. Subscript is added to  $u$  to precise its orientation:

- $u_f$  means that  $u$  is read in forward orientation (i.e. the original sequence of  $u$ , hence  $u_f \in \mathcal{R}_{aw}$ )
- $u_r$  means that  $u$  is read in reverse orientation (i.e. the reverse-complemented sequence of  $u$ , hence  $u_r \in \mathcal{R}_{ev}$ )

Now the set of all oriented reads  $\mathcal{R}$  was built, the reverse operation can be defined:

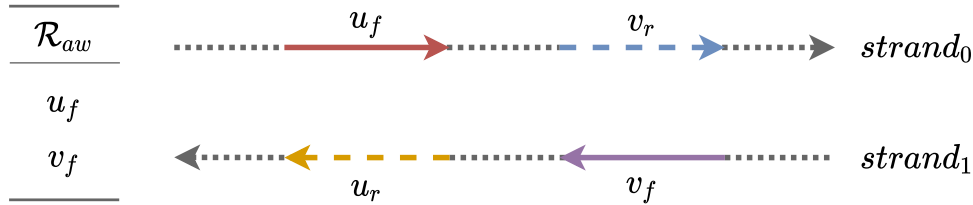
► **Definition 1.4: The reverse operation for reads**

For a given read  $u \in \mathcal{R}$ ,  $\bar{u}$  denotes its reverse, where:

- $\bar{u}_{rid} = u_{rid}$  (i.e. a read  $u$  and its reverse  $\bar{u}$  have the same identifier)
- $\bar{u}_{seq} = \overline{u_{seq}} = \overline{b_1 b_2 \cdots b_n} = \overline{b_{n-1}} \cdots \overline{b_1}$  where  $n \in \mathbb{N}$  is the length of the nucleotides sequence, and  $\forall i \in \llbracket 1; n \rrbracket, b_i$  is the  $i^{th}$  nucleotide base and

$\bar{b}_i$  its complementary nucleotide base.

Note that  $\overline{u_f} = u_r$  and  $\overline{u_r} = u_f$ .



■ **Figure 1 – Double-stranded DNA sequencing.**

$u_f, v_f$  are two reads from the raw reads set  $\mathcal{R}_{aw}$  obtained from a double-stranded DNA sequencing stage. By default,  $f$  (forward) denotes the reading orientation of the raw reads' sequence that gives their sequence, while  $r$  (reverse) denotes the reading orientation of the raw reads' sequence that gives their reversed complemented sequence. In this illustration case,  $u_f$  was sequenced from strand  $strand_0$  ( $\text{strand\_iid}(u_f) = 0$ ) while  $v_f$  was sequenced from strand  $strand_1$  ( $\text{strand\_iid}(v_f) = 1$ ).

► **Axiom 1.2: Belonging to one strand**

The definition of  $\mathcal{R}$  set and Axiom 1.1 point ii. give the following statement:

$$\forall r \in \mathcal{R}, \text{strand\_iid}(r) = 1 - \text{strand\_iid}(\bar{r})$$

## 1.2 Overlaps

Reads must be assembled to reconstruct the entire genome since they are shorter than the length of the strand from which they have been sequenced (see Axiom 1.1). To assemble the reads means to find the original order of the (oriented) reads. Thus, the assembly stage is at least based on oriented reads alignments that define a succession relationship between them. One alignment type that answers this issue is overlap.

► **Definition 1.5: Overlap**

- An overlap between two nucleotides sequences  $(u, v)$  is a semi-global alignment or a close mapping between the end of sequence  $u_{seq}$  and the beginning of sequence  $v_{seq}$ .



— Let define  $\mathcal{O}$  the set of overlaps.

All four scenarios for sequence overlap are depicted in Figure 2.

► **Property 1.1: Overlap reverse symmetry**

One **overlap** between two sequences always implies one **overlap** between the reversed sequences:

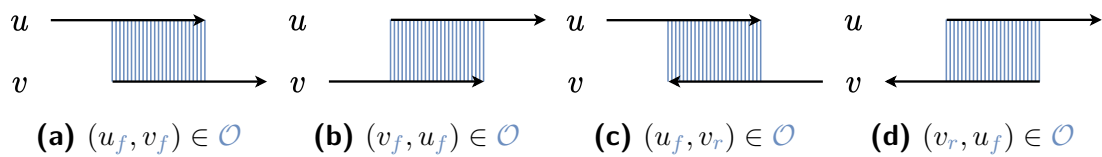
$$\forall (q, r) \in \mathcal{R}^2, (q, r) \in \mathcal{O} \iff (\bar{r}, \bar{q}) \in \mathcal{O}$$

▷ Proof

From Definition 1.3, it is sufficient to take one **alignment** between two sequences  $u$  and  $v$ , evaluated by an **alignment** score, that represents an **overlap** (hence **succession relationship**), and remark that:

- i. the **alignment** score remains the same if the **alignment** is (reversely) read from  $v$  to  $u$ ,
- ii. the score remains the same if each **nucleotide** is transformed to its complement,
- iii. combining points i. and ii. results in an **alignment** between  $\bar{v}$  and  $\bar{u}$  with the same **alignment** score as **alignment**  $u$   $v$

□



■ **Figure 2 – Overlap cases between two sequences**

$u$  and  $v$  are two sequences, represented by an arrow that shows the **forward** orientation. Each subfigure illustrates one **overlap** case: (a)  $u_f$  overlaps  $v_f$  (hence  $v_r$  overlaps  $u_r$ ) (b)  $v_f$  overlaps  $u_f$  (hence  $u_r$  overlaps  $v_r$ ) (c)  $u_f$  overlaps  $v_r$  (hence  $v_f$  overlaps  $u_r$ ) (d)  $v_r$  overlaps  $u_f$  (hence  $u_r$  overlaps  $v_f$ ).

Analogously with Definition 1.4, Definition 1.6 extends the reverse operation to **overlaps**:

► **Definition 1.6: The reverse operation for overlaps**

For a given *overlap*  $(q, r) \in \mathcal{O}$ ,  $\overline{(q, r)}$  denotes its *reverse* i.e.  $(\bar{r}, \bar{q})$ .

## 2 Storing Overlaps in a Graph Structure

*Overlaps* between the *reads* can be represented in a *graph* structure. A *graph* is a mathematical object composed of *vertices* connected by *edges*. This *graph* structure would respect the following requirements:

1. Querying requirements

- given an oriented *read*, getting every overlapping oriented *reads*
- given two oriented *reads*, answering true if and only if they overlap

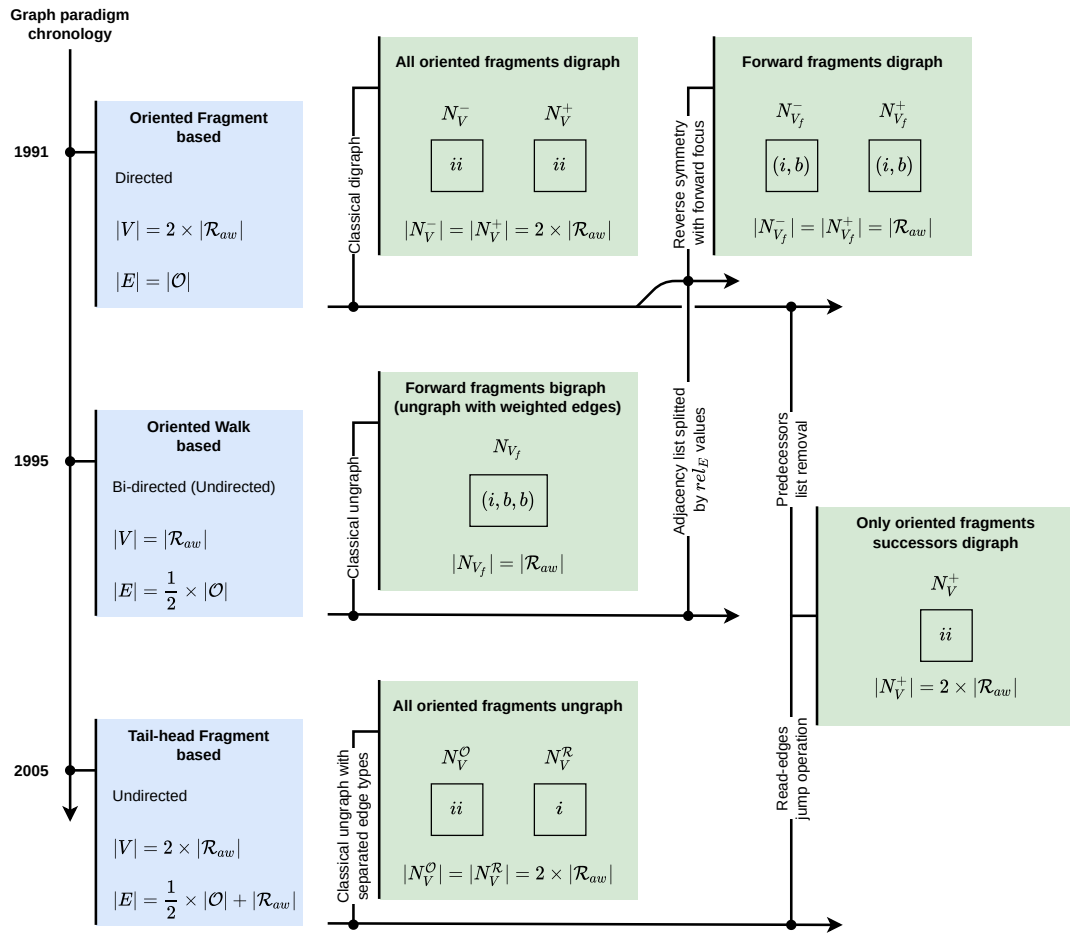
2. Dynamic requirements

- adding a *read/overlap*
- removing a *read/overlap*

Given all *reads*, the *overlaps* between them could be represented in a squared sparse matrix (of size  $\mathcal{R}^2$ ). Sparse matrix compression as *Compressed Sparse Row (CSR)* or *Compressed Sparse Column (CSC)* are known for efficiently storing the matrix and enable fast *overlap* existence querying. Finally, they enable fast *edges* iteration, but suffer from requirements 2 and can not be immediately adapted for handling *overlaps* reverse symmetry.

Thus, adjacency list structure seems to be adapted to dynamically add *vertices* and *edges*.

Sections 2.1 to 2.3 below offer a complete review of *graph* paradigms, while Sections 3.1 to 3.3 propose *graph* implementations for each *graph* paradigm in the way of its philosophy. Figure 3 summarises the different *graph* paradigms and their implementations.



LEGEND

- Graph paradigms ordered chronologically
  - Graph paradigm implementations ordered by close logic
  - $\mathcal{R}_{aw}$  Set of raw reads (fragments)
  - $\mathcal{O}$  Set of overlaps
  - $V$  Vertices set
  - $E$  Edges set
- Graph paradigm**
- Theoretical graph type
  - Theoretical number of vertices and edges

**Graph paradigm implementation**

$V$	Vertices sets of all oriented fragments	Adjacency lists
$V_f$	and only forward fragments	$i$ integer of range $0 \leq i <  \mathcal{R}_{aw} $
$N_V^-$	Lists of predecessors and successors for all the vertices	$ii$ integer of range $0 \leq ii < 2 \times  \mathcal{R}_{aw} $
$N_V^+$		$b$ boolean value $b \in \{0; 1\}$
$N_V^O$	Overlap-edges set Read-edges set	
$N_V^R$		

■ **Figure 3 – Graph paradigms and their implementations respectively ordered chronologically and by logical adaptation.**

**Blue boxes:** they correspond to graph paradigms. They are described by the type of *graph* they represent, the number of *vertices* and *edges* if they have to be drawn. **Green boxes:** they correspond to discussed *graph* implementations. After their name, the second line gives the adjacency list(s) definition set(s), following by the description of the type it (they) contain(s), finally the last line give the length of the adjacency list(s) (i.e. the number of *vertices* for which the *neighbours* are given). *Ungraph*, *digraph*, and *bigraph* respectively stand for *undirected graph*, *directed graph* and *bi-directed graph*.

### Fix Oriented FragmentS Based and Tail-head FragmentS based

## 2.1 Directed Graph: Oriented Fragments Based

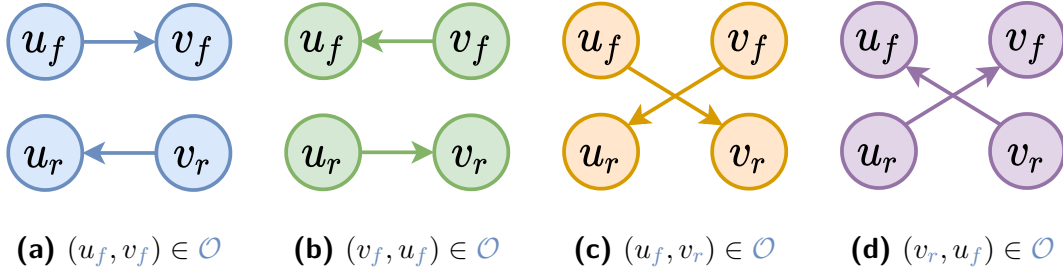
The *directed graph* structure was the first idea to represent oriented *reads* and their *overlaps*, as *overlaps* are *succession relationships*. As far as we know, this structure was first mentioned in 1991 by Kececioğlu [1]. Later, it was also described by Chin et al. [2], Kamath et al. [3], Andonov et al. [4], Shafin et al. [5] and Cheng et al. [6] in their respective *assembly* method.

### ► Definition 2.1: Oriented fragment based directed graph

Let  $G = (V, E)$  a *directed graph* such that:

- $V$  is the set of *vertices*, where each  $v \in V$  represents an oriented *read*. For each *read*, there are two *vertices* (one for the *forward* orientation, and one for the *reverse* orientation). Thus, each  $v \in V$  has the following attributes:
  - $v_{rid} \in \mathbb{N}$  the identifier of the *read* it represents
  - $v_{or} \in \{0; 1\}$  the orientation of the *read* it represents (equals to 0 if *forward*, else equals to 1 if *reverse*)
- $E$  is the set of *edges*, where each  $(u, v) \in E$  represents oriented *read*  $u$  overlapping oriented *read*  $v$ .

Figure 4 shows how each *overlap* case is represented in the *directed graph*.



■ **Figure 4 – Overlap cases in the directed graph.**

$u$  and  $v$  are two sequences represented by two vertices each  $(u_f, u_r$  and  $v_f, v_r)$ . Each subfigure illustrates one overlap impact on the directed graph: **(a)**  $(u_f, v_f) \in E$  (hence  $(v_r, u_r) \in E$ ) **(b)**  $(v_f, u_f) \in E$  (hence  $(u_r, v_r) \in E$ ) **(c)**  $(u_f, v_r) \in E$  (hence  $(v_f, u_r) \in E$ ) **(d)**  $(v_r, u_f) \in E$  (hence  $(u_r, v_f) \in E$ ).

► **Property 2.1: Sizes of the oriented fragment based directed graph**

Let  $G = (V, E)$  be an oriented fragment based directed graph:

- $|V| = 2 \times |\mathcal{R}_{aw}| = |\mathcal{R}|$
- $|E| = |\mathcal{O}|$

Because there are two vertices for each read, such that one of them represents the reverse of the other one, it is possible to define a reverse operation for this graph. Definition 2.2 extends Definitions 1.4 and 1.6:

► **Definition 2.2: The reverse operation for vertices and edges in the oriented fragment based directed graph**

**Vertices** For a given vertex  $v \in V$ ,  $\bar{v}$  denotes its reverse, where:

- $\bar{v}_{rid} = v_{rid}$
- $\bar{v}_{or} = 1 - v_{or}$

**Edges** For a given edge  $e = (u, v) \in E$ ,  $\bar{e} = (\bar{v}, \bar{u})$  denotes its reverse.

Definition 2.3 specifies what a valid walk (and a valid path) is in this graph:

► **Definition 2.3: Walking in oriented fragment based directed graph**

A valid **walk** results in a valid **path**  $p = v_0v_1 \cdots v_{n-1} \in V^n \mid n \in \mathbb{N}$  in the graph  $G = (V, E)$  that respects the following criteria:

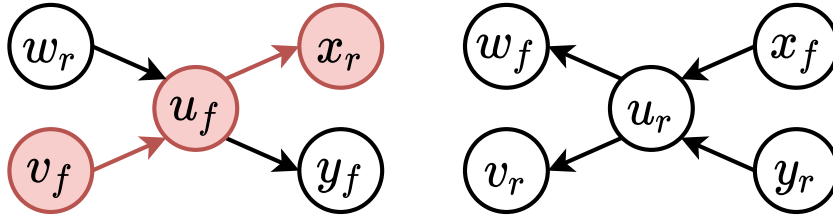
**Contiguity** Two consecutive **vertices** in the **path** must be connected by an **edge** in the **graph**:

$$\forall i \mid 0 \leq i < n - 1, (v_i, v_{i+1}) \in E$$

**Exclusive orientation** There is at most one orientation for each **vertex** in **path**  $p$ :

$$\forall v \in p, \bar{v} \notin p$$

Note that walking in this **graph** is quite similar that walking in a classic **directed graph**, except that it is not possible to walk through a **vertex** and its **reverse**. Figure 5 gives an example of a valid **walk** in the **directed graph**.



■ **Figure 5 – Valid walk example in the oriented fragment based directed graph.**

A valid **walk** that begins from **vertex**  $v_f$  is coloured in red. This **walk** results in a **path**  $p = v_f u_f x_r$ .

Finally, Property 2.2 suggests that this **directed graph** inherits a reverse symmetry property analogous to Property 1.1:

► **Property 2.2: Reverse symmetry for the vertices and the edges**

$$- \forall v \in V, \bar{v} \in V.$$

$$- \forall (u, v) \in E, \overline{(u, v)} = (\bar{v}, \bar{u}) \in E.$$

## ▷ Proof

By construction (see Definition 2.1), the first point is immediate. Concerning the second point, it is sufficient to note that if  $(r, q) \in \mathcal{O}$  (hence  $(u, v) \in E$ ), then  $(\bar{q}, \bar{r}) \in \mathcal{O}$  (hence  $(\bar{v}, \bar{u}) \in E$ ), where  $r$  and  $q$  are two oriented reads in  $\mathcal{R}$ , respectively represented by vertices  $u$  and  $v$  in  $V$ .

□

Figure 4 also illustrates Property 2.2.

## 2.2 Bi-directed Graph: Oriented Walk Based

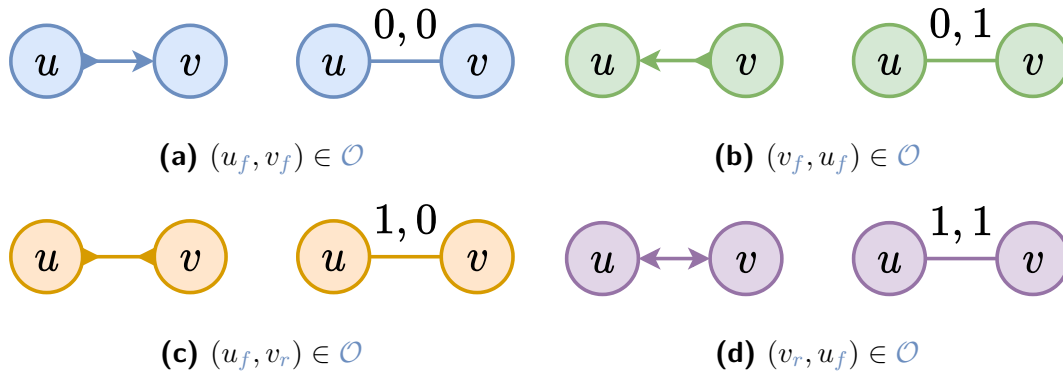
In order to avoid creating two vertices for each read (and thus creating two edges for each overlap), the bi-directed graph structure to store overlaps was introduced by E. W. Myers in 1995 [7]. The key idea is to represent each read with only one vertex and keep the strict necessary overlap information between two reads on the edge connecting two vertices. Definition 2.4 provides a formal description. It was also described by Sommer et al. [8], Hernandez et al. [9] and Salmela et al. [10] in their respective assembly method.

► **Definition 2.4: Oriented walk based bi-directed graph**

Let  $G = (V, E, or_E, rel_E)$  a bi-directed graph such that:

- $V$  is the set of vertices, where each  $v \in V$  represents a non-oriented read. For each read, there are only one vertex. Thus, each vertex  $v \in V$  has an identifier of the read it represents  $v_{rid} \in \mathbb{N}$ .
- $E$  is the set of edges, where each  $(u, v) \in E$  represents an overlap between oriented read  $u$  and oriented read  $v$ . The overlap information are given by  $or_{uv}$  and  $rel_{uv}$ :
  - the orientation of  $u$  relatively to this of  $v$  is given by edge attribute  $or_E$ : if the orientations are the same, then  $or_{uv} = 0$ , else  $or_{uv} = 1$
  - given that  $u$  identifier is lexicographically less than that of  $v$ ,  $rel_{uv} = 0$  if  $u$  in forward orientation overlaps  $v$  (in orientation given by  $or_{uv}$ ), else,  $rel_{uv} = 1$  if  $v$  (in orientation given by  $or_{uv}$ ) overlaps  $u$  in forward orientation

While visually each edge has two extremities in the bi-directed graph, in memory the graph is undirected. Figure 6 shows how visually and how internally each overlap is stored in the bi-directed graph.



■ **Figure 6 – Overlap cases in the bi-directed graph.**

$u$  and  $v$  are two unoriented sequences. Each subfigure illustrates one **overlap** impact on the **bi-directed graph**. For each, the first subfigure corresponds to the visualisation of a bi-directed **edge**, and the second shows the same information but with an undirected **edge** with  $or_{uv}$  and  $rel_{uv}$  attributes. **(a)**  $u$  and  $v$  have the same orientation in the **overlap** hence  $or_{uv} = 0$ ,  $u$  in **forward** orientation overlaps  $v$  hence  $rel_{uv} = 0$  **(b)**  $u$  and  $v$  have the same orientation in the **overlap** hence  $or_{uv} = 0$ ,  $u$  in **forward** orientation is overlapped by  $v$  hence  $rel_{uv} = 1$  **(c)**  $u$  and  $v$  do not have the same orientation in the **overlap** hence  $or_{uv} = 1$ ,  $u$  in **forward** orientation overlaps  $v$  hence  $rel_{uv} = 0$  **(d)**  $u$  and  $v$  do not have the same orientation in the **overlap** hence  $or_{uv} = 1$ ,  $u$  in **forward** orientation is overlapped by  $v$  hence  $rel_{uv} = 1$

► **Property 2.3: Sizes of the oriented walks based bi-directed graph**

Let  $G = (V, E)$  be an oriented walks based bi-directed graph:

- $|V| = |\mathcal{R}_{aw}| = \frac{1}{2} \times |\mathcal{R}|$
- $|E| = \frac{1}{2} \times |\mathcal{O}|$

As the **vertices** represent unoriented **reads** and the **overlaps** are **succession relationships** between two oriented **reads**, it is necessary to define a variable for each **vertex** that gives its orientation:

► **Definition 2.5: Vertex orientation variable**

At each **vertex**  $v \in V$  is associated a variable  $v_{or}$  that gives its orientation.

Given an **overlap** between two oriented **reads**, the variables of the two **vertices** that correspond to them (but unoriented) must respect their orientation. Definition 2.6 formalises this constraint that named co-validity constraint:



► **Definition 2.6: Co-validity for the orientation variables of the edge' extremities**

Let  $(u, v) \in E$  be an edge. The orientation variables  $u_{or}$  and  $v_{or}$  are co-valid according the edge's attributes when:

- they are co-valid for  $or_{uv}$  attribute:

$$|u_{or} - v_{or}| = or_{uv}$$

- they are co-valid for  $rel_{uv}$  attribute:

$$u_{or} = \begin{cases} rel_{uv} & \text{if } u_{rid} < v_{rid} \\ |(1 - or_{uv}) - rel_{uv}| & \text{if } u_{rid} > v_{rid} \end{cases}$$

To know if the orientation variables are co-valid, the identifiers of the two consecutive vertices have to be compared. To avoid this verification, one may remark that the attribute  $rel_{uv}$  equals to  $1 - rel_{vu}$ , and can suggest a change in the attribute value, depending on the vertex  $u$  or  $v$  from which the edge  $(u, v)$  is considered. But this would contradict the bi-directed graph paradigm philosophy (which is a undirected graph paradigm) since having two different edge attributes  $rel_{uv}$  and  $rel_{vu}$  implies that the graph is directed.

As the graph is bi-directed, it is possible to define a reverse operation. Analogously to Definition 2.2, Definition 2.7 formalises the reverse operation in the bi-directed graph.

► **Definition 2.7: The reverse operation for vertices and edges in the oriented walk based bi-directed graph**

**Vertices** For a given vertex  $v \in V$  and a fixed orientation variable  $v_{or}$ ,  $\bar{v}$  denotes its reverse, where:

- $\bar{v}_{rid} = v_{rid}$
- $\bar{v}_{or} = 1 - v_{or}$

**Edges** For a given edge  $e = (u, v) \in E$  and fix co-valid orientation variables  $u_{or}$  and  $v_{or}$ ,  $\bar{e} = (\bar{v}, \bar{u})$  denotes its reverse.

According to Definition 2.4, given  $(u, v) \in E$  and one orientation for  $u$  or for  $v$ , it is necessary to check  $rel_{uv}$  value to determine the succession relationship between oriented  $u$  and oriented  $v$ . This particularity makes the bi-directed graph oriented

*walk dependent*. Definition 2.8 specifies what a valid *walk* is in this *graph*.

► **Definition 2.8: Walking in oriented walk based bi-directed graph**

A valid *walk* results in a *path*  $p = v_0v_1 \cdots v_{n-1} \in V^n \mid n \in \mathbb{N}$  in the *graph*  $G = (V, E, or_E, rel_E)$  that respects the following criteria:

**Contiguity**

- Two consecutive *vertices* in the *path* must be connected by an *edge* in the *graph*:

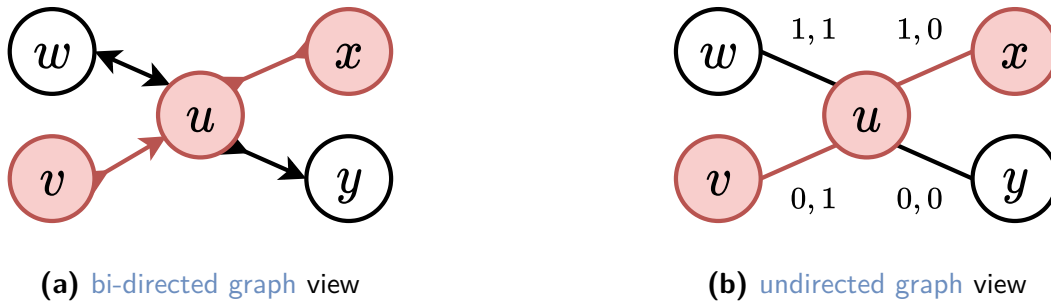
$$\forall i \mid 0 \leq i < n - 1, (v_i, v_{i+1}) \in E$$

- Given the *vertices*' orientation  $v_{i_{or}}$  and  $v_{i+1_{or}}$ , they must be co-valid.

**Exclusive orientation** At most one orientation is chosen for each *vertex*:

$$\forall v_i, v_j \in p, v_i = v_j \implies i = j$$

Figure 7 gives an example of a valid *walk* in the *bi-directed graph*. Note that walking in this *graph* is not similar to walking in a classical *undirected graph* because the orientation variables must be co-valid.



■ **Figure 7 – Valid walk example in the oriented walks based bi-directed graph.**

A valid *walk* is coloured in red. As the *graph* is undirected, the resulting *path* can be read in two possible ways: from *vertex*  $v$  or from *vertex*  $x$ . Let  $p_v = vux$  the one that begins from  $v$  such that  $v_{or} = 0, u_{or} = 0, x_{or} = 1$ . Thus,  $p_x = xuv$  is the one that begins from  $x$  such that  $x_{or} = 0, u_{or} = 1, v_{or} = 1$ . Note that  $p_x = \overline{p_v}$ . In (a) edges are drawn bidirected while in (b) they are drawn undirected.

Thanks to *edges* attributes  $or_E$  and  $rel_E$ , the *bi-directed graph* inherits a reverse symmetry property analogous to Property 1.1:

► **Property 2.4: Reverse symmetry for the edges**

Given an edge  $(u, v) \in E$  and its attributes  $or_{uv}$  and  $rel_{uv}$ , an overlap and its reverse are encoded in the graph.

▷ Proof

Without any loss of generality, let suppose that  $u$  identifier is lexicographically less than that of  $v$ .

- If  $or_{uv} = 0$ :
  - if  $rel_{uv} = 0$ , then: if  $u$  orientation is fixed forward, then overlap  $(u_f, v_f)$  is obtained; while if  $u$  orientation is fixed reverse, then  $(v_r, u_r)$  is obtained (because  $rel_{uv}$  determines if oriented  $v$  is after or before  $u$  in forward orientation)
  - else: if  $u$  orientation is fixed forward, then overlap  $(v_f, u_f)$  is obtained; while if  $u$  orientation is fixed reverse, then  $(u_r, v_r)$  is obtained.
- Else:
  - if  $rel_{uv} = 0$ , then: if  $u$  orientation is fixed forward, then overlap  $(u_f, v_r)$  is obtained; while if  $u$  orientation is fixed reverse, then  $(v_f, u_r)$  is obtained.
  - else: if  $u$  orientation is fixed forward, then overlap  $(v_r, u_f)$  is obtained; while if  $u$  orientation is fixed reverse, then  $(u_r, v_f)$  is obtained.

□

## 2.3 Undirected Graph: Tail-Head Fragments Based

A new undirected graph structure was presented by E. W. Myers in 2005 [11]: one read is represented by its tail and its head. Both the tail and the head are vertices, and there is one edge from the tail to the head. Passing through first the tail and then the head corresponds to choose the read in forward orientation, while passing through first the head and then the tail corresponds to choose it in reverse orientation. This new type of edges are called read-edges, at the opposite of overlap-edges that correspond to overlaps. A walk in the graph must alternate between read-edges and overlap-edges. This representation was also described in

the Mäkinen et al.'s book [12] and by Li [13] for its assembly method.

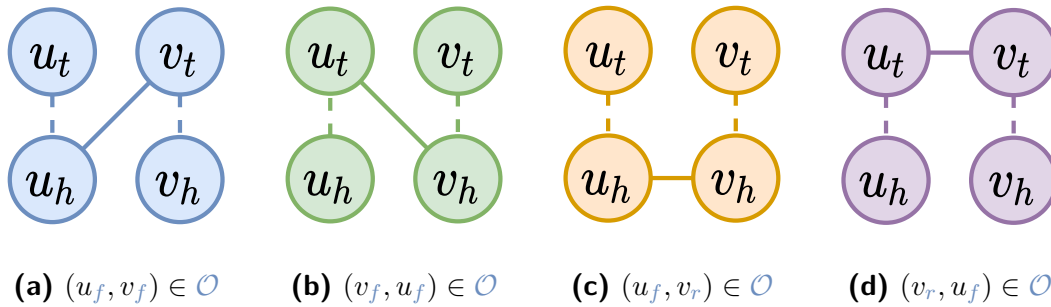
► **Definition 2.9: Tail-head Fragments Based undirected graph**

Let  $G = (V, E)$  a **undirected graph** such that:

- $V$  is the set of **vertices**, where each  $v \in V$  represents one of the extremity of one **read**. For each **read**, there are two **vertices**  $v_t, v_h$  (respectively for the tail of the **read**, and for its head). Thus, each  $v \in V$  has the following attributes:
  - $v_{rid} \in \mathbb{N}$  the identifier of the **read** it represents
  - $v_{ext} \in \{0; 1\}$  the extremity of the **read** it represents (equals to 0 if it is the head, else equals to 1 if it is the tail<sup>a</sup>)
- $E = E^{\mathcal{R}} \cup E^{\mathcal{O}}$  is the set of **edges**, such that:
  - each undirected **read-edge**  $(v_t, v_h) \in E^{\mathcal{R}}$  encodes the two possible **read** orientations (**forward** if from  $v_t$  to  $v_h$ , else **reverse**)
  - each undirected **overlap-edge**  $(u, v) \in E^{\mathcal{O}}$  represents which extremity of **read**  $u$  is overlapping which extremity of **read**  $v$

<sup>a</sup>The values for the extremities are arbitrary: while selecting 0 for the tail and 1 for the head appears more natural, switching the values will be interesting later.

Figure 8 shows for all the **overlap** cases how they are represented in the **undirected graph**.



■ **Figure 8 – Overlap cases in the undirected graph.**

$u_t, u_h$  and  $v_t, v_h$  are respectively the tail and the head of **reads**  $u$  and  $v$ . Dashed **edges** correspond to **read-edges** while plain **edges** correspond to **overlap-edges**. Each subfigure illustrates one **overlap** impact on the **undirected graph**: **(a)**  $(u_h, v_t) \in E$  **(b)**  $(v_h, u_t) \in E$  **(c)**  $(u_h, v_h) \in E$  **(d)**  $(v_t, u_t) \in E$ .

► **Property 2.5: Sizes of the tail-head fragment based undirected graph**

Let  $G = (V, E)$  be an oriented walks based bi-directed graph:

- $|V| = 2 \times |\mathcal{R}_{aw}| = |\mathcal{R}|$
- $|E| = \frac{1}{2} \times |\mathcal{O}| + |\mathcal{R}_{aw}|$

As for each read there are two vertices that represent their two extremities, it is possible to define a reverse operation. Analogously to Definitions 2.2 and 2.7, Definition 2.10 formalises the reverse operation in the undirected graph.

► **Definition 2.10: The reverse operation for vertices and edges in the tail-head fragment based undirected graph**

**Vertices** For a given vertex  $v \in V$ ,  $\bar{v}$  denotes its reverse, where:

- $\bar{v}_{rid} = v_{rid}$
- $\bar{v}_{or} = 1 - v_{or}$

**Edges** For a given edge  $e = (u, v) \in E$ ,  $\bar{e} = (v, u)$  denotes its reverse.

Because of the partition of edges set  $E$  in two subsets, walking in this graph does not correspond to a traditional undirected graph walk. Definition 2.11 specifies what a valid walk is in this graph.

► **Definition 2.11: Walking in tail-head fragments based undirected graph**

A valid walk results in a path  $p = v_0 v_1 \cdots v_{2k-1} \in V^{2k} \mid k \in \mathbb{N}$  in the graph  $G = (V, E^{\mathcal{R}} \cup E^{\mathcal{O}})$  that respects the following criteria:

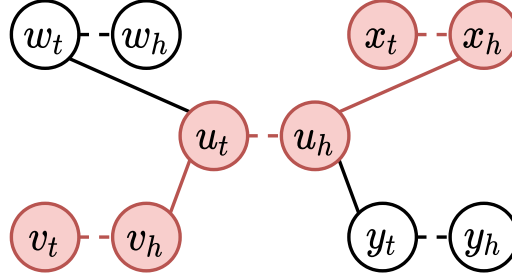
**Contiguity** Read-edges and overlap-edges alternate in the path:

- odd-numbered edges in the path are read-edges:
 
$$\forall i \mid 0 \leq i < k, (v_{2i}, v_{2i+1}) \in E^{\mathcal{R}}$$
- even-numbered edges in the path are overlap-edges:
 
$$\forall i \mid 1 \leq i < k, (v_{2i-1}, v_{2i}) \in E^{\mathcal{O}}$$

**Exclusive orientation** At most one orientation is chosen for each vertex:

$$\forall v_i, v_j \in p, v_i = v_j \implies i = j$$

Note that walking in this **graph** is not similar that walking in a classic **undirected graph**, because a valid **walk** must alternate between **read-edges** and **overlap-edges**. Figure 5 gives an example of a valid **walk** in the **undirected graph**.



■ **Figure 9 – Valid walk example in the tail-head fragment based undirected graph.**

A valid **walk** is coloured in red. As the **graph** is undirected, the resulting **path** can be read in two possible ways: from **vertex**  $v_t$  or from **vertex**  $x_t$ . Let  $p_v = v_t v_h u_t u_h x_h x_t$  the one that begins from  $v_t$ , thus  $p_x = x_t x_h u_h u_t v_h v_t$  is the one that begins from  $x_t$ . Note that  $p_x = \overline{p_v}$ .

Because the set of **vertices** contains the two extremities of each **read**, the **undirected graph** inherits a reverse symmetry property analogous to Property 1.1:

► **Property 2.6: Reverse symmetry for the read-edges and the overlap-edges**

Given one **overlap-edge**  $(u, v) \in E^{\mathcal{O}}$  and the two **read-edges**  $(u_t, u_h) \in E^{\mathcal{R}}$  and  $(v_t, v_h) \in E^{\mathcal{R}}$ , one **overlap** and its reverse are encoded in the **graph**.

- $\forall (v_t, v_h) \in E, (v_h, v_t) \in E$ , where  $v_t$  corresponds to the tail of **read**  $v$  and  $v_h$  corresponds to its head.
- $\forall (u, v) \in E, (v, u) \in E$ .

▷ **Proof**

- **Read-edge**  $(v_t, v_h)$  corresponds to reading **read**  $v$  in **forward** orientation, hence  $(v_h, v_t)$  correspond to reading it in **reverse** orientation.
- Let  $(u, v) \in E^{\mathcal{O}}$  be an **overlap-edge**, hence **read-edges**  $(\bar{u}, u)$  and  $(v, \bar{v})$  are considered. Moreover, **overlap-edge**  $(v, u)$  is also in  $E^{\mathcal{O}}$ , hence **read-edges**  $(u, \bar{u})$  and  $(\bar{v}, v)$  are considered, and they correspond to the

*reverse of the previous read-edges.*

□

Figure 8 also illustrates Property 2.6.

## 3 Overlap Graph Implementations

For each graph paradigm in Section 2 at least one implementation is proposed in Sections 3.1 to 3.3. A map of all implementations and the graph paradigms they follow is illustrated in Figure 3.

In the following, let consider that all the overlaps in overlaps set  $\mathcal{O}$  are kept, and reads in reads set  $\mathcal{R}_{aw}$  participate in at least one overlap. Also let suppose that  $\min_{r \in \mathcal{R}_{aw}} r_{rid} = 0$  and  $\max_{r \in \mathcal{R}_{aw}} r_{rid} = |\mathcal{R}_{aw}| - 1$ . The previous assertions permit to easily build an index on read identifiers  $rid$ . Also, mathematical terms as  $G = (V, E)$  remain the same as in Section 2.

### 3.1 Directed Graph: Oriented Fragments Based

Section 3.1.1 describes the first implementation idea for the oriented fragment based directed graph, while Sections 3.1.2 and 3.1.3 takes the benefits of reverse symmetry to build a more clever implementation.

#### 3.1.1 All Oriented Fragments Directed Graph (DGA)

For each read, there are two vertices in the directed graph. Remind that each read identifier  $r_{rid}$  corresponds to a unique integer identifier (it is an index, see Definition 1.1).

The All Oriented Fragments Directed Graph (DGA) implementation is the first and the simplest idea that follows a classical directed graph implementation. Thus, for each read  $r \in \mathcal{R}_{aw}$  there are two vertices  $v_f, v_r \in V$  such that the index of  $v_f$  is equal to  $2 \times r_{rid}$  and the index of  $v_r$  is equal to  $2 \times r_{rid} + 1$ . Definition 3.1 formalises indices sets.

► **Definition 3.1: Graph indices set for DGA**

Let  $Vind = \llbracket 0; 2 \times |\mathcal{R}_{aw}| \rrbracket$  be the set of indices for the vertices:

$$— |Vind| = 2 \times |\mathcal{R}_{aw}|$$

- $\forall v_{ind} \in V_{ind}, v_{ind} = 2 \times v_{rid} + v_{or}$ , where  $v_{rid}$  corresponds to the read's identifier and  $v_{or} \in \{0; 1\}$  its orientation

Let  $E_{ind} = \llbracket 0; |\mathcal{O}| \rrbracket$  be the set of indices for the edges:

- $|E_{ind}| = |\mathcal{O}|$
- $\forall e_{ind} \in E_{ind}, e_{ind}$  corresponds to the index of edge  $e$ , such that  $|e_{ind} - \bar{e}_{ind}| = 1$

It is possible now to build the two adjacency lists: the first one for the *predecessors*, the second one for the *successors*. They both contain respectively the index of each predecessor, resp. of each successor vertices as the same way as a classic directed graph implementation. They also contain the index of the edge. Definition 3.2 details the way the implementation is built.

#### add edge index

#### ► Definition 3.2: DGA implementation

DGA graph implementation is built by:

**Predecessor list**  $N_V^- \forall v \in V, \forall u \in N_v^-, u_{ind}$  is the index of a predecessor of vertex  $v$

**Successor list**  $N_V^+ \forall v \in V, \forall w \in N_v^+, w_{ind}$  is the index of a successor of vertex  $v$

Figure 10 illustrates the impact of overlap cases on the structure. For clarity's sake, only the indices of the predecessors (successors) are written.

Proposition 3.1 gives the amount of octet DGA implementation consumes.

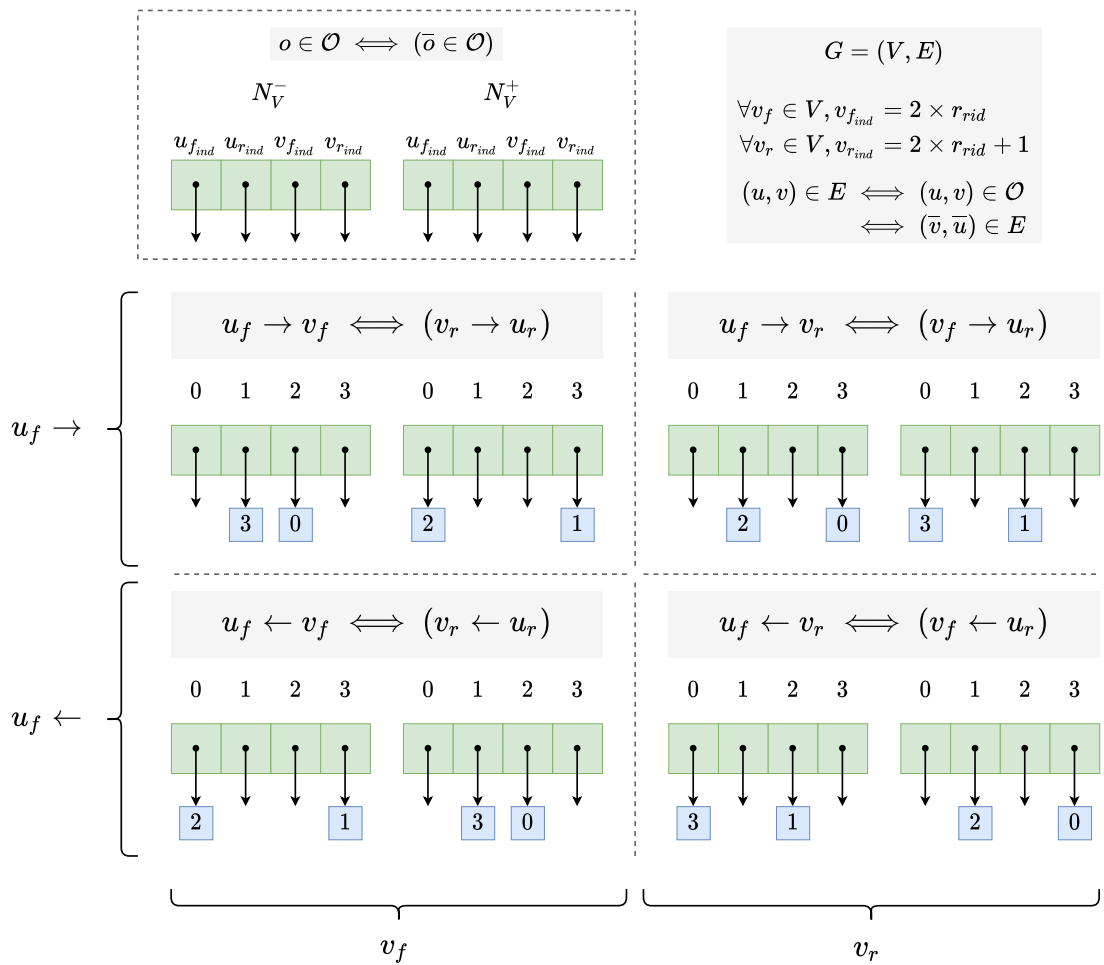
#### ► Proposition 3.1: DGA memory consumption

The memory size of the graph  $Mem(DGA)$  (in octets) is equals to:

$$Mem(DGA) = 2 \times (2 \times |\mathcal{R}_{aw}| + 1) \times P \\ + 2 \times |\mathcal{O}| \times \left( \left\lceil \frac{1 + \log_2 |\mathcal{R}_{aw}|}{8} \right\rceil + \left\lceil \frac{\log_2 |\mathcal{O}|}{8} \right\rceil \right)$$

where  $P$  is the memory size of a memory address.





■ **Figure 10 – All oriented fragments directed graph implementation.** At the top left, the dot lined box corresponds to the legend for each of the four **overlap** cases. The grey mathematical formula above provides the **overlap** and its reverse symmetric **overlap**, under parenthesis, for the illustrated case. Under, there are the two adjacency lists (the first one contains the **predecessors**, the second one contains the **successors**): they both contain the indices of the **predecessors/successors**. At the top right, the grey box gives some **graph** properties.

▷ Proof

*proof here* □

### 3.1.2 Only Oriented Fragments' Successors Directed Graph (DGS)

DGA implementation suffer from redundancies in the data because it does not take benefits from the reverse symmetry. Only Oriented Fragments' Successors Directed Graph (DGS) implementation takes benefits from the symmetry hence while the structure follows the one of DGA, only the successor list is kept. Definition 3.3 details how it is built in memory.

#### add edge index

##### ► Definition 3.3: DGS implementation

DGS graph implementation is built by the successor list  $N_v^+$  such that  $\forall v \in V, \forall w \in N_v^+, w_{ind}$  is the index of a successor of vertex  $v$ . Also, for each successor  $w$  of vertex  $v$

Figure 11 illustrates the impact of each overlap type on DGS implementation. For clarity's sake, only the indices of the successors are written, but not the edge indices.

Proposition 3.2 gives the amount of octet DGS implementation consumes.

##### ► Proposition 3.2: DGS memory consumption

The memory size of the graph  $Mem(DGS)$  (in octets) is equals to:

$$Mem(DGS) = (2 \times |\mathcal{R}_{aw}| + 1) \times P + |\mathcal{O}| \times \left( \left\lceil \frac{1 + \log_2 |\mathcal{R}_{aw}|}{8} \right\rceil + \left\lceil \frac{\log_2 |\mathcal{O}|}{8} \right\rceil \right)$$

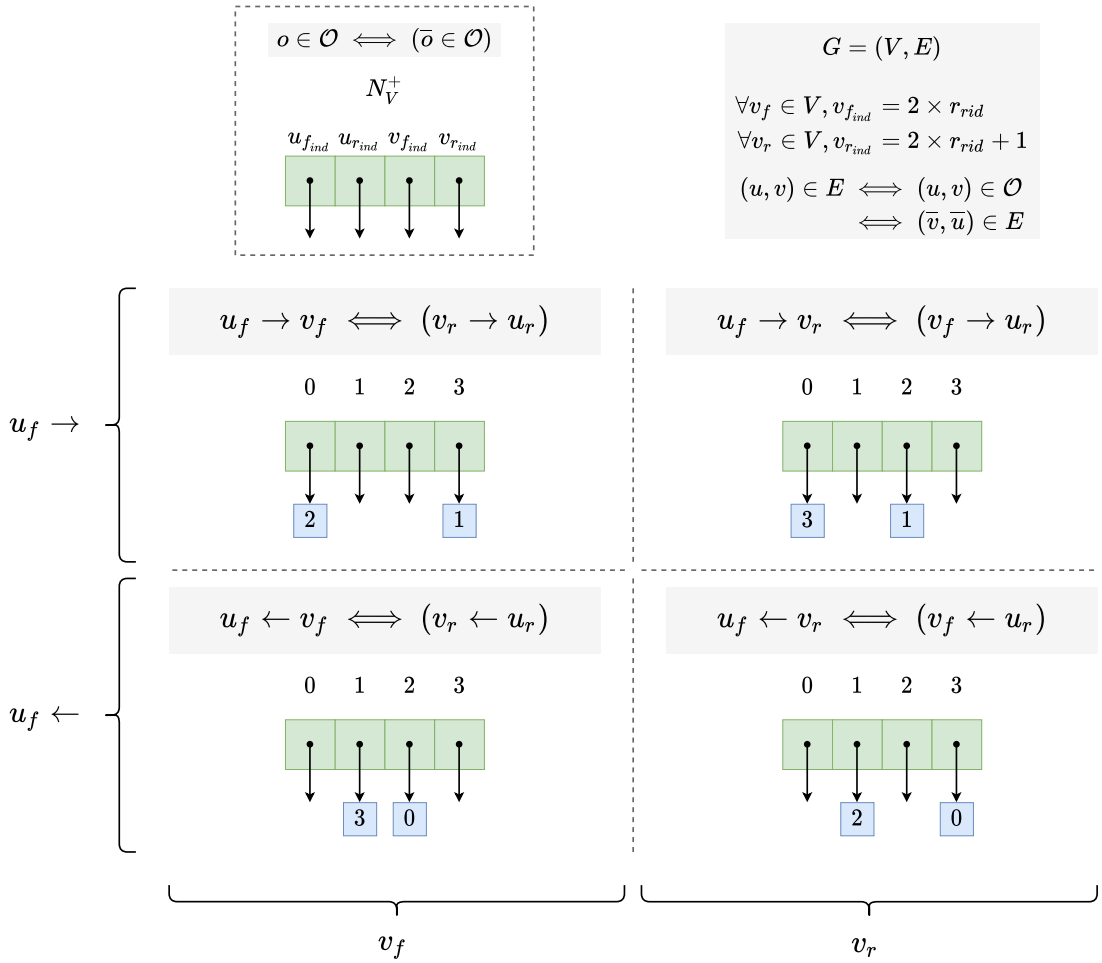
where  $P$  is the memory size of a memory address.

##### ▷ Proof

For each vertex there is a pointer to its successor list, hence  $2 \times |\mathcal{R}_{aw}|$  pointers, and there is one pointer to the vertices' successor list. Then, for each successor, its index is provided ( $\lceil \log_2 2 \times |\mathcal{R}_{aw}| \rceil$  bits), and the edge index is provided too ( $\lceil \log_2 |\mathcal{O}| \rceil$  bits). These numbers are divided by eight and rounded up to the nearest integer to get the number of octets. Then the sum is multiplied by the number of overlaps ( $|\mathcal{O}|$ ).

□

#### verify if same with commented



■ **Figure 11 – Only oriented fragments successors directed graph implementation.**

At the top left, the dot lined square corresponds to the legend for each of the four **overlap** cases. The grey mathematical formula above provides the **overlap** and its reverse symmetric **overlap**, under parenthesis, for the illustrated case. Under, there is the **successor** list: it contains the indices of the **successors**. At the top right, the grey square gives some **graph** properties.

Proposition 3.3 answer the issue of getting the **predecessors**.

► **Proposition 3.3: DGS reverse symmetry to retrieve predecessors**

For each **vertex**  $v \in V$ :

- i. its **predecessors** are the **reverse** of its **reverse'** **successors**

ii. the **edge** index of  $(u, v) \in E$  is the **reverse** of this of  $(\bar{v}, \bar{u}) \in E$

▷ **Proof**

Let  $v \in V$  be a **vertex**. Let  $u \in V$  such that  $(u, v) \in E$ . Thus:

i.  $(\bar{v}, \bar{u}) \in E$ , hence  $\bar{u}$  is the **successor** of  $\bar{v}$  that is the **reverse** of  $v$ .  
Finally,  $\bar{\bar{u}} = u$ , that is the **predecessor** of  $v$ .

ii. If  $\bar{e}_{ind}$  is the **edge** index of **edge**  $(\bar{v}, \bar{u})$ , then  $e_{ind} = \bar{e}_{ind} + c$  (where  $c = 1$  if  $\bar{e}_{ind}$  is even, else  $c = -1$ ) corresponds to the **edge** index of the **reverse** of  $(\bar{v}, \bar{u})$  hence the **edge** index of  $(u, v)$ .

□

### 3.1.3 Forward Fragments Directed Graph (DGF)

As for **DGS** implementation, the last implementation of the **directed graph** structure, **Forward Fragments Directed Graph (DGF)**, takes benefits from the **reverse** symmetry. **DGF** focuses only on the **neighbours** of the **vertices** that represent the **reads** in **forward** orientation (hence the **reads** in raw **reads** set  $\mathcal{R}_{aw}$ ). Therefore, the **vertices** indices set is not build as previously. Hence for each **read**  $r \in \mathcal{R}_{aw}$ , let  $v_{ind} = r_{rid}$  be the index of the **vertex** that represents it. Because an **overlap** involves two oriented **reads**, the **neighbours'** orientation must be included to the adjacency lists. Hence for each **neighbour**  $v \in V$  in the **predecessors/successor** lists of a **forward vertex**, let  $v_{indor} = (v_{ind}, v_{or})$  be its oriented **neighbour**. Definition 3.4 describes the different sets, and Definition 3.5 details the **DGF** implementation. For clarity's sake, only the indices of the **predecessors/successors** are written, but not the **edge** indices.

► **Definition 3.4: Graph indices set for DGA**

Let  $V_{ind} = \llbracket 0; |\mathcal{R}_{aw}| \rrbracket$  be the set of indices for the **vertices**:

- $|V_{ind}| = |\mathcal{R}_{aw}|$
- $\forall v_{ind} \in V_{ind}, v_{ind} = v_{rid}$ , where  $v_{rid}$  corresponds to the **read's** identifier

Let  $V_{indor} = \llbracket 0; |\mathcal{R}_{aw}| \rrbracket \times \{0; 1\}$  be the set of indices enriched by a boolean value for the oriented **vertices**:

- $|V_{indor}| = 2 \times |\mathcal{R}_{aw}|$

- $\forall v_{indor} \in V_{indor}, v_{indor} = (v_{rid}, v_{or})$ , where  $v_{rid}$  corresponds to the read's identifier and  $v_{or}$  to its orientation

Let  $E_{ind} = \llbracket 0; |\mathcal{O}| \rrbracket$  be the set of indices for the edges:

- $|E_{ind}| = |\mathcal{O}|$
- $\forall e_{ind} \in E_{ind}, e_{ind}$  corresponds to the index of edge  $e$ , such that  $|e_{ind} - \bar{e}_{ind}| = 1$

### add edge index

#### ► Definition 3.5: DGF implementation

DGF graph implementation is built by:

**Predecessor list for the forward**  $N_{V_f}^- \forall v \in V_f, \forall u \in N_v^-, u_{ind}$  is the index of a predecessor of vertex  $v$ , and  $u_{or}$  gives its orientation

**Successor list for the forward**  $N_{V_f}^+ \forall v \in V_f, \forall w \in N_v^+, w_{ind}$  is the index of a successor of vertex  $v$ , and  $w_{or}$  gives its orientation

#### ► Proposition 3.4: DGF memory consumption

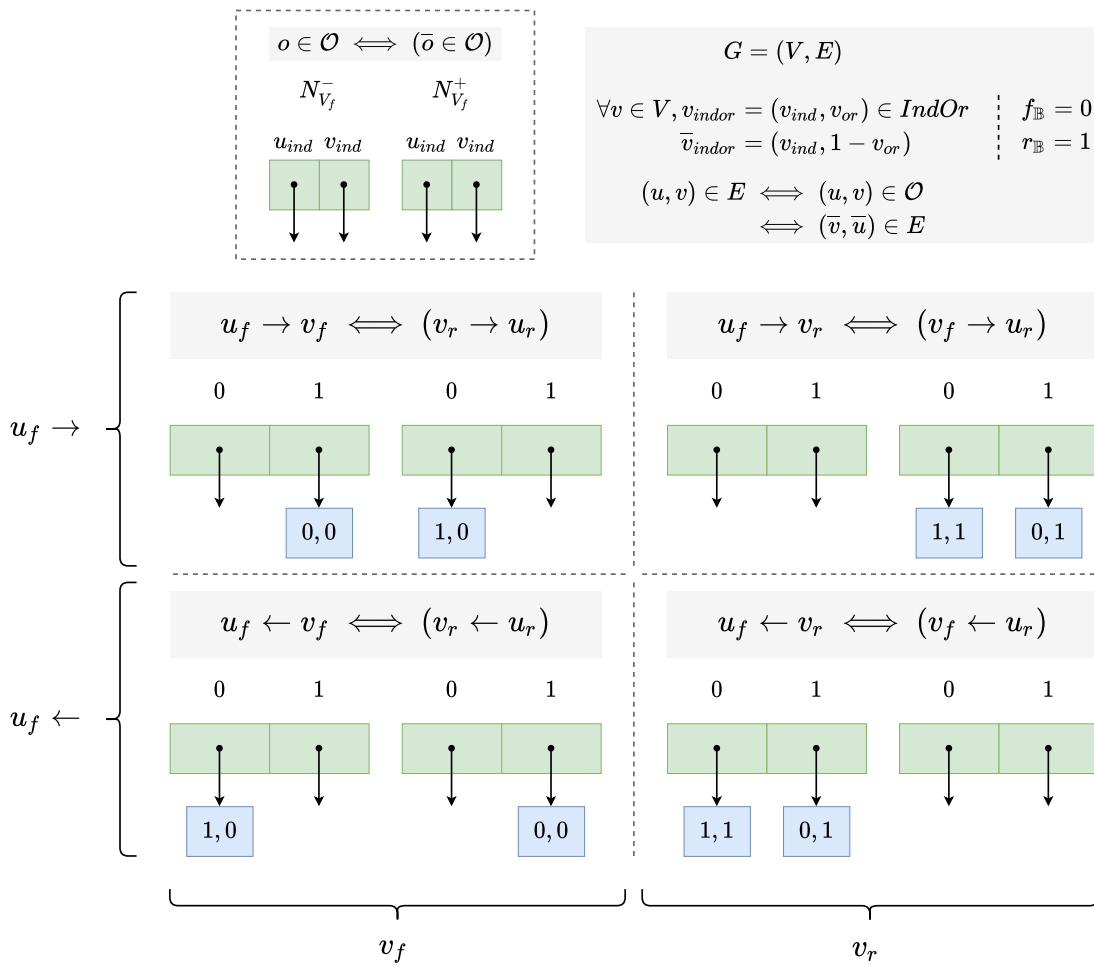
The memory size of the graph  $Mem(DGF)$  (in octets) is equals to:

$$Mem(DGF) = 2 \times (|\mathcal{R}_{aw}| + 1) \times P \\ + |\mathcal{O}| \times \left( \left\lceil \frac{1 + \log_2 |\mathcal{R}_{aw}|}{8} \right\rceil + \left\lceil \frac{\log_2 |\mathcal{O}|}{8} \right\rceil \right)$$

where  $P$  is the memory size of a memory address.

#### ▷ Proof

For each (*forward*) vertex there is a pointer to its predecessor and successor lists, hence  $2 \times |\mathcal{R}_{aw}|$  pointers, and there is one pointer to the vertices' predecessor and successor lists (hence  $+2$ ). Then, for each predecessor/successor, its index and a boolean orientation value are provided ( $\lceil \log_2 2 \times |\mathcal{R}_{aw}| \rceil$  bits), and the edge index is provided too ( $\lceil \log_2 |\mathcal{O}| \rceil$  bits). These numbers are divided by eight and rounded up to the nearest integer to get the number of



■ **Figure 12 – Forward fragments directed graph implementation.**

At the top left, the dot lined square corresponds to the legend for each of the four **overlap** cases. The grey mathematical formula above provides the **overlap** and its reverse symmetric **overlap**, under parenthesis, for the illustrated case. Under, there are the two adjacency lists for the **forward** (the first one contains the **predecessors**, the second one contains the **successors**): they both contain the couples index-orientation of the **predecessors/successors**. At the top right, the grey square gives some **graph** properties.

*octets. Then the sum is multiplied by the number of overlaps ( $|\mathcal{O}|$ ).*

□

**verify if same with commented**

Note that in contrast to Proposition 3.2, the memory consumption of adjacency lists pointers is equals to  $2 \times (|\mathcal{R}_{aw}| + 1)$  (versus  $(2 \times |\mathcal{R}_{aw}| + 1)$ ). In fact, the extra  $P$  is due to an extra adjacency list.

Proposition 3.5 answer the issue of getting the predecessors/successors of the reverse.

► **Proposition 3.5: DGF reverse symmetry to retrieve predecessors**

For each vertex  $v \in V_r$ :

- i. its predecessors are the reverse of its reverse' successors
- ii. its successors are the reverse of its reverse' predecessors
- iii. the edge index of  $(u, v) \in E$  is the reverse of this of  $(\bar{v}, \bar{u}) \in E$

▷ Proof

Let  $v \in V_r$  be a vertex that represents a reverse read.

— Let  $u \in V$  such that  $(u, v) \in E$ . Thus:

i.  $(\bar{v}, \bar{u}) \in E$ , hence  $\bar{u}$  is the successor of  $\bar{v}$  that is the reverse of  $v$ .

Finally,  $\bar{\bar{u}} = u$ , that is the predecessor of  $v$ .

ii. If  $\bar{e}_{ind}$  is the edge index of edge  $(\bar{v}, \bar{u})$ , then  $e_{ind} = \bar{e}_{ind} + c$  (where  $c = 1$  if  $\bar{e}_{ind}$  is even, else  $c = -1$ ) corresponds to the edge index of the reverse of  $(\bar{v}, \bar{u})$  hence the edge index of  $(u, v)$ .

— Let  $w \in V$  such that  $(v, w) \in E$ . Thus:

i.  $(\bar{w}, \bar{v}) \in E$ , hence  $\bar{w}$  is the predecessor of  $\bar{v}$  that is the reverse of  $v$ . Finally,  $\bar{\bar{w}} = w$ , that is the successor of  $v$ .

ii. If  $\bar{e}_{ind}$  is the edge index of edge  $(\bar{w}, \bar{v})$ , then  $e_{ind} = \bar{e}_{ind} + c$  (where  $c = 1$  if  $\bar{e}_{ind}$  is even, else  $c = -1$ ) corresponds to the edge index of the reverse of  $(\bar{w}, \bar{v})$  hence the edge index of  $(v, w)$ .

□

## 3.2 Bi-directed Graph: Oriented Walk Based

### 3.2.1 Forward Fragments Bi-directed Graph (BG)

In the **bi-directed graph**, there is one **vertex** for each raw **read** in  $\mathcal{R}_{aw}$ . Hence for each **vertex**  $v$  is associated an index  $v_{ind} = v_{rid}$ . An **overlap** and its **reverse** are represented by only one **edge** that carries strictly necessary **overlap** attributes, as described in Definition 2.4. Remind that the **bi-directed graph** is equivalent to a **undirected graph**: that's implemented by **Unoriented Fragments Bi-directed Graph (BG)** implementation. Definition 3.4 describes the different sets, and Definition 3.5 details the **DGF** implementation. For clarity's sake, only the indices of the **predecessors/successors** are written, but not the **edge** indices.

► **Definition 3.6: Graph indices set for BG**

Let  $V_{ind} = \llbracket 0; |\mathcal{R}_{aw}| \rrbracket$  be the set of indices for the **vertices**:

- $|V_{ind}| = |\mathcal{R}_{aw}|$
- $\forall v_{ind} \in V_{ind}, v_{ind} = v_{rid}$ , where  $v_{rid}$  corresponds to the **read's** identifier

Let  $E_{ind} = \llbracket 0; \frac{1}{2} \times |\mathcal{O}| \rrbracket$  be the set of indices for the **edges**:

- $|E_{ind}| = \frac{1}{2} \times |\mathcal{O}|$
- $\forall e_{ind} \in E_{ind}, e_{ind}$  corresponds to the index of **edge**  $e$

#### put paragraph in definition

Concerning the edge indices  $e_{ind}$ , it is possible to use their euclidean quotient by two  $e'_{ind}$  in memory and retrieving them thanks to the orientation of the fewer vertex index (let say  $u$ ). In fact, if the considered overlap uses  $u_f$ , then  $e_{ind} = 2 \times e'_{ind}$ , else  $e_{ind} = 2 \times e'_{ind} + 1$

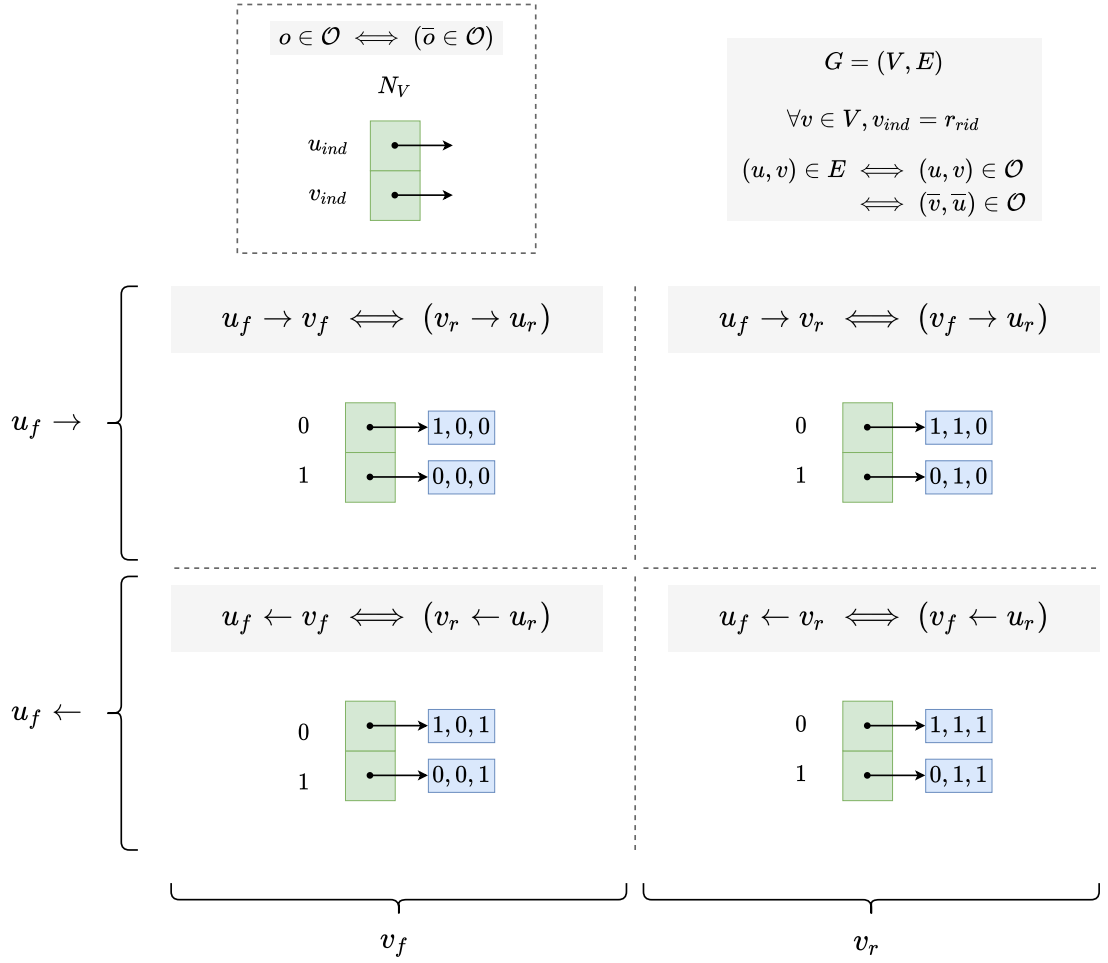
► **Definition 3.7: BG implementation**

**BG graph** implementation is built by:

**Neighbours list for the forward**  $N_V \forall v \in V, \forall w \in N_v, w_{ind}$  is the index of a **neighbour** of **vertex**  $v$

**Edges attributes** for each **neighbour**  $w$  of a **vertex**  $v$  two boolean values are provided:  $or_{vw}$  and  $rel_{vw}$  (see Definition 2.4)





■ **Figure 13** – caption title All four **overlap** cases structured in the **bi-directed graph**.

At the top, the dot lined square corresponds to the legend for the four **overlap** cases. The mathematical formula above the line is the **overlap** such that  $u$  is in **forward** orientation, and the reverse symmetric **overlap** is under parenthesis. Under the line, there is the one list that contains, in order, the index of the **neighbour**, the value of  $or_{uv}$  and the value of  $rel_{uv}$ .

► **Proposition 3.6: BG memory consumption**

The memory size of the graph  $Mem(BG)$  (in octets) equals:

$$\begin{aligned}
 Mem(BG) &= (|\mathcal{R}_{aw}| + 2) \times P \\
 &\quad + |\mathcal{O}| \times \left( \left\lceil \frac{\log_2 |\mathcal{R}_{aw}|}{8} \right\rceil + \left\lceil \frac{\log_2 |\mathcal{O}| - 1}{8} \right\rceil \right) \\
 &\quad + \left\lceil \frac{|\mathcal{O}|}{8} \right\rceil
 \end{aligned}$$

where  $P$  is the memory size of a memory address.

▷ Proof

For each (unoriented) *vertex* there is a pointer to its *neighbour list*, hence  $|\mathcal{R}_{aw}|$  pointers, and there is one pointer to the *vertices' neighbour list* and one to the *edges attribute list* (hence +2). Then, for each *neighbour*, its index is provided ( $\lceil \log_2 |\mathcal{R}_{aw}| \rceil$  bits), and the *edge index* is provided too ( $\lceil \log_2 (\frac{1}{2} \times |\mathcal{O}|) \rceil$  bits). These numbers are divided by eight and rounded up to the nearest integer to get the number of octets. Then the sum is multiplied by the number of *overlaps* ( $|\mathcal{O}|$ ). Finally, for each *edge index* ( $\frac{1}{2} \times |\mathcal{O}|$ ), there are two boolean *edge attributes*.  $\frac{1}{2} \times |\mathcal{O}| \times 2$  is divided by eight and rounded up to get the number of octets. □

**verify if same as comments**

The four combinations of edges' attributes *or* and *rel* let representing all the overlap cases and their reverse without any redundancy. To verify the reverse symmetry, it is sufficient (and easier) to choose an orientation of  $u$  and calculate the overlap case with the edges' attributes.

### 3.2.2 Forward Fragments Directed Graph (DGF)

**description**

**ref figure from digraph implementation revsymg**

**memory**

## 3.3 Undirected Graph: Tail-Head Fragments Based

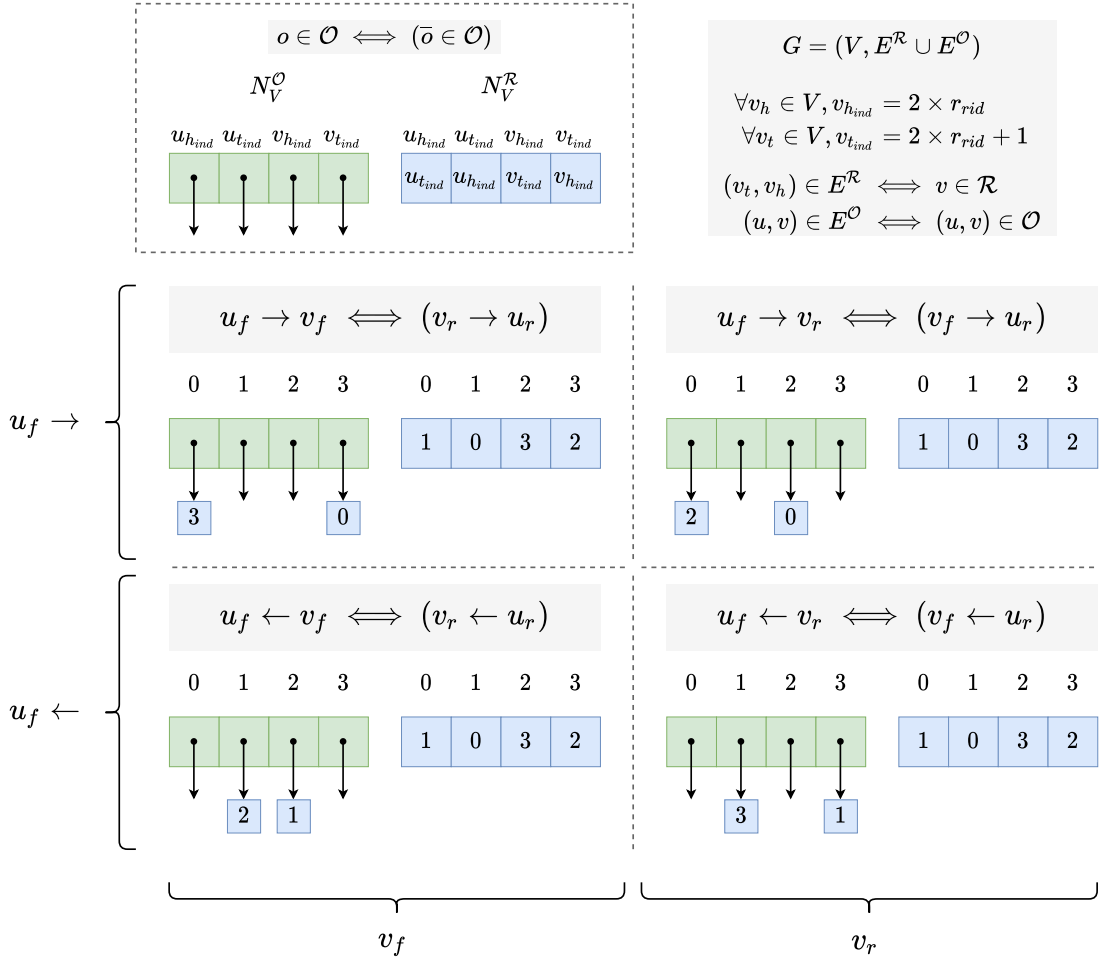
### 3.3.1 All Oriented Fragments Undirected Graph (UGA)

**fix for ungraph**

For each *read*, there are two *vertices* in the *directed graph*. Remind that each *read* identifier  $r_{rid}$  correspond to a unique integer identifier (it is an index,

see Definition 1.1). Thus, for each read  $r \in \mathcal{R}_{aw}$  there are two vertices  $v_f, v_r \in V$  such that the index of  $v_f$  is equal to  $2 \times r_{rid}$  and the index of  $v_r$  is equal to  $2 \times r_{rid} + 1$ .

It is possible now to build the adjacency lists (one list for the predecessors, and another one for the successors).



■ **Figure 14 – caption title** All four overlap cases structured in the directed graph.

At the top left, the dot lined square corresponds to the legend for the four overlap cases. The grey mathematical formula above is the overlap such that  $u$  is in forward orientation, and the reverse symmetric overlap is under parenthesis. Under, there are the two adjacency lists (the first one contains the neighbours for overlap-edges, the second one contains the neighbours for read-edge): they both contain the indices of the neighbours. Top right grey square gives some graph properties.

memory

## 3.3.2 Read-edges Jump Directed Graph (DGS)

description

Cref figure only oriented frag succs digraph

memory

remarque que l'on peut arriver visuellement au digraph

## 4 Memory and Time Costs

For a vertex/edge index that represents the forward/reverse read/edge according to its parity, the reverse algorithm cost two elementary operations. For clarity's sake,  $\text{REV}(ind) = \overline{ind}$ .

Details of complexity calculus

rev function complexities

## ► Algorithm 1: Reverse operation on index

**Require:** Vertex/edge index  $ind$ .**Ensure:** Returns index of the reverse vertex/edge.

```

1: function REV( $ind$ )
2:   if  $ind \mid 2$  then
3:     return  $ind + 1$ 
4:   return  $ind - 1$ 

```

## 4.1 Algorithms

### 4.1.1 Iterating Over the Predecessors

---

#### ► Algorithm 2: Iterate over the predecessors for DGS

---

**Require:** Graph  $G = (V, E)$ , vertex  $v \in V$ , empty list  $preds$ .

**Ensure:** The returned list contains the predecessors of  $v$  in graph  $G$  and the corresponding edge index.

```

1: function DGS_PREDS( $G, v, preds$ )
2:   for  $(u, e_{ind}) \in N_v^+$  do
3:      $preds.APPEND(\overline{u}, \overline{e_{ind}})$ 
4:   return  $preds$ 

```

---

#### ► Algorithm 3: Iterate over the predecessors for DGF

---

**Require:** Graph  $G = (V, E)$ , vertex  $v \in V$ , orientation of  $v$   $v_{or} \in \{0; 1\}$ , empty list  $preds$ .

**Ensure:** The returned list contains the predecessors of  $v$  in graph  $G$  and the corresponding edge index.

```

1: function DGF_PREDS( $G, v, v_{or}, preds$ )
2:   if  $v_{or} = 0$  then ▷  $v$  forward
3:     for  $(u, e_{ind}) \in N_v^-$  do
4:        $preds.APPEND(u, e_{ind})$ 
5:   else
6:     for  $(u, u_{or}, e_{ind}) \in N_v^+$  do
7:        $preds.APPEND(u, 1 - u_{or}, \overline{e_{ind}})$ 
8:   return  $preds$ 

```

---

#### ► Algorithm 4: Iterate over the predecessors for BGU

---

**Require:** Graph  $G = (V, E)$ , vertex  $v \in V$ , orientation of  $v$   $v_{or} \in \{0; 1\}$ , empty list  $preds$ .

**Ensure:** The returned list contains the predecessors of  $v$  in graph  $G$  and the corresponding edge index.

```

1: function BGU_PREDS( $G, v, v_{or}, preds$ )
2:    $orientations \leftarrow (0, 1)$  ▷ forward/reverse orientations
3:   for  $(u, e'_{ind}) \in N_v$  do
4:     if  $v < u$  then

```

---

---

```

5:         if  $v_{or} = 0$  then
6:             if  $rel_{uv} = 1$  then  $\triangleright u_f \rightarrow v_f$  or  $u_r \rightarrow v_f$ 
7:                  $preds.APPEND(u, orientations[or_{uv}], 2 \times e'_{ind})$ 
8:             else if  $rel_{uv} = 0$  then  $\triangleright u_f \rightarrow v_r$  or  $u_r \rightarrow v_r$ 
9:                  $preds.APPEND(u, orientations[1 - or_{uv}], 2 \times e'_{ind} + 1)$ 
10:         else if  $v_{or} = 0$  then
11:             if  $or_{uv} = 0$  then
12:                 if  $rel_{uv} = 0$  then  $\triangleright u_f \rightarrow v_f$ 
13:                      $preds.APPEND(u, 0, 2 \times e'_{ind})$ 
14:                 else if  $rel_{uv} = 1$  then  $\triangleright u_r \rightarrow v_f$ 
15:                      $preds.APPEND(u, 1, 2 \times e'_{ind} + 1)$ 
16:             else if  $or_{uv} = 0$  then
17:                 if  $rel_{uv} = 1$  then  $\triangleright u_r \rightarrow v_r$ 
18:                      $preds.APPEND(u, 1, 2 \times e'_{ind} + 1)$ 
19:             else if  $rel_{uv} = 0$  then  $\triangleright u_f \rightarrow v_r$ 
20:                  $preds.APPEND(u, 0, 2 \times e'_{ind})$ 
21:         return  $preds$ 

```

---

#### 4.1.2 Iterating Over the Successors

---

##### ► Algorithm 5: Iterate over the successors for DGS

---

**Require:** Graph  $G = (V, E)$ , vertex  $v \in V$ , empty list  $succs$ .

**Ensure:** The returned list contains the successors of  $v$  in graph  $G$ .

```

1: function DGS_SUCCS( $G, v, succs$ )
2:     for  $(w, e_{ind}) \in N_v^+$  do
3:          $succs.APPEND(w, e_{ind})$ 
4:     return  $succs$ 

```

---

##### ► Algorithm 6: Iterate over the successors for DGF

---

**Require:** Graph  $G = (V, E)$ , vertex  $v \in V$ , orientation of  $v$   $v_{or} \in \{0; 1\}$ , empty list  $succs$ .

**Ensure:** The returned list contains the successors of  $v$  in graph  $G$ .

```

1: function DGF_SUCCS( $G, v, v_{or}, succs$ )
2:     if  $v_{or} = 0$  then  $\triangleright v$  forward
3:         for  $(w, e_{ind}) \in N_v^+$  do
4:              $succs.APPEND(w, e_{ind})$ 

```

---

---

```

5:   else
6:     for  $(w, w_{or}, e_{ind}) \in N_v^-$  do
7:        $succs.APPEND(w, 1 - w_{or}, \overline{e_{ind}})$ 
8:   return succs

```

---



---

► **Algorithm 7: Iterate over the successors for BGU**

---

**Require:** Graph  $G = (V, E)$ , vertex  $v \in V$ , orientation of  $v$   $v_{or} \in \{0; 1\}$ , empty list  $succs$ .

**Ensure:** The returned list contains the successors of  $v$  in graph  $G$ .

```

1: function BGU_SUCCS( $G, v, v_{or}, succs$ )
2:    $orientations \leftarrow (0, 1)$  ▷ forward/reverse orientations
3:   for  $(w, e'_{ind}) \in N_v$  do
4:     if  $v < w$  then
5:       if  $v_{or} = 0$  then
6:         if  $rel_{vw} = 0$  then ▷  $v_f \rightarrow w_f$  or  $v_f \rightarrow w_r$ 
7:            $succs.APPEND(w, orientations[or_{vw}], 2 \times e'_{ind})$ 
8:         else if  $rel_{uw} = 1$  then ▷  $v_r \rightarrow w_f$  or  $v_r \rightarrow w_r$ 
9:            $succs.APPEND(u, orientations[1 - or_{uw}], 2 \times e'_{ind} + 1)$ 
10:        else if  $v_{or} = 0$  then
11:          if  $or_{uw} = 0$  then
12:            if  $rel_{uw} = 1$  then ▷  $v_f \rightarrow w_f$ 
13:               $succs.APPEND(w, 0, 2 \times e'_{ind})$ 
14:            else if  $rel_{uw} = 0$  then ▷  $v_f \rightarrow w_r$ 
15:               $succs.APPEND(w, 1, 2 \times e'_{ind} + 1)$ 
16:          else if  $or_{uw} = 0$  then
17:            if  $rel_{uw} = 0$  then ▷  $v_r \rightarrow w_r$ 
18:               $succs.APPEND(w, 1, 2 \times e'_{ind} + 1)$ 
19:            else if  $rel_{uw} = 1$  then ▷  $v_r \rightarrow w_f$ 
20:               $succs.APPEND(w, 0, 2 \times e'_{ind})$ 
21:   return succs

```

---

## 4.1.3 Adding a Vertex

---

**► Algorithm 8: Add a new vertex in the graph for DGS**


---

**Require:** Graph  $G = (V, E)$ .

**Ensure:** Returns the new vertex' index (in forward orientation if there is the choice).

```

1: function DGS_ADD_VERTEX( $G$ )
2:    $N_V^+$ .APPEND(EMPTY_LIST( ))
3:    $N_V^+$ .APPEND(EMPTY_LIST( ))
4:   return  $|N_V^+| - 2$ 

```

---



---

**► Algorithm 9: Add a new vertex in the graph for DGF**


---

**Require:** Graph  $G = (V, E)$ .

**Ensure:** Returns the new vertex' index (in forward orientation if there is the choice).

```

1: function DGF_ADD_VERTEX( $G$ )
2:    $N_V^-$ .APPEND(EMPTY_LIST( ))
3:    $N_V^+$ .APPEND(EMPTY_LIST( ))
4:   return  $|N_V^+| - 1$ 

```

---



---

**► Algorithm 10: Add a new vertex in the graph for BGU**


---

**Require:** Graph  $G = (V, E)$ .

**Ensure:** Returns the new vertex' index (in forward orientation if there is the choice).

```

1: function BGU_ADD_VERTEX( $G$ )
2:    $N_V$ .APPEND(EMPTY_LIST( ))
3:   return  $|N_V| - 1$ 

```

---



## 4.1.4 Adding an Edge

---

**► Algorithm 11: Add a new edge and its reverse in the graph for DGS**


---

**Require:** Graph  $G = (V, E)$ ,  $(u, v) \in V^2$ . Note that the vertices are already in the graph.

**Ensure:** Returns the new edge's index (in forward orientation if there is the choice).

```

1: function DGS_ADD_EDGE( $G, u, v$ )
2:   ▷  $ind\_edges$  is the number of edges. It is always even.           ◁
3:    $N_u^+$ .APPEND( $v, ind\_edges$ )
4:    $N_v^+$ .APPEND( $\bar{u}, ind\_edges + 1$ )
5:    $ind\_edges \leftarrow ind\_edges + 2$ 
6:    $card\_edges \leftarrow card\_edges + 2$ 
7:   return  $ind\_edges - 2$ 

```

---



---

**► Algorithm 12: Add a new edge and its reverse in the graph for DGF**


---

**Require:** Graph  $G = (V, E)$ ,  $(u, v) \in V^2$ , with their orientation  $(u_{or}, v_{or}) \in \{0; 1\}^2$ . Note that the vertices are already in the graph.

**Ensure:** Returns the new edge's index (in forward orientation if there is the choice).

```

1: function DGF_ADD_EDGE( $G, u, u_{or}, v, v_{or}$ )
2:   ▷  $ind\_edges$  is the number of edges. It is always even.           ◁
3:   if  $u_{or} = 0$  then                                               ▷  $u_f \rightarrow v_f$  or  $u_f \rightarrow v_r$ 
4:      $N_u^+$ .APPEND( $v, v_{or}, ind\_edges$ )
5:   else                                                             ▷  $u_f \leftarrow v_f$  or  $u_f \leftarrow v_r$ 
6:      $N_u^-$ .APPEND( $v, 1 - v_{or}, ind\_edges + 1$ )
7:   if  $v_{or} = 0$  then                                               ▷  $u_f \rightarrow v_f$  or  $u_r \rightarrow v_f$ 
8:      $N_v^-$ .APPEND( $u, u_{or}, ind\_edges$ )
9:   else                                                             ▷  $u_f \leftarrow v_f$  or  $u_r \leftarrow v_f$ 
10:     $N_v^+$ .APPEND( $u, 1 - u_{or}, ind\_edges + 1$ )
11:     $ind\_edges \leftarrow ind\_edges + 2$ 
12:     $card\_edges \leftarrow card\_edges + 2$ 
13:    return  $ind\_edges - 2$ 

```

---

---

**► Algorithm 13: Add a new edge and its reverse in the graph for BGU**


---

**Require:** Graph  $G = (V, E)$ ,  $(u, v) \in V^2$ , with their orientation  $(u_{or}, v_{or}) \in \{0; 1\}^2$ . Note that the vertices are already in the graph.

**Ensure:** Returns the new edge's index (in forward orientation if there is the choice).

```

1: function BGU_ADD_EDGE( $G, u, u_{or}, v, v_{or}$ )
2:   ▷  $ind\_edges$  is the number of edges.  $E_{attr}$  is a list of edges attributes. ◁
3:    $or \leftarrow |u_{or} - v_{or}|$ 
4:   if  $u < v$  then
5:      $rel \leftarrow u_{or}$ 
6:   else
7:      $rel \leftarrow 1 - v_{or}$ 
8:    $N_u$ .APPEND( $v, ind\_edges$ )
9:    $N_v$ .APPEND( $u, ind\_edges$ )
10:   $E_{attr}$ .APPEND( $or, rel$ )
11:   $ind\_edges \leftarrow ind\_edges + 1$ 
12:   $card\_edges \leftarrow card\_edges + 1$ 
13:  return  $2 \times (ind\_edges - 1)$ 

```

---

## 4.1.5 Deleting a Vertex

---

**► Algorithm 14: Delete a vertex for DGS**


---

**Require:** Graph  $G = (V, E)$ , vertex  $v \in V$  (in forward orientation).

**Ensure:** Delete the vertex and its reverse such that  $V' = V \setminus \{v, \bar{v}\}$ , and  $\forall v \in V', 0 \leq v_{ind} \leq |V'| - 2$ .

```

1: function DGS_DELETE_VERTEX( $G, v$ )
2:   ▷ Delete it from its predecessors ◁
3:    $v\_rev \leftarrow v + 1$ 
4:   for  $(u, e_{ind}) \in N_{v\_rev}^+$  do
5:      $u\_rev \leftarrow \bar{u}$ 
6:      $adj\_ind \leftarrow GET\_INDEX(N_{u\_rev}^+, e_{ind}, 1)$ 
7:     if  $adj\_ind = |N_{u\_rev}^+| - 1$  then
8:       ▷ Just delete the last element ◁
9:        $N_{u\_rev}^+.$ POP( )
10:    else
11:      ▷ Replace the edge by the last in the neighbour list ◁

```

---

---

```

12:      $N_{u\_rev}^+[adj\_ind] \leftarrow N_{u\_rev}^+.POP( )$ 
13:      $card\_edges \leftarrow card\_edges - 2$ 
14:     DELETE( $N_{v\_rev}^+$ )
15:     ▷ Delete it from its successors ◁
16:     for  $(w, e_{ind}) \in N_v^+$  do
17:          $w\_rev \leftarrow \bar{w}$ 
18:          $adj\_ind \leftarrow GET\_INDEX(N_{w\_rev}^+, \bar{e}_{ind}, 1)$ 
19:         if  $adj\_ind = |N_{w\_rev}^+| - 1$  then
20:             ▷ Just delete the last element ◁
21:              $N_{w\_rev}^+.POP( )$ 
22:         else
23:             ▷ Replace the edge by the last in the neighbour list ◁
24:              $N_{w\_rev}^+[adj\_ind] \leftarrow N_{w\_rev}^+.POP( )$ 
25:              $card\_edges \leftarrow card\_edges - 2$ 
26:     DELETE( $N_v^+$ )
27:     ▷ Delete the whole vertex ◁
28:     if  $v = |N_V^+| - 2$  then
29:         ▷ It is the last, just pop it and its reverse ◁
30:          $N_V^+.POP( )$ 
31:          $N_V^+.POP( )$ 
32:     else
33:         ▷ Replace it and its reverse by the last and its reverse ◁
34:          $N_{v\_rev}^+ \leftarrow N_V^+.POP( )$ 
35:          $N_v^+ \leftarrow N_V^+.POP( )$ 
36:         ▷ Update the vertex index ◁
37:         for  $(w, e_{ind}) \in N_v^+$  do
38:              $w\_rev \leftarrow \bar{w}$ 
39:              $e_{ind\_rev} \leftarrow \bar{e}_{ind}$ 
40:              $adj\_ind \leftarrow GET\_INDEX(N_{w\_rev}^+, e_{ind\_rev}, 1)$ 
41:              $N_{w\_rev}^+[adj\_ind] \leftarrow (v\_rev, e_{ind\_rev})$ 
42:         for  $(w, e_{ind}) \in N_{v\_rev}^+$  do
43:              $w\_rev \leftarrow \bar{w}$ 
44:              $e_{ind\_rev} \leftarrow \bar{e}_{ind}$ 
45:              $adj\_ind \leftarrow GET\_INDEX(N_{w\_rev}^+, e_{ind\_rev}, 1)$ 
46:              $N_{w\_rev}^+[adj\_ind] \leftarrow (v, e_{ind\_rev})$ 

```

---

---

**► Algorithm 15: Delete a vertex for DGF**


---

**Require:** Graph  $G = (V, E)$ , vertex  $v \in V$  (in forward orientation).

**Ensure:** Delete the vertex and its reverse such that  $V' = V \setminus \{v, \bar{v}\}$ , and

$$\forall v \in V', 0 \leq v_{ind} \leq |V'| - 2.$$

```

1: function DGF_DELETE_VERTEX( $G, v$ )
2:   ▷ Delete it from its predecessors ◁
3:   for  $(u, u_{or}, e_{ind}) \in N_v^-$  do
4:     if  $u_{or} = 0$  then
5:        $adj\_ind \leftarrow \text{GET\_INDEX}(N_u^+, e_{ind}, 1)$ 
6:       if  $adj\_ind = |N_u^+| - 1$  then
7:         ▷ Just delete the last element ◁
8:          $N_u^+.\text{POP}()$ 
9:       else
10:        ▷ Replace the edge by the last in the neighbour list ◁
11:         $N_u^+[adj\_ind] \leftarrow N_u^+.\text{POP}()$ 
12:      else
13:         $adj\_ind \leftarrow \text{GET\_INDEX}(N_u^-, \overline{e_{ind}}, 1)$ 
14:        if  $adj\_ind = |N_u^-| - 1$  then
15:          ▷ Just delete the last element ◁
16:           $N_u^-.\text{POP}()$ 
17:        else
18:          ▷ Replace the edge by the last in the neighbour list ◁
19:           $N_u^-[adj\_ind] \leftarrow N_u^-.\text{POP}()$ 
20:         $card\_edges \leftarrow card\_edges - 2$ 
21:      DELETE( $N_v^-$ )
22:      ▷ Delete it from its successors ◁
23:      for  $(w, w_{or}, e_{ind}) \in N_v^+$  do
24:        if  $w_{or} = 0$  then
25:           $adj\_ind \leftarrow \text{GET\_INDEX}(N_w^-, e_{ind}, 1)$ 
26:          if  $adj\_ind = |N_w^-| - 1$  then
27:            ▷ Just delete the last element ◁
28:             $N_w^-.\text{POP}()$ 
29:          else
30:            ▷ Replace the edge by the last in the neighbour list ◁
31:             $N_w^-[adj\_ind] \leftarrow N_w^-.\text{POP}()$ 
32:          else
33:             $adj\_ind \leftarrow \text{GET\_INDEX}(N_w^+, \overline{e_{ind}}, 1)$ 
34:            if  $adj\_ind = |N_w^+| - 1$  then
35:              ▷ Just delete the last element ◁

```

```

36:          $N_w^+.$ POP( )
37:     else
38:         ▷ Replace the edge by the last in the neighbour list      ◁
39:          $N_w^+[adj\_ind] \leftarrow N_w^+.$ POP( )
40:          $card\_edges \leftarrow card\_edges - 2$ 
41:     DELETE( $N_v^+$ )
42:     ▷ Delete the whole vertex                                     ◁
43:     if  $v = |N_V^+| - 1$  then
44:         ▷ It is the last, just pop it and its reverse           ◁
45:          $N_V^-.$ POP( )
46:          $N_V^+.$ POP( )
47:     else
48:         ▷ Replace it and its reverse by the last and its reverse ◁
49:          $N_v^- \leftarrow N_V^-.$ POP( )
50:          $N_v^+ \leftarrow N_V^+.$ POP( )
51:         ▷ Update the vertex index                               ◁
52:         for  $(u, u_{or}, e_{ind}) \in N_v^-$  do
53:             if  $u_{or} = 0$  then
54:                  $adj\_ind \leftarrow GET\_INDEX(N_u^+, e_{ind}, 1)$ 
55:                  $N_u^+[adj\_ind] \leftarrow (v, 0, e_{ind})$ 
56:             else
57:                  $u\_rev \leftarrow \bar{u}$ 
58:                  $e_{ind\_rev} \leftarrow \bar{e}_{ind}$ 
59:                  $adj\_ind \leftarrow GET\_INDEX(N_{u\_rev}^-, e_{ind\_rev}, 1)$ 
60:                  $N_{u\_rev}^-[adj\_ind] \leftarrow (v, 1, e_{ind\_rev})$ 
61:             for  $(w, w_{or}, e_{ind}) \in N_v^+$  do
62:                 if  $w_{or} = 0$  then
63:                      $adj\_ind \leftarrow GET\_INDEX(N_w^-, e_{ind}, 1)$ 
64:                      $N_w^-[adj\_ind] \leftarrow (v, 0, e_{ind})$ 
65:                 else
66:                      $w\_rev \leftarrow \bar{w}$ 
67:                      $e_{ind\_rev} \leftarrow \bar{e}_{ind}$ 
68:                      $adj\_ind \leftarrow GET\_INDEX(N_{w\_rev}^+, e_{ind\_rev}, 1)$ 
69:                      $N_{w\_rev}^+[adj\_ind] \leftarrow (v, 1, e_{ind\_rev})$ 

```

---

► **Algorithm 16: Delete a vertex for BGU**

---

**Require:** Graph  $G = (V, E)$ , vertex  $v \in V$  (in forward orientation).

---

**Ensure:** Delete the vertex and its reverse such that  $V' = V \setminus \{v, \bar{v}\}$ , and

$$\forall v \in V', 0 \leq v_{ind} \leq |V'| - 2.$$

```

1: function BGU_DELETE_VERTEX( $G, v$ )
2:   for  $(w, e_{ind}) \in N_v$  do
3:      $adj\_ind \leftarrow \text{GET\_INDEX}(N_w, e_{ind}, 1)$ 
4:     if  $adj\_ind = |N_w| - 1$  then
5:        $\triangleright$  Just delete the last element
6:        $N_w.\text{POP}()$ 
7:     else
8:        $\triangleright$  Replace the edge by the last in the neighbour list
9:        $N_w[adj\_ind] \leftarrow N_w.\text{POP}()$ 
10:   $\text{DELETE}(N_v)$ 
11:   $\triangleright$  Delete the whole vertex
12:  if  $v = |N_V| - 1$  then
13:     $\triangleright$  It is the last, just pop it
14:     $N_V.\text{POP}()$ 
15:  else
16:     $\triangleright$  Replace it by the last
17:     $N_v \leftarrow N_V.\text{POP}()$ 
18:     $\triangleright$  Update the vertex index
19:    for  $(w, e_{ind}) \in N_v$  do
20:       $adj\_ind \leftarrow \text{GET\_INDEX}(N_w, e_{ind}, 1)$ 
21:       $N_w[adj\_ind] \leftarrow (v, e_{ind})$ 
22:      if  $w > v$  then
23:         $\triangleright$  The last (the greatest) identifier becomes a fewer and
24:           $\triangleright$  breaks the identifier order
           $E_{attr}[e_{ind}] \leftarrow (E_{attr}[e_{ind}][0], 1 - E_{attr}[e_{ind}][1])$ 

```

#### 4.1.6 Deleting an Edge

##### ► Algorithm 17: Delete an edge for DGS

**Require:** Graph  $G = (V, E)$ , edge  $(u, v) \in E$  and the edge's index  $e_{ind}$ .

**Ensure:**  $|N_u^+ \setminus edge| = |N_u^+| - 1$  and  $|N_v^- \setminus edge| = |N_v^-| - 1$ .

```

1: function DGS_DELETE_EDGE( $G, u, v, e_{ind}$ )
2:    $\triangleright$  Remove v from u's successors
3:    $adj\_ind \leftarrow \text{GET\_INDEX}(N_u^+, e_{ind}, 1)$ 
4:   if  $adj\_ind = |N_u^+| - 1$  then

```

---

```

5:     ▷ Just delete the last element                                ◁
6:      $N_u^+.$ POP( )
7:   else
8:     ▷ Replace the edge by the last in the neighbour list        ◁
9:      $N_u^+[adj\_ind] \leftarrow N_u^+.$ POP( )
10:  ▷ Remove  $u$  from  $v$ 's predecessors.                               ◁
11:   $v\_rev \leftarrow \bar{v}$ 
12:   $adj\_ind \leftarrow \text{GET\_INDEX}(N_{v\_rev}^+, \overline{e_{ind}}, 1)$ 
13:  if  $adj\_ind = |N_{v\_rev}^+| - 1$  then
14:    ▷ Just delete the last element                                ◁
15:     $N_{v\_rev}^+.$ POP( )
16:  else
17:    ▷ Replace the edge by the last in the neighbour list        ◁
18:     $N_{v\_rev}^+[adj\_ind] \leftarrow N_{v\_rev}^+.$ POP( )
19:   $card\_edges \leftarrow card\_edges - 2$ 

```

---

► **Algorithm 18: Delete an edge for DGF**

---

**Require:** Graph  $G = (V, E)$ , edge  $(u, v) \in E$ , with their orientation  $(u_{or}, v_{or}) \in \{0; 1\}^2$ , and the edge's index  $e_{ind}$ .

**Ensure:**  $|N_u^+ \setminus edge| = |N_u^+| - 1$  and  $|N_v^- \setminus edge| = |N_v^-| - 1$ .

```

1: function DGF_DELETE_EDGE( $G, u, u_{or}, v, v_{or}, e_{ind}$ )
2:   ▷ Remove  $v$  from  $u$ 's successors                                ◁
3:   if  $u_{or} = 0$  then
4:      $list\_succs \leftarrow N_u^+$ 
5:      $adj\_ind \leftarrow \text{GET\_INDEX}(list\_succs, e_{ind}, 2)$ 
6:   else
7:      $list\_succs \leftarrow N_u^-$ 
8:      $adj\_ind \leftarrow \text{GET\_INDEX}(list\_succs, \overline{e_{ind}}, 2)$ 
9:   if  $adj\_ind = |list\_succs| - 1$  then
10:    ▷ Just delete the last element                                ◁
11:     $list\_succs.$ POP( )
12:   else
13:    ▷ Replace the edge by the last in the neighbour list        ◁
14:     $list\_succs[adj\_ind] \leftarrow list\_succs.$ POP( )
15:   ▷ Remove  $u$  from  $v$ 's predecessors.                               ◁
16:   if  $v_{or} = 0$  then
17:      $list\_preds \leftarrow N_v^-$ 
18:      $adj\_ind \leftarrow \text{GET\_INDEX}(list\_preds, e_{ind}, 2)$ 

```

---

---

```

19:  else
20:     $list\_preds \leftarrow N_v^+$ 
21:     $adj\_ind \leftarrow \text{GET\_INDEX}(list\_preds, \overline{e_{ind}}, 2)$ 
22:    if  $adj\_ind = |list\_preds| - 1$  then
23:       $\triangleright$  Just delete the last element ◁
24:       $list\_preds.POP()$ 
25:    else
26:       $\triangleright$  Replace the edge by the last in the neighbour list ◁
27:       $list\_preds[adj\_ind] \leftarrow list\_preds.POP()$ 
28:     $card\_edges \leftarrow card\_edges - 2$ 

```

---



---

► **Algorithm 19: Delete an edge for BGU**

---

**Require:** Graph  $G = (V, E)$ , edge  $(u, v) \in E$ , with their orientation  $(u_{or}, v_{or}) \in \{0; 1\}^2$ , and the edge's index  $e_{ind}$ .

**Ensure:**  $|N_u^+ \setminus edge| = |N_u^+| - 1$  and  $|N_v^- \setminus edge| = |N_v^-| - 1$ .

```

1: function BGU_DELETE_EDGE( $G, u, u_{or}, v, v_{or}, e_{ind}$ )
2:    $e'_{ind} \leftarrow e_{ind} \div 2$  ◁ Euclidian division
3:    $\triangleright$  Remove v from u's neighbours. ◁
4:    $adj\_ind \leftarrow \text{GET\_INDEX}(N_u, e'_{ind}, 1)$ 
5:   if  $adj\_ind = |N_u| - 1$  then
6:      $\triangleright$  Just delete the last element ◁
7:      $N_u.POP()$ 
8:   else
9:      $\triangleright$  Replace the edge by the last in the neighbour list ◁
10:     $N_u[adj\_ind] \leftarrow N_u.POP()$ 
11:     $\triangleright$  Remove u from v's neighbours. ◁
12:     $adj\_ind \leftarrow \text{GET\_INDEX}(N_v, e'_{ind}, 1)$ 
13:    if  $adj\_ind = |N_v| - 1$  then
14:       $\triangleright$  Just delete the last element ◁
15:       $N_v.POP()$ 
16:    else
17:       $\triangleright$  Replace the edge by the last in the neighbour list ◁
18:       $N_v[adj\_ind] \leftarrow N_v.POP()$ 
19:     $card\_edges \leftarrow card\_edges - 1$ 

```

---



---

► **Algorithm 20: Find place of edge index in the neighbour list. Cost:**  
**worst**  $3 \times |list|$ , **best** 3, **average**  $3 \times \left\lceil \frac{|list|}{2} \right\rceil$

---

**Require:** List of tuple  $list$ , edge index  $e_{ind}$ , index  $t_{ind}$  of the place of edge index in the tuple contained in  $list$ . The edge index must be in a tuple.

**Ensure:** Returns the index of the tuple containing  $e_{ind}$  in  $list$ .

```

1: function GET_INDEX( $list, e_{ind}, t_{ind}$ )
2:    $list_{ind} \leftarrow 0$ 
3:   while  $list[list_{ind}][t_{ind}] \neq e_{ind}$  and  $list_{ind} < |list| - 1$  do
4:      $list_{ind} \leftarrow list_{ind} + 1$ 
5:   return  $list_{ind}$ 

```

---

## 4.2 Time Complexities

■ **Table 1** – Algorithmic costs of iterating over the neighbours for DGS, DGF and BGU.

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

**complete the cases**

— DGS best case:

■ **Table 2** – 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 3** – Algorithmic costs of deleting an edge for DGS, DGF and BGU.

	Best	Worst	Average
DGS	17	$3 \times (o_u^+ + o_v^-) + 13$	$3 \times \left( \left\lceil \frac{o_u^+}{2} \right\rceil + \left\lceil \frac{o_v^-}{2} \right\rceil \right) + 13 - \frac{1}{o_u^+} - \frac{1}{o_v^-}$
DGF	17	$3 \times (o_u^- + o_v^+) + 17$	$\frac{3}{2} \times \left( \left\lceil \frac{o_u^-}{2} \right\rceil + \left\lceil \frac{o_u^+}{2} \right\rceil + \left\lceil \frac{o_v^-}{2} \right\rceil + \left\lceil \frac{o_v^+}{2} \right\rceil \right) + 13 - 2 \times \left( \frac{1}{o_u} + \frac{1}{o_v} \right)$
BGU	14	$3 \times (o_u + o_v) + 10$	$3 \times \left( \left\lceil \frac{o_u}{2} \right\rceil + \left\lceil \frac{o_v}{2} \right\rceil \right) + 11 - \frac{1}{o_u} - \frac{1}{o_v}$

— DGS worst case:

$$\begin{aligned}
 & 9 \times o_v \\
 & + 3 \times \left( \sum_{u \in N_v^-} o_u^+ + \sum_{w \in N_v^+} o_w^- + \sum_{x \in N_z^-} o_x^+ + \sum_{y \in N_z^+} o_y^- \right) \\
 & + 5 \times (o_z + 1)
 \end{aligned}$$

— DGS average case

$$\begin{aligned}
& 10 \times o_v + \frac{10 \times \mathcal{R}_{aw} - 5}{2 \times \mathcal{R}_{aw}} \times o_z \\
& + 3 \times \left( \sum_{u \in N_v^-} \left\lceil \frac{o_u^+}{2} \right\rceil + \sum_{w \in N_v^+} \left\lceil \frac{o_w^-}{2} \right\rceil \right) \\
& + \frac{6 \times \mathcal{R}_{aw} - 3}{2 \times \mathcal{R}_{aw}} \times \left( \sum_{x \in N_z^-} \left\lceil \frac{o_x^+}{2} \right\rceil + \sum_{y \in N_z^+} \left\lceil \frac{o_y^-}{2} \right\rceil \right) \\
& - \left( \sum_{u \in N_v^-} \frac{1}{o_u^+} + \sum_{w \in N_v^+} \frac{1}{o_w^-} \right) \\
& + 5
\end{aligned}$$

— DGF best case:

$$4$$

— DGF worst case:

$$\begin{aligned}
& 9 \times o_v + 4 \times o_z \\
& + 3 \times \left( \sum_{u \in N_v^-} o_u^- + \sum_{w \in N_v^+} o_w^+ + \sum_{x \in N_z^-} o_x^- + \sum_{y \in N_z^+} o_y^+ \right) \\
& + 4
\end{aligned}$$

— DGF average case

$$\begin{aligned}
& \frac{15}{2} \times o_v + \frac{6 \times \mathcal{R}_{aw} - 3}{2 \times \mathcal{R}_{aw}} \times o_z \\
& + \frac{3}{2} \times \sum_{u \in N_v^- \cup N_v^+} \left( \left\lceil \frac{o_u^-}{2} \right\rceil + \left\lceil \frac{o_u^+}{2} \right\rceil \right) \\
& + \frac{6 \times \mathcal{R}_{aw} - 3}{4 \times \mathcal{R}_{aw}} \times \sum_{x \in N_z^- \cup N_z^+} \left( \left\lceil \frac{o_x^-}{2} \right\rceil + \left\lceil \frac{o_x^+}{2} \right\rceil \right) \\
& - \frac{1}{2} \times \left( \sum_{u \in N_v^-} \frac{1}{o_u^+} + \sum_{w \in N_v^+} \frac{1}{o_w^-} \right) \\
& + \frac{6 \times \mathcal{R}_{aw} - 1}{\mathcal{R}_{aw}}
\end{aligned}$$

- BGU best case:
- BGU worst case:
- BGU average case:

$$\begin{aligned}
& 5 \times o_v + \frac{3 \times \mathcal{R}_{aw} - 3}{\mathcal{R}_{aw}} \times o_z \\
& + 3 \times \sum_{w \in N_v} \left\lceil \frac{o_w}{2} \right\rceil + \frac{3 \times \mathcal{R}_{aw} + 3}{2 \times \mathcal{R}_{aw}} \times \sum_{y \in N_z} \left\lceil \frac{o_y}{2} \right\rceil \\
& - \sum_{w \in N_v} \frac{1}{o_w} \\
& + \frac{4 \times \mathcal{R}_{aw} - 1}{\mathcal{R}_{aw}}
\end{aligned}$$



## References

- [1] John Dimitri Kececioglu. *Exact and Approximation Algorithms for DNA Sequence Reconstruction*. The University of Arizona, 1991.
- [2] Chen-Shan Chin, Paul Peluso, Fritz J. Sedlazeck, Maria Nattestad, Gregory T. Concepcion, Alicia Clum, Christopher Dunn, Ronan O'Malley, Rosa Figueroa-Balderas, Abraham Morales-Cruz, Grant R. Cramer, Massimo Delledonne, Chongyuan Luo, Joseph R. Ecker, Dario Cantu, David R. Rank, and Michael C. Schatz. Phased diploid genome assembly with single-molecule real-time sequencing. *Nature Methods*, 13(12):1050–1054, December 2016.
- [3] Govinda M. Kamath, Ilan Shomorony, Fei Xia, Thomas A. Courtade, and David N. Tse. HINGE: Long-read assembly achieves optimal repeat resolution. *Genome Research*, 27(5):747–756, January 2017.
- [4] Rumen Andonov, Hristo Djidjev, Sebastien François, and Dominique Lavenier. Complete assembly of circular and chloroplast genomes based on global optimization. *Journal of Bioinformatics and Computational Biology*, 17(3):1950014, June 2019.
- [5] Kishwar Shafin, Trevor Pesout, Ryan Lorig-Roach, Marina Haukness, Hugh E. Olsen, Colleen Bosworth, Joel Armstrong, Kristof Tigyi, Nicholas Maurer, Sergey Koren, Fritz J. Sedlazeck, Tobias Marschall, Simon Mayes, Vania Costa, Justin M. Zook, Kelvin J. Liu, Duncan Kilburn, Melanie Sorensen, Katy M. Munson, Mitchell R. Vollger, Jean Monlong, Erik Garrison, Evan E. Eichler, Sofie Salama, David Haussler, Richard E. Green, Mark Akeson, Adam Phillippy, Karen H. Miga, Paolo Carnevali, Miten Jain, and Benedict Paten. Nanopore sequencing and the Shasta toolkit enable efficient de novo assembly of eleven human genomes. *Nature Biotechnology*, 38(9):1044–1053, September 2020.
- [6] Haoyu Cheng, Gregory T. Concepcion, Xiaowen Feng, Haowen Zhang, and Heng Li. Haplotype-resolved de novo assembly using phased assembly graphs with hifiasm. *Nature Methods*, 18(2):170–175, February 2021.
- [7] E. W. Myers. Toward simplifying and accurately formulating fragment assembly. *Journal of Computational Biology: A Journal of Computational Molecular Cell Biology*, 2(2):275–290, 1995.
- [8] Daniel D. Sommer, Arthur L. Delcher, Steven L. Salzberg, and Mihai Pop. Minimus: A fast, lightweight genome assembler. *BMC bioinformatics*, 8:64, February 2007.

- [9] David Hernandez, Patrice François, Laurent Farinelli, Magne Østerås, and Jacques Schrenzel. De novo bacterial genome sequencing: Millions of very short reads assembled on a desktop computer. *Genome Research*, 18(5):802–809, January 2008.
- [10] Leena Salmela, Veli Mäkinen, Niko Välimäki, Johannes Ylinen, and Esko Ukkonen. Fast scaffolding with small independent mixed integer programs. *Bioinformatics*, 27(23):3259–3265, December 2011.
- [11] Eugene W. Myers. The fragment assembly string graph. *Bioinformatics*, 21(suppl\_2):ii79–ii85, January 2005.
- [12] Veli Mäkinen, Djamel Belazzougui, Fabio Cunial, and Alexandru I. Tomescu. *Genome-Scale Algorithm Design*. Cambridge University Press, May 2015.
- [13] Heng Li. Minimap and miniasm: Fast mapping and de novo assembly for noisy long sequences. *Bioinformatics*, 32(14):2103–2110, July 2016.

## Acronyms

**B | C | D**

### B

**BG** Unoriented Fragments Bi-directed Graph 31, 32

### C

**CSC** Compressed Sparse Column 9

**CSR** Compressed Sparse Row 9

### D

**DGA** All Oriented Fragments Directed Graph 22, 23, 25, 27

**DGF** Forward Fragments Directed Graph 27, 28, 30, 31

**DGS** Only Oriented Fragments' Successors Directed Graph 25–27

**DNA** Deoxyribonucleic Acid 1, 5, 7

## Symbols

**Attributes | Functions | Graph | Operations | Sets**

### Attributes

*f* Forward orientation 6–8, 12, 13, 15, 18, 19, 22, 28, 34, 37, 38, 40, see [forward](#)

*ind* Index 22, 23, 25, 27, 28, 30, 31, 35–38, 41–48

*indor* Index with boolean orientation 27, 28

*or* Orientation 11, 12, 15–17, 20, 22, 23, 27, 28, 36–38, 40, 41, 43, 44, 46, 47, see [forward & reverse](#)

*r* Reverse orientation 6–8, 12, 13, 15, 18, 19, 22, 30, 34, 37, 38, 40, see [reverse](#)

*rid* Read identifier 5, 6, 11, 12, 14, 16, 19, 20, 22, 27, 28, 31, 33, 34

*seq* Nucleotide sequence 5–7



## Functions

`strand_iid` Read identifier function 5–7

## Graph

$G$  Graph object 11–13, 15, 17, 20, 22, 36–41, 43–47, see [undirected graph](#), [directed graph](#), [bi-directed graph](#), [vertex](#) & [edge](#)

$V$  Vertices set 11–17, 19, 20, 22, 23, 25–28, 30, 31, 34, 36–47, see [vertex](#)

$E$  Edges set 11–23, 27, 28, 30, 31, 36–41, 43–47, see [edge](#)

$N$  Neighbours set 31, 36, 38, 39, 41, 45, 47, 51, see [edge](#)

$N^-$  Predecessors set 23, 28, 36, 38–40, 43–47, 49, 50, see [edge](#),  $N$  &  $N^+$

$N^+$  Successors set 23, 25, 28, 36, 37, 39–47, 49, 50, see [edge](#),  $N$  &  $N^-$

## Operations

$\bar{\cdot}$  Reverse operation 6–9, 12–14, 16, 17, 20, 21, 23, 27, 28, 30, 35, 36, 38, 40–47, see [reverse](#) & [rev](#)

## Sets

$\mathbb{N}$  The set of natural (positive) integer 5, 6, 11, 13, 14, 17, 19, 20

$\mathcal{O}$  Set of overlaps 8, 9, 12, 14, 15, 19–23, 25, 28, 29, 31–33

$\mathcal{R}_{aw}$  Set of raw reads 5–7, 12, 15, 20, 22, 23, 25, 27, 28, 30–34, 50, 51

$\mathcal{R}$  Set of reads 6–9, 12, 14, 15, 19–21

$\mathcal{R}_{ev}$  Set of reversed raw reads 6

$\Sigma$  Alphabet set 5

# Glossary

**A | B | C | D | E | F | G | N | O | P | R | S | U | V | W**

## A

**alignment** Nucleotide comparison between at least two sequences [7](#), [8](#)

**assembly** Method used to reconstruct genomes from fragments [1](#), [6](#), [7](#), [11](#), [14](#)

**assembly graph** One of the output of genome assembly method see [assembler](#)

## B

**bi-directed graph** Graph that contains 3 edge types [1](#), [11](#), [14–17](#), [20](#), [31](#), [32](#)

## C

**contig** Merge, consensus of reads obtain by assembly method(s) see [assembler](#)

## D

**directed graph** Graph where the edges are directed [1](#), [11–13](#), [22–24](#), [26](#), [27](#), [29](#), [33](#), [34](#)

## E

**edge** Component of a graph that connects two vertices [9](#), [11–23](#), [25](#), [27](#), [28](#), [30](#), [31](#), [33](#), [34](#), see [graph](#) & [vertex](#)

## F

**forward** Original sequence orientation [6–8](#), [11](#), [14](#), [15](#), [18](#), [19](#), [21](#), [27–29](#), [31](#), [32](#), [34](#), see [reverse](#)

**fragment** Generic nucleotide fragment [1](#), [5](#), [11–13](#), [20–22](#), [24](#), [26](#), [29](#)

## G

**genome** Entire set of DNA instructions found in a cell [7](#)

**graph** Object composed of vertices connected by edges [1](#), [9](#), [11–14](#), [16–18](#), [20–29](#), [31](#), [34](#), see [vertex](#) & [edge](#)

**N**

**neighbour** Vertex connected to a given one 11, 27, 31–34

**nucleotide** Basic building block of nucleic acids (RNA and DNA) 5–8, see DNA

**O**

**overlap** Suffix-prefix alignment type between two sequences 1, 7–9, 11, 12, 14, 15, 18–27, 29, 31–34, see alignment

**P**

**path** A sequence of vertices such that two consecutive vertices are connected by an edge in the graph 12, 13, 17, 20, 21, see walk, graph, vertex & edge

**predecessor** In-vertex of one edge of one given vertex 23, 24, 26–31, 34, see successor

**R**

**read** DNA fragment from one DNA's strand, output by a sequencer 5–7, 9, 11, 12, 14, 15, 18–22, 27, 28, 30, 31, 33, 34, see sequencer

**reverse** Reverse-complement sequence orientation 1, 6, 7, 9, 11–13, 16, 18–22, 25–27, 30, 31, see forward

**S**

**sequencer** Sequencing technology machine 5

**sequencing** Method to generate nucleotide fragments 5–7

**strand** The DNA molecule is made up of two strands, each of which has a complementary sequence to the other 5–7

**string graph** Graph structure that stores overlaps between reads see overlap

**succession relationship** When moving from one to the other is feasible, two things have a succession relationship 1, 7, 8, 11, 15, 16

**successor** Out-vertex of one edge of one given vertex 23–31, 34, see predecessor

**U**

**undirected graph** Graph where the edges are undirected 11, 16–21, 31

**V**

**vertex** Component of a graph 9, 11–23, 25–28, 30, 31, 33, 34, see [graph & edge](#)

**W**

**walk** A sequence of vertices such that two consecutive vertices are connected by an edge in the graph 12, 13, 15–18, 20, 21, see [path](#), [graph](#), [vertex & edge](#)