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-03815190v3

HAL Id: hal-03815190
https://inria.hal.science/hal-03815190v3
Preprint submitted on 26 Oct 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 Graph for Assembling and Scaffolding Algorithms: Paradigm Review and Implementation Proposals

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

26th 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

1 Introduction ................................................................. 5
   1.1 Reads ....................................................................... 5
   1.2 Overlaps .................................................................... 7

2 Storing Overlaps in a Graph Structure .............................. 9
   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

3 Overlap Graph Implementations ....................................... 22
   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

4 Memory and Time Costs .................................................. 35
   4.1 Algorithms .............................................................. 35
      4.1.1 Subfunctions ..................................................... 35
      4.1.2 Iterating Over the Predecessors ............................. 36
      4.1.3 Iterating Over the Successors ............................... 37
      4.1.4 Adding a Vertex ................................................. 39
      4.1.5 Adding an Edge .................................................. 40
      4.1.6 Deleting a Vertex ............................................... 41
      4.1.7 Deleting an Edge ............................................... 45
   4.2 Time Complexities .................................................... 48
      4.2.1 Complexities Calculus Details ............................... 48
      4.2.2 Costs for Subfunctions ....................................... 48
      4.2.3 Iterating Over the Neighbours ............................... 48
      4.2.4 Costs for Dynamics .......................................... 49

References ................................................................. 53

V. Epain ................................................................. 3
CONTENTS

Acronyms i
Symbols i
Glossary ii
1.1 Reads

A DNA fragment obtained from a sequencing stage is denoted by read.

Definition 1.1: Raw reads

Let $R_{aw}$ be the set of raw reads. $\forall r \in R_{aw}$, 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 if they have been sequenced from the same strand.

The Definition 1.2 formalises Axiom 1.1 point iii.:

Definition 1.2: Strand identifier

Let $\text{strand} _{iid} : R_{aw} \rightarrow \{0; 1\}$ 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 ($A \leftrightarrow T, G \leftrightarrow C$). 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 $\mathcal{R}_{rev} = \{ u_r \mid u_f \in \mathcal{R}_{aw} \}$ denote the set of reversed reads. Thus $\mathcal{R} = \mathcal{R}_{aw} \cup \mathcal{R}_{rev}$. 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 $u_r \in \mathcal{R}_{rev}$)

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 $u \in \mathcal{R}$, $\overline{u}$ denotes its reverse, where:

— $\overline{u}_{rid} = u_{rid}$ (i.e. a read $u$ and its reverse $\overline{u}$ have the same identifier)

— $\overline{u}_{seq} = \overline{u}_{seq} = b_1b_2\cdots b_n = \overline{b}_n \cdots \overline{b}_1$ where $n \in \mathbb{N}$ is the length of the nucleotides sequence, and $\forall i \in [1; n], b_i$ is the $i^{th}$ nucleotide base and
1.2 Overlaps

\[ \bar{b}_i \text{ its complementary nucleotide base.} \]

Note that \( \bar{u}_f = u_r \) and \( \bar{u}_r = u_f \).

\[ \bar{R}_{aw} \]

\[ u_f \]

\[ v_f \]

\[ \text{strands}_0 \]

\[ \text{strands}_1 \]

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

\( u_f, v_f \) are two reads from the raw reads set \( R_{aw} \) 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 \( \text{strands}_0 \) \( (\text{strand}_\text{iid}(u_f) = 0) \) while \( v_f \) was sequenced from strand \( \text{strands}_1 \) \( (\text{strand}_\text{iid}(v_f) = 1) \).

**Axiom 1.2: Belonging to one strand**

The definition of \( R \) set and Axiom 1.1 point ii. give the following statement:

\[ \forall r \in R, \text{strand}_\text{iid}(r) = 1 - \text{strand}_\text{iid}(r) \]

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 \( u_{\text{seq}} \) and the beginning of sequence \( v_{\text{seq}} \).
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:

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

**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 $v$ and $u$ 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.
1.2 Overlaps

▶ Definition 1.6: The reverse operation for overlaps

For a given overlap \((q, r) \in O\), \((r, q)\) denotes its reverse i.e. \((r, q)\).

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 \(R^2\)). 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.
1 INTRODUCTION

1.2 Overlaps

Oriented Fragment based
Directed Graph paradigm chronology
Classical digraph
Reverse symmetry with forward focus
All oriented fragments digraph
Forward fragments digraph
(ungraph with weighted edges)

Classical ungraph
Forward fragments bigraph
(Oriented Walk based
Bi-directed (Undirected)
Adjacency list splitted
by values

Tail-based Fragment
based
Undirected
Classical ungraph with separated edge types

Graph paradigm implementations
ordered by close logic
Set of raw reads (fragments)
Set of overlaps
Vertices set
Edges set
Graph paradigm
Theoretical graph type
Theoretical number of vertices and edges

Graph paradigm implementation
Vertices sets of all oriented fragments
and only forward fragments
Overlap-edges set
Read-edges set

Adjacency list
boolean value

Adjacency list
reverse symmetry
with forward focus

Adjacency list splitted
by read values

Predecessors list removal
Read-edges jump operation

Informal introduction to oriented fragment-based paradigms.
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 Kececioglu [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.

Fix Oriented Fragments Based and Tail-head Fragments based

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:
  - \( v_{\text{rid}} \in \mathbb{N} \) the identifier of the read it represents
  - \( v_{\text{or}} \in \{0; 1\} \) 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.
2 GRAPH PARADIGMS

2.1 Digraph: Oriented Fragments Based

Figure 4 – Overlap cases in the directed graph.

\[ (u_f, v_f) \in \mathcal{O} \]  
\[ (v_f, u_f) \in \mathcal{O} \]  
\[ (u_f, v_r) \in \mathcal{O} \]  
\[ (v_r, u_f) \in \mathcal{O} \]

\[ (v_r, u_r) \in \mathcal{O} \]  
\[ (u_r, v_r) \in \mathcal{O} \]  
\[ (v_f, u_f) \in \mathcal{O} \]  
\[ (u_r, v_f) \in \mathcal{O} \]  

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 \( p \):

\[
\forall v \in p, \overline{v} \notin p
\]

Note that walking in this graph is quite similar to 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.](image)

A valid walk that begins from vertex \( v_f \) is coloured in red. This walk results in a path \( p = v_fu_fx_r \).

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**

1. \( \forall v \in V, \overline{v} \in V \).
2. \( \forall (u, v) \in E, (\overline{u}, \overline{v}) = (\overline{v}, \overline{u}) \in E \).
Proof

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

\[\square\]

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 that this 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.
2.2 Bigraph: Oriented Walk Based

(a) \((u_f, v_f) \in O\)

(b) \((v_f, u_f) \in O\)

(c) \((u_f, v_r) \in O\)

(d) \((v_r, u_f) \in O\)

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:

- \(|V| = |R_{uv}| = \frac{1}{2} \times |R|\)
- \(|E| = \frac{1}{2} \times |O|\)

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:
  \[
u_{or} = \begin{cases} \text{rel}_{uv} & \text{if } u_{rid} < v_{rid} \\ |(1 - or_{uv}) - \text{rel}_{uv}| & \text{if } u_{rid} > v_{rid} \end{cases}\]

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 \(v_{or}\), \(\overline{v}\) denotes its reverse, where:

- \(\overline{v}_{rid} = v_{rid}\)
- \(\overline{v}_{or} = 1 - v_{or}\)

**Edges** For a given edge \(e = (u, v) \in E\) and fix co-valid orientation variables \(u_{or}\) and \(v_{or}\), \(\overline{e} = (\overline{v}, \overline{u})\) 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 $p = v_0 v_1 \cdots v_{n-1} \in V^n \mid n \in \mathbb{N}$ 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 that this 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.

\[\square\]

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
2.3 Ungraph: Tail-Head Fragments Based 2 GRAPH PARADIGMS


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:
  - $v_{\text{rid}} \in \mathbb{N}$ the identifier of the read it represents
  - $v_{\text{ext}} \in \{0; 1\}$ the extremity of the read it represents (equals to 0 if it is the head, else equals to 1 if it is the tail)

- $E = E^R \cup E^O$ is the set of edges, such that:
  - each undirected read-edge $(v_t, v_h) \in E^R$ encodes the two possible read orientations (forward if from $v_t$ to $v_h$, else reverse)
  - each undirected overlap-edge $(u, v) \in E^O$ represents which extremity of read $u$ is overlapping which extremity of read $v$

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.](image)

$u_t, u_h$ 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 $v \in V$, $\overline{v}$ denotes its reverse, where:

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

Edges For a given edge $e = (u, v) \in E$, $\overline{e} = (v, u)$ 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 $p = v_0v_1 \cdots v_{2k-1} \in V^{2k} \mid k \in \mathbb{N}$ in the graph $G = (V, E^R \cup E^O)$ that respects the following criteria:

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

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

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 to 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.

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 \( p_x = x_t x_h u_h u_t v_t v_h \) 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 \((u, v) \in E^O\) and the two read-edges \((u_t, u_h) \in E^R\) and \((v_t, v_h) \in E^R\), 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 \((u, v) \in E^O\) be an overlap-edge, hence read-edges \((u, \overline{v})\) and \((v, \overline{u})\) are considered. Moreover, overlap-edge \((v, u)\) is also in \( E^O \), hence read-edges \((u, \overline{v})\) and \((\overline{u}, v)\) are considered, and they correspond to the
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 $O$ are kept, and reads in reads set $R_{aw}$ participate in at least one overlap. Also let suppose that $\min_{r \in R_{aw}} r_{rid} = 0$ and $\max_{r \in R_{aw}} r_{rid} = |R_{aw}| - 1$. 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 $r \in R_{aw}$ 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 $V_{ind} = [0; 2 \times |R_{aw}|]$ be the set of indices for the vertices:

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

Let $E_{ind} = \left[0; |O|\right]$ be the set of indices for the edges:

— $|E_{ind}| = |O|$

— $\forall e_{ind} \in E_{ind}, e_{ind}$ corresponds to the index of edge $e$, such that $|e_{ind} - \overline{e_{ind}}| = 1$

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.

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:

$$Mem(DGA) = 2 \times (2 \times |R_{aw}| + 1) \times P$$

$$+ 2 \times |O| \times \left(\left\lceil \frac{1 + \log_2 |R_{aw}|}{8} \right\rceil + \left\lceil \frac{\log_2 |O|}{8} \right\rceil \right)$$

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.

Definition 3.3: DGS implementation

DGS graph implementation is built by the successor list \( N_v^+ \) such that \( \forall v \in V, \forall w \in N_v^+ \), \( w_{\text{ind}} \) 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 \( \text{Mem}(DGS) \) (in octets) is equals to:

\[
\text{Mem}(DGS) = (2 \times |R_{aw}| + 1) \times P + |O| \times \left( \left\lceil \frac{1 + \log_2 |R_{aw}|}{8} \right\rceil + \left\lceil \frac{\log_2 |O|}{8} \right\rceil \right)
\]

where \( P \) is the memory size of a memory address.

Proof

For each vertex there is a pointer to its successor list, hence \( 2 \times |R_{aw}| \) pointers, and there is one pointer to the vertices’ successor list. Then, for each successor, its index is provided (\( \left\lceil \log_2 2 \times |R_{aw}| \right\rceil \) bits), and the edge index is provided too (\( \left\lceil \log_2 |O| \right\rceil \) 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 (\(|O|\)).

\( \Box \)
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
3.1 Digraph: Oriented Fragments Based

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

\[
\begin{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, \(\bar{u} = u\), that is the predecessor of \(v\).

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

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 \(\mathcal{R}_{raw}\)). Therefore, the vertices indices set is not build as previously. Hence for each read \(r \in \mathcal{R}_{raw}\), 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.

\[
\begin{definition}

\textbf{Definition 3.4: Graph indices set for DGA}

Let \(Vind = [0; |\mathcal{R}_{aw}|]\) be the set of indices for the vertices:

- \(|Vind| = |\mathcal{R}_{aw}|\)

- \(\forall v_{ind} \in Vind, v_{ind} = v_{rid}\), where \(v_{rid}\) corresponds to the read’s identifier

Let \(Vindor = [0; |\mathcal{R}_{aw}|] \times \{0; 1\}\) be the set of indices enriched by a boolean value for the oriented vertices:

- \(|Vindor| = 2 \times |\mathcal{R}_{aw}|\)

\end{definition}

V. Epain
— \( \forall v_{\text{indor}} \in V_{\text{indor}}, v_{\text{indor}} = (v_{\text{rid}}, v_{\text{or}}) \), where \( v_{\text{rid}} \) corresponds to the read’s identifier and \( v_{\text{or}} \) to its orientation

Let \( E_{\text{ind}} = [0; |O|] \) be the set of indices for the edges:

— \( |E_{\text{ind}}| = |O| \)
— \( \forall e_{\text{ind}} \in E_{\text{ind}}, e_{\text{ind}} \) corresponds to the index of edge \( e \), such that \( |e_{\text{ind}} - \overline{e_{\text{ind}}}| = 1 \)

Definition 3.5: DGF implementation

DGF graph implementation is built by:

**Predecessor list for the forward** \( N^-_{V_f} \) \( \forall v \in V_f, \forall u \in N^-_{v} \), \( u_{\text{ind}} \) is the index of a predecessor of vertex \( v \), and \( u_{\text{or}} \) gives its orientation

**Successor list for the forward** \( N^+_Vf \) \( \forall v \in V_f, \forall w \in N^+_v \), \( w_{\text{ind}} \) is the index of a successor of vertex \( v \), and \( w_{\text{or}} \) gives its orientation

Proposition 3.4: DGF memory consumption

The memory size of the graph \( \text{Mem}(DGF) \) (in octets) is equals to:

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

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 (\( \lceil \log_2 |O| \rceil \) bits). These numbers are divided by eight and rounded up to the nearest integer to get the number of
3.1 Digraph: Oriented Fragments Based

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.

\[ G = (V, E) \]

\[
\forall v \in V, v_{\text{ind}} = (v_{\text{ind}}, v_{\text{or}}) \in \text{IndOr} \\
\bar{v}_{\text{ind}} = (v_{\text{ind}}, 1 - v_{\text{or}}) \\
(v, u) \in E \iff (u, v) \in O \\
\iff (\bar{v}, u) \in E
\]

\[ u_f \rightarrow v_f \leftrightarrow (v_r \rightarrow u_r) \\
0 \quad 1 \quad 0 \quad 1 \]

\[ u_f \leftarrow v_f \leftrightarrow (v_r \leftarrow u_r) \\
0 \quad 1 \quad 0 \quad 1 \]

\[ u_f \rightarrow v_r \leftrightarrow (u_f \rightarrow u_r) \\
0 \quad 1 \quad 0 \quad 1 \]

\[ u_f \leftarrow v_r \leftrightarrow (u_f \leftarrow u_r) \\
0 \quad 1 \quad 0 \quad 1 \]

Then the sum is multiplied by the number of overlaps \((|O|)\).
Note that in contrast to Proposition 3.2, the memory consumption of adjacency lists pointers is equal to $2 \times (|R_{aw}| + 1)$ (versus $(2 \times |R_{aw}| + 1)$). In fact, the extra $P$ is due to an extra adjacency list.

Proposition 3.5 answer the issue of getting the predecessors succecssors of the reverse.

\begin{proposition} \textbf{DGF reverse symmetry to retrieve predecessors} \\
For each vertex $v \in V_r$:

\begin{enumerate}
  \item its predecessors are the reverse of its reverse’ successors
  \item its successors are the reverse of its reverse’ predecessors
  \item the edge index of $(u, v) \in E$ is the reverse of this of $(\bar{v}, \bar{u}) \in E$
\end{enumerate}
\end{proposition}

\begin{proof}
Let $v \in V_r^\mathcal{R}$ be a vertex that represents a reverse read.

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

\begin{enumerate}
  \item $(\bar{v}, \bar{u}) \in E$, 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 $v$.
  \item $e_{ind}$ 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 $(u, v)$.
\end{enumerate}

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

\begin{enumerate}
  \item $(\bar{w}, \bar{v}) \in E$, 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 $v$.
  \item $e_{ind}$ 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 $(v, w)$.
\end{enumerate}
\end{proof}
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 \( R_{aw} \). 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 \( V_{ind} = [0; |R_{aw}|] \) be the set of indices for the vertices:
- \( |V_{ind}| = |R_{aw}| \)
- \( \forall v_{ind} \in V_{ind}, v_{ind} = v_{rid} \), where \( v_{rid} \) corresponds to the read’s identifier

Let \( E_{ind} = [0; \frac{1}{2} \times |O|] \) be the set of indices for the edges:
- \( |E_{ind}| = \frac{1}{2} \times |O| \)
- \( \forall e_{ind} \in E_{ind}, e_{ind} \) corresponds to the index of edge \( e \)

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 — All four overlap cases structured in the bidirected 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 $o_{uv}$ and the value of $rel_{uv}$.

**Proposition 3.6: BG memory consumption**

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

$$Mem(BG) = (|R_{aw}| + 2) \times P$$

$$+ |O| \times \left( \left\lceil \frac{\log_2 |R_{aw}|}{8} \right\rceil + \left\lceil \frac{\log_2 |O| - 1}{8} \right\rceil \right)$$
3.3 Ungraph: Tail-Head Fragments Based

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

\textbf{Proof}

For each (unoriented) vertex there is a pointer to its neighbour list, hence $|\mathcal{R}_{\text{au}}|$ 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 ($\lceil \log_2|\mathcal{R}_{\text{au}}| \rceil$ 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.

\[ \square \]

\textbf{verify if same as comments}

The four combinations of edges’ attributes $\text{or}$ and $\text{rel}$ 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)

\textbf{description}

\textbf{ref figure from digraph implementation revsymg}

\textbf{memory}

3.3 Undirected Graph: Tail-Head Fragments Based

3.3.1 All Oriented Fragments Undirected Graph (UGA)

\textbf{fix for ungraph}

For each read, there are two vertices in the directed graph. Remind that each read identifier $r_{\text{rid}}$ 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).

\[
\begin{align*}
N^O_v & \quad N^E_v \\
\end{align*}
\]

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

\[
\begin{align*}
G & = (V, E^R \cup E^O) \\
& \quad \forall v_h \in V, v_{had} = 2 \times r_{rid} \\
& \quad \forall v_l \in V, v_{lad} = 2 \times r_{rid} + 1 \\
& \quad (v_l, v_h) \in E^R \iff v \in \mathcal{R} \\
& \quad (u, v) \in E^O \iff (u, v) \in \mathcal{O}
\end{align*}
\]

\[\begin{array}{l}
\text{Memory} \\
\end{array}\]

\[34\] V. Epain
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

4.1 Algorithms

4.1.1 Subfunctions

▶ Algorithm 1: Reverse operation on index

**Require:** Vertex/edge index \( \text{ind} \).

**Ensure:** Returns index of the reverse vertex/edge.

1: function \( \text{REV}(\text{ind}) \)
2: \[ \text{if } \text{ind} \mid 2 \text{ then} \]
3: \[ \text{return } \text{ind} + 1 \]
4: \[ \text{return } \text{ind} - 1 \]

For clarity’s sake, \( \text{REV}(\text{ind}) = \overline{\text{ind}} \).

▶ Algorithm 2: Find place of edge index in the neighbour list. Cost:

\[
\text{worst } 3 \times |\text{list}|, \text{ best } 3, \text{ average } 3 \times \left\lceil \frac{|\text{list}|}{2} \right\rceil
\]

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

**Ensure:** Returns the index of the tuple containing \( e_{\text{ind}} \) in \( \text{list} \).

1: function GET__INDEX(\( \text{list}, e_{\text{ind}}, t_{\text{ind}} \))
2: \[ \text{list}_{\text{ind}} \leftarrow 0 \]
4 PERFORMANCE

4.1 Algorithms

```
3: while list[tind][ind] ≠ e_ind and list_ind < |list| - 1 do
4:     list_ind ← list_ind + 1
5: return list_ind
```

4.1.2 Iterating Over the Predecessors

▶ Algorithm 3: Iterate over the predecessors for DGS

Require: Graph $G = (V, E)$, vertex $v ∈ V$, empty list $peds$.
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 $(u, e_{ind}) ∈ N^+_v$ do
3:     preds.append($u, e_{ind}$)
4: return preds

▶ Algorithm 4: Iterate over the predecessors for DGF

Require: Graph $G = (V, E)$, vertex $v ∈ V$, orientation of $v v_{or} ∈ \{0; 1\}$, empty list $peds$.
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
3:     for $(u, e_{ind}) ∈ N^-_v$ do
4:       preds.append($u, e_{ind}$)
5:   else
6:     for $(u, u_{or}, e_{ind}) ∈ N^+_v$ do
7:       preds.append($u, 1 - u_{or}, e_{ind}$)
8:   return preds

▶ Algorithm 5: Iterate over the predecessors for BGU

Require: Graph $G = (V, E)$, vertex $v ∈ V$, orientation of $v v_{or} ∈ \{0; 1\}$, empty list $peds$.
Ensure: The returned list contains the predecessors of $v$ in graph $G$ and the corresponding edge index.
4.1 Algorithms

4.1.3 Iterating Over the Successors

▶ Algorithm 6: Iterate over the successors for DGS

**Require:** Graph $G = (V, E)$, vertex $v \in V$, empty list $suc$s.
**Ensure:** The returned list contains the successors of $v$ in graph $G$.

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

▶ Algorithm 7: 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 $suc$s.
**Ensure:** The returned list contains the successors of $v$ in graph $G$.

```plaintext
1: function BGU_PREDs($G, v, v_{or}, preds$)
2: orientations ← (0, 1) \quad \triangleright \text{ 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 \quad \triangleright \ u_f \rightarrow v_f \text{ or } u_r \rightarrow v_f
7:         preds.APPEND($u$, orientations[0], $2 \times e'_{ind}$)
8:       else if $rel_{uv} = 0$ then \quad \triangleright \ u_f \rightarrow v_r \text{ or } u_r \rightarrow v_r
9:         preds.APPEND($u$, orientations[1], $2 \times e'_{ind} + 1$)
10:     else if $v_{or} = 0$ then
11:       if $or_{uv} = 0$ then
12:         if $rel_{uv} = 0$ then \quad \triangleright \ u_f \rightarrow v_f
13:           preds.APPEND($u$, $0 \times e'_{ind}$)
14:         else if $rel_{uv} = 1$ then \quad \triangleright \ u_r \rightarrow v_f
15:           preds.APPEND($u$, $1 \times e'_{ind} + 1$)
16:       else if $or_{uv} = 0$ then
17:         if $rel_{uv} = 1$ then \quad \triangleright \ u_r \rightarrow v_r
18:           preds.APPEND($u$, $1 \times e'_{ind} + 1$)
19:         else if $rel_{uv} = 0$ then \quad \triangleright \ u_f \rightarrow v_r
20:           preds.APPEND($u$, $0 \times e'_{ind}$)
21: return preds
```
1: \textbf{function} DGF\_SUCCS\((G, v, v_{or}, succs)\)  
2: \hspace{1em} \textbf{if} \(v_{or} = 0\) \textbf{then} \hspace{1em} \triangleright v \text{ forward}  
3: \hspace{2em} \textbf{for} \((w, e_{ind}) \in N^+_v\) \textbf{do}  
4: \hspace{3em} succs.\text{APPEND}(w, e_{ind})  
5: \hspace{1em} \textbf{else}  
6: \hspace{2em} \textbf{for} \((w, w_{or}, e_{ind}) \in N^-_v\) \textbf{do}  
7: \hspace{3em} succs.\text{APPEND}(w, 1 - w_{or}, e_{ind})  
8: \textbf{return} succs

\begin{algorithm}
\begin{algorithmic}
\State \textbf{function} DGF\_SUCCS\((G, v, v_{or}, succs)\)
\State \hspace{1em} \textbf{if} \(v_{or} = 0\) \textbf{then} \hspace{1em} \triangleright v \text{ forward}
\State \hspace{2em} \textbf{for} \((w, e_{ind}) \in N^+_v\) \textbf{do}
\State \hspace{3em} succs.\text{APPEND}(w, e_{ind})
\State \hspace{1em} \textbf{else}
\State \hspace{2em} \textbf{for} \((w, w_{or}, e_{ind}) \in N^-_v\) \textbf{do}
\State \hspace{3em} succs.\text{APPEND}(w, 1 - w_{or}, e_{ind})
\State \textbf{return} succs
\end{algorithmic}
\end{algorithm}

\begin{algorithm}
\begin{algorithmic}
\State \textbf{function} BGU\_SUCCS\((G, v, v_{or}, succs)\)
\State \hspace{1em} \textbf{orientations} \leftarrow (0, 1) \hspace{1em} \triangleright \text{forward/reverse orientations}
\State \hspace{1em} \textbf{for} \((w, e'_{ind}) \in N_v\) \textbf{do}
\State \hspace{2em} \textbf{if} \(v < w\) \textbf{then}
\State \hspace{3em} \textbf{if} \(v_{or} = 0\) \textbf{then}
\State \hspace{4em} \textbf{if} \(rel_{vw} = 0\) \textbf{then} \hspace{1em} \triangleright v \text{ forward or } v \text{ reverse}
\State \hspace{5em} succs.\text{APPEND}(w, \text{orientations}[\text{or}_{v_{or}}], 2 \times e'_{ind})
\State \hspace{4em} \textbf{else if} \(rel_{uv} = 1\) \textbf{then} \hspace{1em} \triangleright v \text{ reverse or } v \text{ forward}
\State \hspace{5em} succs.\text{APPEND}(w, \text{orientations}[1 - \text{or}_{u_{or}}], 2 \times e'_{ind} + 1)
\State \hspace{2em} \textbf{else if} \(v_{or} = 0\) \textbf{then}
\State \hspace{3em} \textbf{if} \(or_{uv} = 0\) \textbf{then}
\State \hspace{4em} \textbf{if} \(rel_{uv} = 1\) \textbf{then} \hspace{1em} \triangleright v \text{ forward}
\State \hspace{5em} succs.\text{APPEND}(w, 0, 2 \times e'_{ind})
\State \hspace{4em} \textbf{else if} \(rel_{uv} = 0\) \textbf{then} \hspace{1em} \triangleright v \text{ forward}
\State \hspace{5em} succs.\text{APPEND}(w, 1, 2 \times e'_{ind} + 1)
\State \hspace{2em} \textbf{else if} \(or_{uv} = 0\) \textbf{then}
\State \hspace{3em} \textbf{if} \(rel_{uv} = 0\) \textbf{then} \hspace{1em} \triangleright v \text{ reverse}
\State \hspace{4em} succs.\text{APPEND}(w, 1, 2 \times e'_{ind} + 1)
\State \hspace{4em} \textbf{else if} \(rel_{uv} = 1\) \textbf{then} \hspace{1em} \triangleright v \text{ reverse}
\State \hspace{5em} succs.\text{APPEND}(w, 0, 2 \times e'_{ind})
\State \hspace{2em} \textbf{else if} \(rel_{uv} = 0\) \textbf{then} \hspace{1em} \triangleright v \text{ reverse}
\State \hspace{4em} succs.\text{APPEND}(w, 1, 2 \times e'_{ind} + 1)
\State \hspace{4em} \textbf{else if} \(rel_{uv} = 1\) \textbf{then} \hspace{1em} \triangleright v \text{ forward}
\State \hspace{5em} succs.\text{APPEND}(w, 0, 2 \times e'_{ind})
\State \hspace{2em} \textbf{return} succs
\end{algorithmic}
\end{algorithm}

\textbf{Algorithm 8: Iterate over the successors for BGU}

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

\textbf{Ensure:} The returned list contains the successors of \(v\) in graph \(G\).
4.1 Algorithms

4.1.4 Adding a Vertex

▶ Algorithm 9: 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 10: 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 11: 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 $|N_V^-| - 1$
4.1.5 Adding an Edge

► Algorithm 12: 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: \[ \triangleright \text{ind\_edges is the number of edges. It is always even.} \]
3: \[ N_u^+.\text{APPEND}(v, \text{ind\_edges}) \]
4: \[ N_v^+.\text{APPEND}(u, \text{ind\_edges} + 1) \]
5: \[ \text{ind\_edges} \leftarrow \text{ind\_edges} + 2 \]
6: \[ \text{card\_edges} \leftarrow \text{card\_edges} + 2 \]
7: \[ \text{return} \text{ind\_edges} - 2 \]

► Algorithm 13: 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: \[ \triangleright \text{ind\_edges is the number of edges. It is always even.} \]
3: \[ \text{if} u_{or} = 0 \text{ then} \]
4: \[ N_u^+.\text{APPEND}(v, v_{or}, \text{ind\_edges}) \]
5: \[ \text{else} \]
6: \[ N_u^-.\text{APPEND}(v, 1 - v_{or}, \text{ind\_edges} + 1) \]
7: \[ \text{if} v_{or} = 0 \text{ then} \]
8: \[ N_u^-.\text{APPEND}(u, u_{or}, \text{ind\_edges}) \]
9: \[ \text{else} \]
10: \[ N_u^+.\text{APPEND}(u, 1 - u_{or}, \text{ind\_edges} + 1) \]
11: \[ \text{ind\_edges} \leftarrow \text{ind\_edges} + 2 \]
12: \[ \text{card\_edges} \leftarrow \text{card\_edges} + 2 \]
13: \[ \text{return} \text{ind\_edges} - 2 \]
4.1 Algorithms 4 PERFORMANCE

Algorithm 14: 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: $\triangleright$ `ind_edges` is the number of edges. `E_attr` is a list of edges attributes. $	riangleright$
3: or ← $|u_{or} - v_{or}|$
4: if $u < v$ then
5: rel ← $u_{or}$
6: else
7: rel ← $1 - v_{or}$
8: $N_u$.APPEND($v$, `ind_edges`)
9: $N_v$.APPEND($u$, `ind_edges`)
10: `E_attr`.APPEND(`or`, `rel`)
11: `ind_edges` ← `ind_edges` + 1
12: `card_edges` ← `card_edges` + 1
13: return $2 \times (\text{ind_edges} - 1)$

4.1.6 Deleting a Vertex

Algorithm 15: 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 $V' = V \setminus \{v, \overline{v}\}$, and $\forall v \in V', 0 \leq v_{ind} \leq |V'| - 2$.

1: function `dgs_delete_vertex(G, v)`
2: $\triangleright$ Delete it from its predecessors $	riangleright$
3: $v_{rev}$ ← $v + 1$
4: for $(u, e_{ind}) \in N^+_u_{rev}$ do
5: $u_{rev}$ ← $\overline{u}$
6: `adj_ind` ← GET_INDEX($N^+_u_{rev}$, $e_{ind}$, 1)
7: if `adj_ind` = $|N^+_u_{rev}| - 1$ then
8: $\triangleright$ Just delete the last element $	riangleright$
9: $N^+_u_{rev}$.POP()$	riangleright$
10: else
11: $\triangleright$ Replace the edge by the last in the neighbour list

V. Epain 41
\[N^+_u.\text{rev}[\text{adj\_ind}] \leftarrow N^+_u.\text{POP}( )\]
\[\text{card\_edges} \leftarrow \text{card\_edges} - 2\]
\[\text{DELETE}(N^+_u)\]
\[\triangleright \text{Delete it from its successors}\]
\[\text{for } (w, e\text{\_ind}) \in N^+_v \text{ do}\]
\[w.\text{rev} \leftarrow \bar{w}\]
\[\text{adj\_ind} \leftarrow \text{GET\_INDEX}(N^+_u.\text{rev}, \bar{e}\text{\_ind}, 1)\]
\[\text{if } \text{adj\_ind} = |N^+_u.\text{rev}| - 1 \text{ then}\]
\[\triangleright \text{Just delete the last element}\]
\[N^+_u.\text{rev}.\text{POP}( )\]
\[\text{else}\]
\[\triangleright \text{Replace the edge by the last in the neighbour list}\]
\[N^+_u.\text{rev}[\text{adj\_ind}] \leftarrow N^+_u.\text{rev}.\text{POP}( )\]
\[\text{card\_edges} \leftarrow \text{card\_edges} - 2\]
\[\text{DELETE}(N^+_u)\]
\[\triangleright \text{Delete the whole vertex}\]
\[\text{if } v = |N^+_v| - 2 \text{ then}\]
\[\triangleright \text{It is the last, just pop it and its reverse}\]
\[N^+_v.\text{POP}( )\]
\[N^+_v.\text{POP}( )\]
\[\text{else}\]
\[\triangleright \text{Replace it and its reverse by the last and its reverse}\]
\[N^+_u.\text{rev} \leftarrow N^+_v.\text{POP}( )\]
\[N^+_u \leftarrow N^+_v.\text{POP}( )\]
\[\triangleright \text{Update the vertex index}\]
\[\text{for } (w, e\text{\_ind}) \in N^+_u \text{ do}\]
\[w.\text{rev} \leftarrow \bar{w}\]
\[e\text{\_ind} \leftarrow \bar{e}\text{\_ind}\]
\[\text{adj\_ind} \leftarrow \text{GET\_INDEX}(N^+_u.\text{rev}, \bar{e}\text{\_ind} \cdot \text{rev}, 1)\]
\[N^+_u.\text{rev}[\text{adj\_ind}] \leftarrow (v.\text{rev}, e\text{\_ind} \cdot \text{rev})\]
\[\text{for } (w, e\text{\_ind}) \in N^+_u.\text{rev} \text{ do}\]
\[w.\text{rev} \leftarrow \bar{w}\]
\[e\text{\_ind} \leftarrow \bar{e}\text{\_ind}\]
\[\text{adj\_ind} \leftarrow \text{GET\_INDEX}(N^+_u.\text{rev}, e\text{\_ind} \cdot \text{rev}, 1)\]
\[N^+_u.\text{rev}[\text{adj\_ind}] \leftarrow (v, e\text{\_ind} \cdot \text{rev})\]
**Algorithm 16: 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 $V' = V \setminus \{v, \overline{v}\}$, and $\forall v \in V', 0 \leq v_{\text{ind}} \leq |V'| - 2$.

```plaintext
1: function DGF_DELETE_VERTEX($G$, $v$)
2:     ▶ Delete it from its predecessors
3:     for $(u, u_{\text{or}}, e_{\text{ind}}) \in N_+^-$ do
4:         if $u_{\text{or}} = 0$ then
5:             $adj_{\text{ind}} \leftarrow \text{GET\_INDEX}(N_+^+, e_{\text{ind}}, 1)$
6:             if $adj_{\text{ind}} = |N_+^+| - 1$ then
7:                 ▶ Just delete the last element
8:                     $N_+^+$.POP( )
9:             else
10:                 ▶ Replace the edge by the last in the neighbour list
11:                     $N_+^+[adj_{\text{ind}}] \leftarrow N_+^+.POP( )$
12:         else
13:             $adj_{\text{ind}} \leftarrow \text{GET\_INDEX}(N_-^+, e_{\text{ind}}, 1)$
14:             if $adj_{\text{ind}} = |N_-^+| - 1$ then
15:                 ▶ Just delete the last element
16:                     $N_-^+$.POP( )
17:             else
18:                 ▶ Replace the edge by the last in the neighbour list
19:                     $N_-^-[adj_{\text{ind}}] \leftarrow N_-^+.POP( )$
20:             end for
21:         end if
22:     end for
23:     ▶ Delete it from its successors
24:     for $(w, w_{\text{or}}, e_{\text{ind}}) \in N_+^+$ do
25:         if $w_{\text{or}} = 0$ then
26:             $adj_{\text{ind}} \leftarrow \text{GET\_INDEX}(N_-^-, e_{\text{ind}}, 1)$
27:             if $adj_{\text{ind}} = |N_-^-| - 1$ then
28:                 ▶ Just delete the last element
29:                     $N_-^-.[adj_{\text{ind}}] \leftarrow N_-^-..POP( )$
30:             else
31:                 ▶ Replace the edge by the last in the neighbour list
32:                     $N_-^-[adj_{\text{ind}}] \leftarrow N_-^-.POP( )$
33:             end if
34:         else
35:             $adj_{\text{ind}} \leftarrow \text{GET\_INDEX}(N_-^+, e_{\text{ind}}, 1)$
36:             if $adj_{\text{ind}} = |N_-^+| - 1$ then
37:                 ▶ Just delete the last element
38:                     $N_-^+.[adj_{\text{ind}}] \leftarrow N_-^+.POP( )$
39:             end if
40:         end if
41:     end for
42:     $\text{card\_edges} \leftarrow \text{card\_edges} - 2$
43:     $\text{DELETE}(N_-^-)$
44:     ▶ Delete it from its successors
45:     for $(w, w_{\text{or}}, e_{\text{ind}}) \in N_-^+$ do
46:         if $w_{\text{or}} = 0$ then
47:             $adj_{\text{ind}} \leftarrow \text{GET\_INDEX}(N_+^-, e_{\text{ind}}, 1)$
48:             if $adj_{\text{ind}} = |N_+^-| - 1$ then
49:                 ▶ Just delete the last element
50:                     $N_+^-.[adj_{\text{ind}}] \leftarrow N_+^-..POP( )$
51:             else
52:                 ▶ Replace the edge by the last in the neighbour list
53:                     $N_+^-[adj_{\text{ind}}] \leftarrow N_+^--.POP( )$
54:             end if
55:         else
56:             $adj_{\text{ind}} \leftarrow \text{GET\_INDEX}(N_+^+, e_{\text{ind}}, 1)$
57:             if $adj_{\text{ind}} = |N_+^+| - 1$ then
58:                 ▶ Just delete the last element
59:                     $N_+^+.[adj_{\text{ind}}] \leftarrow N_+^+.POP( )$
60:             end if
61:         end if
62:     end for
```

**Algorithm 16: Delete a vertex for DGF**

1. **Require:** Graph $G = (V, E)$, vertex $v \in V$ (in forward orientation).
2. **Ensure:** Delete the vertex and its reverse such that $V' = V \setminus \{v, \overline{v}\}$, and $\forall v \in V', 0 \leq v_{\text{ind}} \leq |V'| - 2$.
3. **Function** DGF_DELETE_VERTEX($G$, $v$)
   4. ▶ Delete it from its predecessors
5.  for $(u, u_{\text{or}}, e_{\text{ind}}) \in N_+^-$ do
6.      if $u_{\text{or}} = 0$ then
7.          $adj_{\text{ind}} \leftarrow \text{GET\_INDEX}(N_+^+, e_{\text{ind}}, 1)$
8.          if $adj_{\text{ind}} = |N_+^+| - 1$ then
9.              ▶ Just delete the last element
10.             $N_+^+$.POP( )
11.          else
12.              ▶ Replace the edge by the last in the neighbour list
13.             $N_+^+[adj_{\text{ind}}] \leftarrow N_+^+.POP( )$
14.      else
15.          $adj_{\text{ind}} \leftarrow \text{GET\_INDEX}(N_-^+, e_{\text{ind}}, 1)$
16.          if $adj_{\text{ind}} = |N_-^+| - 1$ then
17.              ▶ Just delete the last element
18.             $N_-^+$.POP( )
19.          else
20.              ▶ Replace the edge by the last in the neighbour list
21.             $N_-^-[adj_{\text{ind}}] \leftarrow N_-^+.POP( )$
22.      end if
23.  end for
24. ▶ Delete it from its successors
25.  for $(w, w_{\text{or}}, e_{\text{ind}}) \in N_+^+$ do
26.      if $w_{\text{or}} = 0$ then
27.          $adj_{\text{ind}} \leftarrow \text{GET\_INDEX}(N_-^-, e_{\text{ind}}, 1)$
28.          if $adj_{\text{ind}} = |N_-^-| - 1$ then
29.              ▶ Just delete the last element
30.             $N_-^-.[adj_{\text{ind}}] \leftarrow N_-^-..POP( )$
31.          else
32.              ▶ Replace the edge by the last in the neighbour list
33.             $N_-^-[adj_{\text{ind}}] \leftarrow N_-^--.POP( )$
34.      else
35.          $adj_{\text{ind}} \leftarrow \text{GET\_INDEX}(N_+^+, e_{\text{ind}}, 1)$
36.          if $adj_{\text{ind}} = |N_+^+| - 1$ then
37.              ▶ Just delete the last element
38.             $N_+^+.[adj_{\text{ind}}] \leftarrow N_+^+.POP( )$
39.          end if
40.      end if
41.  end for
42. $\text{card\_edges} \leftarrow \text{card\_edges} - 2$
43. $\text{DELETE}(N_-^-)$
44. ▶ Delete it from its successors
45.  for $(w, w_{\text{or}}, e_{\text{ind}}) \in N_-^+$ do
46.      if $w_{\text{or}} = 0$ then
47.          $adj_{\text{ind}} \leftarrow \text{GET\_INDEX}(N_+^-, e_{\text{ind}}, 1)$
48.          if $adj_{\text{ind}} = |N_+^-| - 1$ then
49.              ▶ Just delete the last element
50.             $N_+^-.[adj_{\text{ind}}] \leftarrow N_+^-..POP( )$
51.          else
52.              ▶ Replace the edge by the last in the neighbour list
53.             $N_+^-[adj_{\text{ind}}] \leftarrow N_+^--.POP( )$
54.      else
55.          $adj_{\text{ind}} \leftarrow \text{GET\_INDEX}(N_-^+, e_{\text{ind}}, 1)$
56.          if $adj_{\text{ind}} = |N_-^+| - 1$ then
57.              ▶ Just delete the last element
58.             $N_-^+.[adj_{\text{ind}}] \leftarrow N_-^+.POP( )$
59.          end if
60.      end if
61.  end for
4 PERFORMANCE

4.1 Algorithms

```plaintext
36:     \texttt{N}_u^+ \texttt{.POP( )}
37:     \textbf{else}
38:     \quad \triangleright \text{Replace the edge by the last in the neighbour list}
39:     \quad \texttt{N}_u^+ [\texttt{adj\_ind}] \leftarrow \texttt{N}_u^+ \texttt{.POP( )}
40:     \quad \texttt{card\_edges} \leftarrow \texttt{card\_edges} - 2
41:     \quad \texttt{DELETE( N}_u^+ \texttt{)}
42:     \quad \triangleright \text{Delete the whole vertex}
43:     \quad \textbf{if} \ v = \left| N}_V^+ \right| - 1 \textbf{ then}
44:     \quad \quad \triangleright \text{It is the last, just pop it and its reverse}
45:     \quad \quad \texttt{N}_V^- \texttt{.POP( )}
46:     \quad \quad \texttt{N}_V^+ \texttt{.POP( )}
47:     \quad \textbf{else}
48:     \quad \quad \triangleright \text{Replace it and its reverse by the last and its reverse}
49:     \quad \quad \texttt{N}_u^- \leftarrow \texttt{N}_V^- \texttt{.POP( )}
50:     \quad \quad \texttt{N}_u^+ \leftarrow \texttt{N}_V^+ \texttt{.POP( )}
51:     \quad \quad \triangleright \text{Update the vertex index}
52:     \quad \quad \texttt{for} (u, u_{\text{or}}, e_{\text{ind}}) \in \texttt{N}_u^- \texttt{ do}
53:     \quad \quad \quad \textbf{if} \ u_{\text{or}} = 0 \textbf{ then}
54:     \quad \quad \quad \quad \texttt{adj\_ind} \leftarrow \texttt{GET\_INDEX( N}_u^+, e_{\text{ind}}, 1)
55:     \quad \quad \quad \quad \texttt{N}_u^+ [\texttt{adj\_ind}] \leftarrow (v, 0, e_{\text{ind}})
56:     \quad \quad \quad \textbf{else}
57:     \quad \quad \quad \quad u_{\text{rev}} \leftarrow \overline{u}
58:     \quad \quad \quad \quad e_{\text{ind\_rev}} \leftarrow \overline{e_{\text{ind}}}
59:     \quad \quad \quad \quad \texttt{adj\_ind} \leftarrow \texttt{GET\_INDEX( N}_{\text{u\_rev}}^- , e_{\text{ind\_rev}}, 1)
60:     \quad \quad \quad \quad \texttt{N}_{\text{u\_rev}}^- [\texttt{adj\_ind}] \leftarrow (v, 1, e_{\text{ind\_rev}})
61:     \quad \quad \texttt{for} (w, w_{\text{or}}, e_{\text{ind}}) \in \texttt{N}_u^+ \texttt{ do}
62:     \quad \quad \quad \textbf{if} \ w_{\text{or}} = 0 \textbf{ then}
63:     \quad \quad \quad \quad \texttt{adj\_ind} \leftarrow \texttt{GET\_INDEX( N}_w^- , e_{\text{ind}}, 1)
64:     \quad \quad \quad \quad \texttt{N}_w^- [\texttt{adj\_ind}] \leftarrow (v, 0, e_{\text{ind}})
65:     \quad \quad \quad \textbf{else}
66:     \quad \quad \quad \quad w_{\text{rev}} \leftarrow \overline{w}
67:     \quad \quad \quad \quad e_{\text{ind\_rev}} \leftarrow \overline{e_{\text{ind}}}
68:     \quad \quad \quad \quad \texttt{adj\_ind} \leftarrow \texttt{GET\_INDEX( N}_{w\_rev}^+, e_{\text{ind\_rev}}, 1)
69:     \quad \quad \quad \quad \texttt{N}_{w\_rev}^+ [\texttt{adj\_ind}] \leftarrow (v, 1, e_{\text{ind\_rev}})
```

▶ Algorithm 17: Delete a vertex for BGU

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

V. Epain
4.1 Algorithms

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

1: function $\text{BGU\_DELETE\_VERTEX}(G, v)$
2: for $(w, e_{\text{ind}}) \in N_u$ do
3:   $\text{adj\_ind} \leftarrow \text{GET\_INDEX}(N_w, e_{\text{ind}}, 1)$
4:   if $\text{adj\_ind} = |N_w| - 1$ then
5:       $\triangleright$ Just delete the last element
6:       $N_u..\text{POP}( )$
7:   else
8:       $\triangleright$ Replace the edge by the last in the neighbour list
9:       $N_u[\text{adj\_ind}] \leftarrow N_u..\text{POP}( )$
10: end
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 $(w, e_{\text{ind}}) \in N_v$ do
20:   $\text{adj\_ind} \leftarrow \text{GET\_INDEX}(N_w, e_{\text{ind}}, 1)$
21:   $N_w[\text{adj\_ind}] \leftarrow (v, e_{\text{ind}})$
22:   if $w > v$ then
23:       $\triangleright$ The last (the greatest) identifier becomes a fewer and
       breaks the identifier order
24:       $E\_\text{attr}[e_{\text{ind}}] \leftarrow (E\_\text{attr}[e_{\text{ind}}][0], 1 - E\_\text{attr}[e_{\text{ind}}][1])$

4.1.7 Deleting an Edge

▶ Algorithm 18: Delete an edge for DGS

Require: Graph $G = (V, E)$, edge $(u, v) \in E$ and the edge’s index $e_{\text{ind}}$.
Ensure: $|N_u^+ \setminus \text{edge}| = |N_u^+| - 1$ and $|N_v^- \setminus \text{edge}| = |N_v^-| - 1$.

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

V. Epain 45
5: \( \triangleright \) Just delete the last element
6: \( N^+_u.\text{POP}( ) \)
7: \( \text{else} \)
8: \( \triangleright \) Replace the edge by the last in the neighbour list
9: \( N^+_u[\text{adj\_ind}] \leftarrow N^+_u.\text{POP}( ) \)
10: \( \triangleright \) Remove \( u \) from \( v \)’s predecessors.
11: \( v\_\text{rev} \leftarrow \overline{v} \)
12: \( \text{adj\_ind} \leftarrow \text{GET\_INDEX}(N^+_v, e\_\text{ind}, 1) \)
13: \( \text{if} \ \text{adj\_ind} = |N^+_v| - 1 \ \text{then} \)
14: \( \triangleright \) Just delete the last element
15: \( N^+_v.\text{POP}( ) \)
16: \( \text{else} \)
17: \( \triangleright \) Replace the edge by the last in the neighbour list
18: \( N^+_v[\text{adj\_ind}] \leftarrow N^+_v.\text{POP}( ) \)
19: \( \text{card\_edges} \leftarrow \text{card\_edges} - 2 \)

\[\text{Function DGF\_DELETE\_EDGE}(G, u, u\_or, v, v\_or, e\_\text{ind})\]

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 \( e\_\text{ind} \).

Ensure: \( |N^+_u \setminus \text{edge}| = |N^+_u| - 1 \) and \( |N^-_v \setminus \text{edge}| = |N^-_v| - 1 \).

1: \( \triangleright \) Remove \( v \) from \( u \)’s successors
2: \( \text{if} \ u\_\text{or} = 0 \ \text{then} \)
3: \( \text{list\_sucs} \leftarrow N^+_u \)
4: \( \text{adj\_ind} \leftarrow \text{GET\_INDEX}(\text{list\_sucs}, e\_\text{ind}, 2) \)
5: \( \text{else} \)
6: \( \text{list\_sucs} \leftarrow N^-_u \)
7: \( \text{adj\_ind} \leftarrow \text{GET\_INDEX}(\text{list\_sucs}, e\_\text{ind}, 2) \)
8: \( \text{if} \ \text{adj\_ind} = |\text{list\_sucs}| - 1 \ \text{then} \)
9: \( \triangleright \) Just delete the last element
10: \( \text{list\_sucs.\text{POP}( )} \)
11: \( \text{else} \)
12: \( \triangleright \) Replace the edge by the last in the neighbour list
13: \( \text{list\_sucs[adj\_ind]} \leftarrow \text{list\_sucs.\text{POP}( )} \)
14: \( \triangleright \) Remove \( u \) from \( v \)’s predecessors.
15: \( \text{if} \ v\_\text{or} = 0 \ \text{then} \)
16: \( \text{list\_preds} \leftarrow N^-_v \)
17: \( \text{adj\_ind} \leftarrow \text{GET\_INDEX}(\text{list\_preds}, e\_\text{ind}, 2) \)
4.1 Algorithms 4 PERFORMANCE

![Algorithm 20: Delete an edge for BGU](image)

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

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

4.2.1 Complexities Calculus Details

Table 1 details the calculus of algorithmic costs.

**Table 1** – Calculus detail of basic operations costs. $c(x)$ is the cost of element $x$.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Cost</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>0</td>
<td>Memory access</td>
</tr>
<tr>
<td>$x \pm y$</td>
<td>$1 + c(x) + c(y)$</td>
<td>Basic operation</td>
</tr>
<tr>
<td>$x \leftarrow y$</td>
<td>$c(y)$</td>
<td>Affectation</td>
</tr>
<tr>
<td>$</td>
<td>x</td>
<td>$</td>
</tr>
<tr>
<td>if $x \leq y$</td>
<td>$1 + c(x) + c(y)$</td>
<td>Conditional</td>
</tr>
<tr>
<td>EMPTY_LIST()</td>
<td>1</td>
<td>Empty list constructor</td>
</tr>
<tr>
<td>list.APPEND($x$)</td>
<td>$1 + c(x)$</td>
<td>Add $x$ to the end of the list $list$</td>
</tr>
<tr>
<td>list.POP()</td>
<td>1</td>
<td>Delete the last element of the list $list$</td>
</tr>
<tr>
<td>DELETE(list)</td>
<td>$</td>
<td>list</td>
</tr>
<tr>
<td>for $i \in [a; b]$ do $x$</td>
<td>$(b - a + 1) \times c(x)$</td>
<td>For loop</td>
</tr>
</tbody>
</table>

**best, worst and average cases are respectively rather lower and upper bounds, and equiprobable independent event’s costs**

4.2.2 Costs for Subfunctions

Table 2 gives the algorithmic costs of subfunctions in Algorithms 1 and 2.

**Table 2** – Algorithmic costs for subfunctions.

<table>
<thead>
<tr>
<th></th>
<th>Best</th>
<th>Worst</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>REV</td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>GET_INDEX</td>
<td>3</td>
<td>$3 \times</td>
<td>list$</td>
</tr>
</tbody>
</table>

4.2.3 Iterating Over the Neighbours

Table 3 gives the algorithmic costs of Algorithms 3 to 8.
Table 3 – Algorithmic costs of iterating over the neighbours for DGS, DGF and BGU.

(a) Iterate over the predecessors

<table>
<thead>
<tr>
<th></th>
<th>Best</th>
<th>Worst</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>DGS</td>
<td>$5 \times o_v - 2$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DGF</td>
<td>$o_v - 1$</td>
<td>$4 \times o_v + 3$</td>
<td>$2.5 \times o_v + 2$</td>
</tr>
<tr>
<td>BGU</td>
<td>$3 \times o_v$</td>
<td>$6 \times o_v$</td>
<td>$4.75 \times o_v$</td>
</tr>
</tbody>
</table>

(b) Iterate over the successors

<table>
<thead>
<tr>
<th></th>
<th>Best</th>
</tr>
</thead>
<tbody>
<tr>
<td>DGS</td>
<td>$o_v^+$</td>
</tr>
<tr>
<td>DGF</td>
<td>$o_v^+ - 1$</td>
</tr>
<tr>
<td>BGU</td>
<td>$3 \times o_v$</td>
</tr>
</tbody>
</table>

4.2.4 Costs for Dynamics

Table 4 gives the algorithmic costs of Algorithms 9 to 14. Table 5 gives the algorithmic costs of Algorithms 18 to 20. Table 6 gives the algorithmic costs of Algorithms 15 to 17.

Table 4 – Algorithmic costs of adding a vertex or an edge for DGS, DGF and BGU.

<table>
<thead>
<tr>
<th></th>
<th>Add a vertex</th>
<th>Add an edge</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Best</td>
<td>Worst</td>
</tr>
<tr>
<td>DGS</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>DGF</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>BGU</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>
Table 5 – Algorithmic costs of deleting an edge for DGS, DGF and BGU.

(a) Best and Worst cases

<table>
<thead>
<tr>
<th></th>
<th>Best</th>
<th>Worst</th>
</tr>
</thead>
<tbody>
<tr>
<td>DGS</td>
<td>17</td>
<td>$3 \times (o_u^+ + o_v^-) + 13$</td>
</tr>
<tr>
<td>DGF</td>
<td>17</td>
<td>$3 \times (o_u^- + o_v^+) + 17$</td>
</tr>
<tr>
<td>BGU</td>
<td>14</td>
<td>$3 \times (o_u + o_v) + 10$</td>
</tr>
</tbody>
</table>

(b) Average case

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DGS</td>
<td>$3 \times \left(\left\lceil \frac{o_u^+}{2} \right\rceil + \left\lceil \frac{o_v^-}{2} \right\rceil \right) + 13 - \frac{1}{o_u^+} - \frac{1}{o_v^-}$</td>
</tr>
<tr>
<td>DGF</td>
<td>$\frac{3}{2} \times \left(\left\lceil \frac{o_u^-}{2} \right\rceil + \left\lceil \frac{o_v^+}{2} \right\rceil + \left\lceil \frac{o_u^-}{2} \right\rceil + \left\lceil \frac{o_v^+}{2} \right\rceil \right) + 13 - 2 \times \left(\frac{1}{o_u} + \frac{1}{o_v}\right)$</td>
</tr>
<tr>
<td>BGU</td>
<td>$3 \times \left(\left\lceil \frac{o_u}{2} \right\rceil + \left\lceil \frac{o_v}{2} \right\rceil \right) + 11 - \frac{1}{o_u} - \frac{1}{o_v}$</td>
</tr>
</tbody>
</table>
4.2 Time Complexities

**Table 6** – Algorithmic costs of deleting a vertex for DGS, DGF and BGU.

(a) **Best cases**

<table>
<thead>
<tr>
<th></th>
<th>DGS</th>
<th>DGF</th>
<th>BGU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

(b) **Worst cases**

- **DGS**
  
  \[9 \times o_v + 3 \times \left( \sum_{u \in N^-} o_u^+ + \sum_{w \in N^+_w} o_w^- + \sum_{x \in N^-_x} o_x^+ + \sum_{y \in N^+_y} o_y^- \right) + 5 \times (o_z + 1)\]

- **DGF**
  
  \[9 \times o_v + 4 \times o_z + 3 \times \left( \sum_{u \in N^-} o_u^+ + \sum_{w \in N^+_w} o_w^- + \sum_{x \in N^-_x} o_x^+ + \sum_{y \in N^+_y} o_y^- \right) + 4\]

- **BGU**
  
  \[5 \times o_v + 3 \times o_z + 3 \times \left( \sum_{w \in N_w} o_w + \sum_{y \in N_y} o_y \right) + 4\]

(c) **Average cases (DGS, DGF and BGU)**

\[
10 \times o_v + \frac{10 \times R_{aw} - 5}{2 \times R_{aw}} \times o_z + 3 \times \left( \sum_{u \in N^-} \left[ \frac{o_u^+}{2} \right] + \sum_{w \in N^+_w} \left[ \frac{o_w^-}{2} \right] \right) \\
+ \frac{6 \times R_{aw} - 3}{2 \times R_{aw}} \times \left( \sum_{x \in N^-_x} \left[ \frac{o_x^+}{2} \right] + \sum_{y \in N^+_y} \left[ \frac{o_y^-}{2} \right] \right) - \left( \sum_{u \in N^-} \frac{1}{o_u^+} + \sum_{w \in N^+_w} \frac{1}{o_w^-} \right) + 5 \\
+ \frac{15}{2} \times o_v + \frac{6 \times R_{aw} - 3}{2 \times R_{aw}} \times o_z + \frac{3}{2} \times \sum_{w \in N_w \cup N^+_w} \left( \left[ \frac{o_w^-}{2} \right] + \left[ \frac{o_w^+}{2} \right] \right) \\
+ \frac{6 \times R_{aw} - 3}{4 \times R_{aw}} \times \sum_{x \in N^-_x \cup N^+_y} \left( \left[ \frac{o_x^+}{2} \right] + \left[ \frac{o_y^-}{2} \right] \right) \\
- \frac{1}{2} \times \left( \sum_{w \in N_w} \frac{1}{o_w^+} + \sum_{w \in N^+_w} \frac{1}{o_w^-} \right) + \frac{6 \times R_{aw} - 1}{R_{aw}} \]

\[
5 \times o_v + \frac{3 \times R_{aw} - 3}{R_{aw}} \times o_z + 3 \times \sum_{w \in N_w} \left[ \frac{o_w}{2} \right] + \frac{3 \times R_{aw} + 3}{2 \times R_{aw}} \times \sum_{y \in N_y} \left[ \frac{o_y}{2} \right] - \sum_{w \in N_w} \frac{1}{o_w} + \frac{4 \times R_{aw} - 1}{R_{aw}} \]

V. Epain 51
References


### 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, 41–47  
- **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, 37–39, 41, 45, 47, 51, see edge

\( N^\neg \) Predecessors set 23, 28, 36, 38–40, 43–47, 51, see edge, \( N \) & \( N^+ \)

\( N^+ \) Successors set 23, 25, 28, 36–47, 51, see edge, \( N \) & \( N^\neg \)

**Operations**

\( \neg \) 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

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

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

\( \mathcal{R}_{\text{rev}} \) 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

genoome  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
vertex  Component of a graph 9, 11–23, 25–28, 30, 31, 33, 34, see graph & edge

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