Linear-Time Superbubble Identification Algorithm for Genome Assembly

Ljiljana Brankovic\textsuperscript{a,b}, Costas S. Iliopoulos\textsuperscript{b}, Ritu Kundu\textsuperscript{b}, Manal Mohamed\textsuperscript{b}, Solon P. Pissis\textsuperscript{b,*}, Fatima Vayani\textsuperscript{b}

\textsuperscript{a}School of Electrical Engineering and Computer Science, The University of Newcastle, Newcastle NSW 2308, Australia.
\textsuperscript{b}Department of Informatics, King’s College London, London WC2R 2LS, United Kingdom

Abstract

DNA sequencing is the process of determining the exact order of the nucleotide bases of an individual’s genome in order to catalogue sequence variation and understand its biological implications. Whole-genome sequencing techniques produce masses of data in the form of short sequences known as reads. Assembling these reads into a whole genome constitutes a major algorithmic challenge. Most assembly algorithms utilize de Bruijn graphs constructed from reads for this purpose. A critical step of these algorithms is to detect typical motif structures in the graph caused by sequencing errors and genome repeats, and filter them out; one such complex subgraph class is a so-called superbubble. In this paper, we propose an $O(n + m)$-time algorithm to detect all superbubbles in a directed acyclic graph with $n$ nodes and $m$ (directed) edges, improving the best-known $O(m \log m)$-time algorithm by Sung et al.

Keywords: genome assembly, de Bruijn graphs, superbubble

\*Corresponding author

Email addresses: ljiljana.brankovic@newcastle.edu.au (Ljiljana Brankovic), costas.iliopoulos@kcl.ac.uk (Costas S. Iliopoulos), ritu.kundu@kcl.ac.uk (Ritu Kundu), manal.mohamed@kcl.ac.uk (Manal Mohamed), solon.pissis@kcl.ac.uk (Solon P. Pissis), fatima.vayani@kcl.ac.uk (Fatima Vayani)
1. Introduction

Since the publication of the first draft of the human genome [1, 2], the field of genomics has changed dramatically. Recent developments in sequencing technologies (see [3], for example) have made it possible to sequence new genomes at a fraction of the time and cost required only a few years ago. With applications including sequencing the genome of a new species, an individual within a population, RNA molecules from a particular sample, and using DNA sequencing as a readout platform in molecular biology techniques such as nuclear run-on assays, sequencing remains at the core of genomics.

Whole-genome sequencing creates masses of data, in the order of tens of gigabytes, in the form of short sequences (reads); genome assembly involves piecing together these reads to form a set of contiguous sequences (contigs) representing the DNA sequence in the sample. Traditional assembly algorithms rely on the overlap-layout-consensus approach [4], representing each read as a node in an overlap graph and each detected overlap as a directed edge between the nodes corresponding to overlapping reads. These methods have proved their use through numerous \textit{de novo} genome assemblies [5].

Subsequently, a fundamentally different approach based on de Bruijn graphs was proposed [6], where representation of data elements was organised around words of \(k\) nucleotides, or \(k\)-mers, instead of reads. Unlike in an overlap graph, in a \textit{de Bruijn graph} [7], each \(k-1\) nucleotide long prefix and suffix of the \(k\)-mers is represented as a node and each \(k\)-mer is represented as a directed edge between its prefix and suffix nodes. The marginal information contained by a \(k\)-mer is its last nucleotide. The sequence of those final nucleotides is called the sequence of the node. In a de Bruijn graph, the assembly problem is reduced to finding an Eulerian path, that is, a trail that visits each edge in the graph exactly once.

However, sequencing errors and genome repeats significantly complicate the de Bruijn graph by adding false vertices and edges to it. Efficient and robust filtering methods were proposed to simplify the graph by filtering out motifs such as tips, bubbles, and cross links, which proved to be caused by sequencing errors [8]. Specifically, a \textit{bubble} consists of multiple directed unipaths (where a unipath is a path in which all internal vertices are of degree 2) from a vertex \(v\) to a vertex \(u\) and is commonly caused by a small number of errors in the centre of reads. Although these types of motifs are simple and can easily be identified and filtered out, more complicated motifs prove to be more challenging.
Recently, a complex generalization of a bubble, the so-called superbubble, was proposed as an important subgraph class for analyzing assembly graphs [9]. A *superbubble* is defined as a minimal subgraph $H$ in the de Bruijn graph with exactly one start node $s$ and one end node $t$ such that: (1) $H$ is a directed, acyclic, single source ($s$), single sink ($t$) graph (2) there is no edge from a node not in $H$ going to a node in $H\backslash\{s\}$ and (3) there is no edge from a node in $H\backslash\{t\}$ going to a node not in $H$. It is clear that many superbubbles are formed as a result of sequencing errors, inexact repeats, diploid/polyploid genomes, or frequent mutations. Thus, efficient detection of superbubbles is essential for the application of genome assembly [9].

Onodera et al. gave an $O(nm)$-time algorithm to detect superbubbles, where $n$ is the number of nodes and $m$ is the number of edges in the graph [9]. Very recently, Sung et al. gave an improved $O(m \log m)$-time algorithm to solve this problem [10]. The algorithm partitions the given graph into a set of subgraphs such that the set of superbubbles in all the subgraphs is the same as the set of superbubbles in the given graph. This set consists of subgraphs corresponding to each non-singleton strongly connected component and a subgraph corresponding to the set of all the vertices involved in singleton strongly connected components. Superbubbles are then detected in each subgraph; if it is cyclic, it is first converted into a directed acyclic subgraph by means of depth-first search and by duplicating some nodes.

Note that the cost of partitioning the graph and transforming it into the directed acyclic subgraphs is linear with respect to the size of the graph. However, computing the superbubbles in each directed acyclic subgraph requires $O(m \log m)$ time [10], which dominates the time bound of the algorithm. In this paper, we propose a new $O(n + m)$-time algorithm to compute all superbubbles in a directed acyclic graph.

This paper is organised as follows: In Section 2 we define superbubbles and introduce some of their properties, and in Section 3 we outline the $O(n + m)$-time algorithm for computing superbubbles in a directed acyclic graph. In Section 4 we explain a method to validate a candidate superbubble in constant time. The algorithm is analysed in Section 5, while Section 6 provides some final remarks and directions for future research.

### 2. Properties

The concept of superbubbles was introduced and formally defined in [9] as follows.
Definition 1 ([9]). Let $G = (V, E)$ be a directed graph. For any ordered pair of distinct vertices $s$ and $t$, $(s, t)$ is called a superbubble if it satisfies the following:

- **reachability**: $t$ is reachable from $s$;
- **matching**: the set of vertices reachable from $s$ without passing through $t$ is equal to the set of vertices from which $t$ is reachable without passing through $s$;
- **acyclicity**: the subgraph induced by $U$ is acyclic, where $U$ is the set of vertices satisfying the matching criterion;
- **minimality**: no node in $U$ other than $t$ forms a pair with $s$ that satisfies the conditions above;

Nodes $s$ and $t$, and $U\setminus\{s, t\}$ used in the above definition are the superbubbles entrance, exit and interior, respectively.

We note that a supperbubble $(s, t)$ in the above definition is equivalent to a single source, single sink, acyclic directed subgraph of $G$ with source $s$ and sink $t$, which does not have any cut vertices and preserves all in-degrees and out-degrees of vertices in $U\setminus\{s, t\}$, as well as the out-degree of $s$ and in-degree of $t$.

We next state a few important properties of superbubbles which enable the linear-time enumeration of superbubbles. Lemmas 1 and 2 were proved by Onodera et al. [9] and Sung et al. [10], respectively.

**Lemma 1 ([9])**. Any node can be the entrance (respectively exit) of at most one superbubble.

Note that Lemma 1 does not exclude the possibility that a node is the entrance of a superbubble and the exit of another superbubble.

**Lemma 2 ([10])**. Let $G$ be a directed acyclic graph. We have the following two observations.

1) Suppose $(p, c)$ is an edge in $G$, where $p$ has one child and $c$ has one parent, then $(p, c)$ is a superbubble in $G$.

2) For any superbubble $(s, t)$ in $G$, there must exist some parent $p$ of $t$ such that $p$ has exactly one child $t$. 

4
In this paper we prove another important property of superbubbles that is closely related to Lemma 2.

**Lemma 3.** For any superbubble \(\langle s, t \rangle\) in a directed acyclic graph \(G\), there must exist some child \(c\) of \(s\) such that \(c\) has exactly one parent \(s\).

**Proof.** Assume that all the children of \(s\) have more than one parent. Then, there must be some cycle or some child \(c\) which has a parent that does not belong to the superbubble \(\langle s, t \rangle\). This is a contradiction. \(\square\)

3. Finding a Superbubble in a Directed Acyclic Graph

The main contribution of this paper is an algorithm, **SuperBubble**, that reports all superbubbles in a directed acyclic graph \(G = (V, E)\). The algorithm expects \(G\) to have exactly one source (the node with in-degree 0) and one sink (node with out-degree 0). If \(G\) has more than one source then a new source node \(r'\) is added to \(V\) and an edge from \(r'\) to each existing source is added to \(E\). The same can be done if \(G\) has more than one sink; in this case, a new sink node \(t'\) is added to \(V\) and an edge from each existing sink to \(t'\) is added to \(E\). Consequently, the algorithm is expected to report only those superbubbles which do not start at \(r'\) and do not end at \(t'\).

Algorithm **SuperBubble** starts by topologically ordering the nodes of the graph \(G\) and then identifying all the candidates of the possible entrances and exits of superbubbles according to Lemmas 2 and 3. The aim of this algorithm is accomplished with the help of algorithm **ValidateSuperBubble**, explained in the following section, which checks whether a given candidate \(\langle s, t \rangle\) is a superbubble or not; if it is not, the algorithm returns a possible entrance for a superbubble that ends at \(t\) (if any).

The topological ordering of a directed acyclic graph \(G\), which is the first step of Algorithm **SuperBubble**, is defined as follows. A topological ordering \(\text{ord}D\) of a directed acyclic graph \(G\) maps each node to a priority value such that \(\text{ord}D(x) < \text{ord}D(y)\) holds for all edges \((x, y) \in E\). There exist classical linear-time algorithms for computing the topological ordering of a directed acyclic graph [11, 12]. In its recursive form, the topological-sort algorithm visits an unvisited vertex of the graph, finds its unvisited neighbour, say \(v\), and performs another topological sort starting from \(v\). The algorithms returns if the current vertex does not have unvisited neighbours.
The RecursiveTopologicalSort algorithm is given below and a topological ordering of graph \( G \) in Figure 1 obtained by this algorithm is given in Figure 2.

**Proposition 1.** For any topological ordering \( \text{ordD} \) of vertices in graph \( G \), if vertex \( u \) is reachable from \( v \), i.e., if there is a path from \( v \) to \( u \) then \( \text{ordD}[v] < \text{ordD}[u] \).

**Proof.**

Importantly, in this paper we do not consider all topological orderings of graph \( G \) but only those obtained by RecursiveTopologicalSort algorithm. We next present another important property of such orderings.

**Proposition 2.** If node \( u \) is reachable from \( v \), and \( \text{ordD}[v] < \text{ordD}[w] < \text{ordD}[u] \) for some node \( w \), then \( w \) is reachable from \( v \), i.e., there is a path from \( v \) to \( w \).

**Proof.**

We next show that in an ordering obtained by RecursiveTopologicalSort all vertices that form a supperbubble are consecutive.

**Lemma 4.** Given a directed graph \( G = (V, E) \) containing a supperbubble \( \langle s, t \rangle \), a topological ordering obtained by RecursiveTopologicalSort has the following properties.

1. For all \( x \) such that \( x \in U \setminus \{s, t\} \), \( \text{ordD}[s] < \text{ordD}[x] < \text{ordD}[t] \)
2. For all \( y \) such that \( y \notin U \), \( \text{ordD}[y] < \text{ordD}[s] \) or \( \text{ordD}[y] \geq \text{ordD}[t] \).

**Proof.** Let \( G = (V, E) \) be a directed graph containing a supperbubble \( \langle s, t \rangle \).

1. Since there is a path from the start \( s \) of the supperbubble to all \( x \in U \setminus \{s\} \), by Proposition 1 \( \text{ordD}[s] < \text{ordD}[x] \) for all \( x \) such that \( x \in U \setminus \{s, t\} \). Similarly, since there is a path from all \( x \in U \setminus \{s\} \) to the end \( t \) of the supperbubble, by Proposition 1 \( \text{ordD}[x] < \text{ordD}[t] \) for all \( x \) such that \( x \in U \setminus \{t\} \). Therefore, for all \( x \) such that \( x \in U \setminus \{s, t\} \), \( \text{ordD}[s] < \text{ordD}[x] < \text{ordD}[t] \).
Figure 1: A directed acyclic graph $G = (V, E)$ with set of nodes $V = \{v_1, v_2, \cdots, v_{15}\}$. Note that $G$ has single source $v_1$ and single sink $v_{14}$.

Figure 2: Nodes of Figure 1 in topological order, where $\text{ord}_D[v_1] = 1$, $\text{ord}_D[v_2] = 2$, $\text{ord}_D[v_3] = 3$, $\text{ord}_D[v_4] = 11$, $\text{ord}_D[v_5] = 6$, $\text{ord}_D[v_6] = 8$, $\text{ord}_D[v_7] = 10$, $\text{ord}_D[v_8] = 12$, $\text{ord}_D[v_9] = 7$, $\text{ord}_D[v_{10}] = 9$, $\text{ord}_D[v_{11}] = 4$, $\text{ord}_D[v_{12}] = 5$, $\text{ord}_D[v_{13}] = 13$, $\text{ord}_D[v_{14}] = 15$ and $\text{ord}_D[v_{15}] = 14$

2. Suppose the opposite, that is, suppose there exist $y \notin U$ such that $\text{ord}_D[s] < \text{ord}_D[y] < \text{ord}_D[t]$. Since there exist a path from $s$ to $t$, by Proposition 2 there also exists a path from $s$ to $y$. However, by the definition of the supperbubble, the only vertices reachable from $s$ without going through $t$ are the internal nodes of the supperbubble - a contradiction. Therefore, for all $y$ such that $y \notin U$, $\text{ord}_D[y] < \text{ord}_D[s]$ or $\text{ord}_D[y] \geq \text{ord}_D[t]$. 

\[\langle s, t \rangle\] we have $\text{ord}_D[s] < \text{ord}_D[x] < \text{ord}_D[t]$ for all $x \in U\{s, t\}$, ...

Algorithm SUPERBUBBLE then checks each node in $V$, in topological or-
Figure 3: Candidates list for Figure 1, candidates = \{v_1, v_3(\text{exit}), v_11, v_{12}, v_5, v_{10}, v_7, v_8(\text{exit}), v_8(\text{entrance}), v_{13}, v_{14}\}. Note that both v_3 and v_8 appear twice in the list.

derer, to identify whether it is an exit or an entrance candidate (or both). According to Lemmas 2 and 3 a node v is an exit candidate if it has at least one parent with exactly one child (out-degree 1) and an entrance candidate if it has at least one child with exactly one parent (in-degree 1). The size of the candidates list is at most 2n, thus the cost of constructing a doubly-linked list of all the candidates is clearly linear in the size of G. The elements of the candidates list are expected to be ordered according to \(ordD\), and each candidate is labelled as an exit or an entrance candidate. Note that if node v is both an exit and an entrance candidate, then v appears twice in the candidates list; first as an exit and then as an entrance candidate (Figure 3). Additionally, each node in the candidates list points to the nearest previous entrance candidate; this is the node returned by a call to PreviousEntrance() in the algorithm below.

Algorithm SuperBubble processes the candidates list of G in decreasing topological order (backwards). Let \(v'_1, v'_2, \ldots, v'_\ell\) be the list of candidates, SuperBubble examines the candidates in decreasing order:

- If \(v'_j\) is an entrance candidate, then delete \(v'_j\);
- If \(v'_j\) is an exit candidate, then ReportSuperBubble is called to find and report the superbubble ending at \(v'_j\), that is superbubble \(\langle v'_i, v'_j \rangle\), for some entrance candidate \(v'_i\). ReportSuperBubble also recursively finds and reports all nested superbubbles between \(v'_i\) and \(v'_j\).

Note that ReportSuperBubble is called for each exit candidate in decreasing order either by algorithm SuperBubble or through a recursive call to identify a nested superbubble. A call to ReportSuperBubble\((\text{start}, \text{exit})\) checks the possible entrance candidates between \text{start} and \text{exit} starting with
the nearest previous entrance candidate (to exit). For the graph in Figure 1, the algorithm detects and reports five superbubbles: \(v_8, v_{14}\), \(v_3, v_8\), \(v_5, v_7\), \(v_{11}, v_{12}\) and \(v_1, v_3\). Here, both \(v_5, v_7\) and \(v_{11}, v_{12}\) are nested superbubbles.

**SuperBubble** (G)

1. for each node \(v\) in topological order do
2.   if Exit\((v)\) then
3.     candidates ← InsertExit\((v)\)
4.   if Entrance\((v)\) then
5.     candidates ← InsertEntrance\((v)\)
6. while candidates is not empty do
7.   \(v'\) ← Tail\((candidates)\)
8.   if Entrance\((v')\) then
9.     DeleteTail\((candidates)\)
10. else ReportSuperBubble\((\text{Head}(candidates), v')\)

**ReportSuperBubble**\((\text{start}, \text{exit})\)

1. ▶ Report the superbubble ending at exit (if any)
2. ▶ this superbubble may start anywhere between start and exit
3. if \((\text{start} = \text{nil}) \lor (\text{exit} = \text{nil}) \lor (\text{ordD}[\text{start}] \geq \text{ordD}[\text{exit}])\) then
4.   DeleteTail\((candidates)\)
5. return
6. \(s\) ← PreviousEntrance\((\text{exit})\)
7. while \((\text{ordD}[s] \geq \text{ordD}[\text{start}])\) do
8.   valid ← ValidateSuperBubble\((s, \text{exit})\)
9.   if \((\text{valid} = s) \lor (\text{valid} = \text{mark}[s]) \lor (\text{valid} = -1)\) then
10.     break
11.   \text{mark}[s] ← valid
12.   \(s\) ← valid
13. DeleteTail\((candidates)\)
14. if \((\text{valid} = s)\) then
15.   Print\((\langle s, \text{exit} \rangle)\)
16. while (Tail\((candidates)\) is not \(s\)) do
17.   if Exit\((\text{Tail}(candidates))\) then
18.     ▶ Check for nested superbubbles
19.     ReportSuperBubble\((\text{Next}(s), \text{Tail}(candidates))\)
20. else DeleteTail\((candidates)\)
21. return
Remark 1. We use mark to keep track of the entrance candidates which we have already checked prior to the current exit position being considered. It allows us to avoid checking the same path of entrance candidates repeatedly.

Remark 2. It is also possible to design the algorithm so as to move forward in topological order instead of backwards.

For the graph $G$ in Figure 1, $\text{SUPERBUBBLE}(G)$ makes exactly three calls to $\text{REPORTSUPERBUBBLE}$:

1. $\text{REPORTSUPERBUBBLE}(v_1, v_{14})$: First, it checks the exit candidate $v_{14}$ against the nearest previous entrance candidate, i.e., node $v_{13}$. $\text{VALIDATESUPERBUBBLE}(v_{13}, v_{14})$ returns $v_8$ as an alternative entrance candidate. The new candidate is then checked and the superbubble $\langle v_8, v_{14} \rangle$ is reported.

2. $\text{REPORTSUPERBUBBLE}(v_1, v_8)$: First, it checks the exit candidate $v_8$ against the nearest previous entrance candidate, i.e., node $v_5$. $\text{VALIDATESUPERBUBBLE}(v_5, v_8)$ returns $v_3$ as an alternative entrance candidate. The new candidate is then checked and the superbubble $\langle v_3, v_8 \rangle$ is reported. Additionally, two recursive calls are made:
   (a) $\text{REPORTSUPERBUBBLE}(v_{11}, v_7)$: First, it validates $\langle v_5, v_7 \rangle$ and reports it. Then, it makes a recursive call to $\text{REPORTSUPERBUBBLE}(v_{10}, v_{10})$ which terminates without reporting any superbubble.
   (b) $\text{REPORTSUPERBUBBLE}(v_{11}, v_{12})$: It validates $\langle v_{11}, v_{12} \rangle$ and reports it.

3. $\text{REPORTSUPERBUBBLE}(v_1, v_3)$: It validates $\langle v_1, v_3 \rangle$ and reports it.

4. Validating a Superbubble

In this section, we describe the $\text{VALIDATESUPERBUBBLE}$ subroutine. The ability to validate a candidate superbubble depends on the following result related to the Range Minimum Query problem.

The Range Minimum Query problem, RMQ for short, is to preprocess a given array $A[1 \ldots n]$ for subsequent queries of the form: “Given indices $i, j$, what is the minimum value of $A[i \ldots j]$?”. The problem has been studied intensively for decades and several $\langle O(n), O(1) \rangle$-RMQ data structures have been proposed, many of which depend on the equivalence between the range minimum query and the lowest common ancestor problems [13, 14, 15].

In order to check whether a superbubble candidate $\langle s, t \rangle$ is a superbubble or not, we propose to utilize the range min/max query problem as follows:
Figure 4: OutParent and OutChild arrays for the graph in Figure 1.

- For a given graph \( G = (V, E) \) and for each node \( v \in V \) with topological order \( \text{ord}_D(v) \), calculate the topological orderings of the parent and the child of \( v \) that are topologically furthest from \( v \).

\[
\text{OutParent}[\text{ord}_D(v)] = \min(\{\text{ord}_D(u) \mid (u, v) \in E\})
\]

\[
\text{OutChild}[\text{ord}_D(v)] = \max(\{\text{ord}_D(u) \mid (v, u) \in E\})
\]

- For a given superbubble candidate \( \langle s, t \rangle \), where \( s \) and \( t \) are an entrance and an exit candidate respectively (satisfying Lemmas 1 and 2), if \( \langle s, t \rangle \) is a superbubble then the following two conditions are valid

\[
\text{RangeMin}(\text{OutParent}, \text{ord}_D[s] + 1, \text{ord}_D[t]) = \text{ord}_D[s],
\]

\[
\text{RangeMax}(\text{OutChild}, \text{ord}_D[s], \text{ord}_D[t] - 1) = \text{ord}_D[t],
\]

where for an integer array \( A \), and indices \( i \) and \( j \), \( \text{RangeMin}(A, i, j) \) returns the index of the minimum value of \( A[i \ldots j] \). On the other hand, \( \text{RangeMax}(A, i, j) \) returns the index of the maximum value of \( A[i \ldots j] \).

For example, Figure 4 represents both OutParent and OutChild arrays computed for graph \( G \) in Figure 1. Furthermore, a candidate \( \langle v_5, v_8 \rangle \) is not a superbubble as \( \text{RangeMin}(\text{OutParent}, \text{ord}_D[v_5] + 1, \text{ord}_D[v_8]) = 3 \neq 6 = \text{ord}_D[v_5] \).

It should be clear that after \( O(n + m) \) pre-processing, validating a superbubble requires \( O(1) \) time which is the cost for range max/min query. Algorithm ValidateSuperBubble \( \langle \text{startNode}, \text{endNode} \rangle \) is designed to return a suitable entrance candidate for a superbubble ending at \( \text{endNode} \) as follows.
VALIDATESUPERBUBBLE((startNode, endNode))

1  start ← ordD[startNode]
2  end ← ordD[endNode]
3  outchild ← RANGE_MAX(OutChild, start, end-1)
4  outparent ← RANGE_MIN(OutParent, start+1, end)
5  if outchild ≠ end then
6      return −1
7  if outparent = start then
8      return startNode
9  else return preEnt[Node(outparent)]

Note that Algorithm VALIDATESUPERBUBBLE requires for each node \( v \), the nearest entrance candidate, which can be retrieved in \( O(1) \) time after an \( O(n) \)-time preprocessing. Specifically, the nodes are preprocessed from left to right, according to their topological order, where for each node, the most recent seen entrance node is stored; \( preEnt[v] = s \) such that \( ordD[s] \leq ordD[v] \), and \( ordD[s] \) is as large as possible. Note that in case \( v \) is an entrance candidate, \( preEnt[v] = v \). For instance, note that for the aforementioned example graph, \( preEnt[v_6] = v_5 \) and \( preEnt[v_{13}] = v_{13} \).

5. Algorithm Analysis

In this section, we analyse the correctness and the running time requirement of the proposed algorithm.

**Lemma 5.** For a given exit candidate \( e \), let \( s \) be the first entrance candidate which has been returned by the subroutine VALIDATESUPERBUBBLE\((s, e)\). Then \( \langle s, e \rangle \) satisfies the minimality criterion for the superbubble starting at \( s \) and ending at \( t \).

**Proof.** By contradiction, let \( e' \) be an exit candidate such that \( \langle s, e' \rangle \) is a superbubble and \( ordD[s] < ordD[e'] < ordD[e] \). Then, either \( ordD[e] = ordD[e'] + 1 \) or there is at least one node \( v \) such that \( ordD[e'] < ordD[v] < ordD[e] \).

In the first case, \( ordD[e] = ordD[e'] + 1 \) implies \( e \) is the only child of \( e' \) which not only makes \( e' \) a start candidate but also \( \langle e', e \rangle \) a superbubble by definition. If that had been the case, VALIDATESUPERBUBBLE would have been called for \( \langle e', e \rangle \) before \( \langle s, e \rangle \), and \( e' \) would have been the first entrance candidate returned (not \( s \)).
For the second case, each node \( v \), such that \( \text{ord}D[e'] < \text{ord}D[v] < \text{ord}D[e] \), is reachable from \( s \) through \( e' \). This is because \( \langle s, e \rangle \) is a valid superbubble and it is also true for all the children of \( e' \). However, at least one child of \( e' \) has exactly one parent, otherwise there must be some cycle. Thus, the node \( e' \) is also a start candidate but also \( \langle e', e \rangle \) a superbubble by definition. If that had been the case, \text{VALIDATESUPERBUBBLE} would have been called for \( \langle e', e \rangle \) before \( \langle s, e \rangle \), and \( e' \) would have been the first entrance candidate returned (not \( s \)).

In both situations, \( s \) cannot be the nearest start candidate to the left of \( e \) returned by \text{VALIDATESUPERBUBBLE}(s, e) which completes the proof. □

**Lemma 6.** Given \( s \) and \( t \), the candidates for an entrance and an exit of a superbubble in \( G \), respectively, subroutine \text{VALIDATESUPERBUBBLE} correctly checks whether \( \langle s, t \rangle \) is a superbubble or not.

**Proof.** Let \( \text{start} \) and \( \text{end} \) be two integers such that \( \text{ord}D[s] = \text{start} \) and \( \text{ord}D[t] = \text{end} \). The directed acyclic graph \( G = (V, E) \) as defined, has a single root \( r \) and a single terminal \( r' \); this implies that any node \( v \in V \) is reachable from \( r \) and, at the same time, can reach \( r' \). This is also true for \( s \), \( t \) and any node \( v \) such that \( \text{ord}D[s] < \text{ord}D[v] < \text{ord}D[t] \).

First, we will show that \( t \) is reachable from \( s \). Recall that \( t \) is an exit candidate, so, it has a parent \( p \) with out-degree 1. Assume that \( t \) is not reachable from \( s \), then there must be a path from \( r \rightsquigarrow t \) which does not involve \( s \). This implies that either \( \text{OutParent}[\text{end}] < \text{start} \) or there exists a node \( v \) such that \( \text{start} < \text{ord}D[v] < \text{end} \), \( \text{OutParent}[v] < \text{start} \) and there exists a path \( r \rightsquigarrow v \rightsquigarrow t \) which is a contradiction.

Similarly, we can show that \( s \) can reach \( t \). Consequently, any node \( v \) such that \( \text{start} < \text{ord}D[v] < \text{end} \) satisfies the matching criterion of the superbubble.

The acyclicity criterion is guaranteed by acyclicity of \( G \) and the minimality is satisfied by the design of Algorithm \text{REPORTSUPERBUBBLE} which assigns each exit of a superbubble to the nearest entrance and by the correctness of Lemma 5. □

**Theorem 1.** Given a directed acyclic graph \( G = (V, E) \), where \( n = |V| \) and \( m = |E| \), algorithm \text{SUPERBUBBLE} finds all superbubbles in \( G \) in decreasing order of the topological ordering of their exit nodes in \( O(n + m) \) time.
Proof. Consider a run of SuperBubble and let \( \langle s_1, t_1 \rangle, \langle s_2, t_2 \rangle, \ldots, \langle s_k, t_k \rangle \) be the successive superbubbles reported just after the execution of Line 15 of ReportSuperBubble, where \( \text{ordD}(t_1) > \text{ordD}(t_2) > \cdots > \text{ordD}(t_k) \). First, we show that each \( \langle s_i, t_j \rangle \) reported by the algorithm in Line 15 is a superbubble. This is proved by the correctness of Lemma 6. Also, no superbubble is missed out by the algorithm as all exit candidates are considered exactly once and later deleted from the candidates list.

Secondly, we have already explained that the running time cost of the topological sorting of \( G \) and computing the candidates list is \( \mathcal{O}(n + m) \). Furthermore, all the list’s operations cost constant time each, and sum up to a linear cost \( \mathcal{O}(n) \), as there are at most \( 2n \) candidates in the list. Finally, the total number of times ValidateSuperBubble subroutine is called is \( \mathcal{O}(m) \). This is achieved by marking each entrance candidate (Line 11) to avoid repeated checks. Each call for ValidateSuperBubble costs \( \mathcal{O}(1) \).

Therefore, for a directed acyclic graph \( G = (V, E) \), the total running time for reporting all superbubbles is \( \mathcal{O}(n + m) \), where \( n = |V| \) and \( m = |E| \). □

6. Final Remarks

We presented an \( \mathcal{O}(n + m) \)-time algorithm to compute all superbubbles in a directed acyclic graph, where \( n \) is the number of nodes and \( m \) is the number of edges. Our next goal is to practically evaluate our algorithm and compare its behaviour with an earlier result [9]. It would also be interesting to investigate other superbubble-like structures in assembly graphs, such as complex bulges [16].

References

[1] E. S. Lander, L. M. Linton, B. Birren, C. Nusbaum, M. C. Zody, J. Baldwin, K. Devon, K. Dewar, M. Doyle, W. FitzHugh, et al., Initial sequencing and analysis of the human genome, Nature 409 (6822) (2001) 860–921.

[2] J. C. Venter, M. D. Adams, E. W. Myers, P. W. Li, R. J. Mural, G. G. Sutton, H. O. Smith, M. Yandell, C. A. Evans, R. A. Holt, et al., The sequence of the human genome, Science 291 (5507) (2001) 1304–1351.

[3] S. Balasubramanian, D. Kleereman, C. Barnes, M. Osborne, Patent US20077232656 (2007).
[4] S. Batzoglou, Algorithmic challenges in mammalian genome sequence assembly, Encyclopedia of genomics, proteomics and bioinformatics, John Wiley and Sons, Hoboken (New Jersey).

[5] J. Butler, I. MacCallum, M. Kleber, I. A. Shlyakhter, M. K. Belmonte, E. S. Lander, C. Nusbaum, D. B. Jaffe, ALLPATHS: de novo assembly of whole-genome shotgun microreads, Genome Research 18 (5) (2008) 810–820.

[6] P. A. Pevzner, H. Tang, M. S. Waterman, An Eulerian path approach to DNA fragment assembly, Proceedings of the National Academy of Sciences of the U. S. A. 98 (17) (2001) 9748–9753.

[7] N. G. de Bruijn, A combinatorial problem, Koninklijke Nederlandse Akademie v. Wetenschappen 49 (1946) 758–764.

[8] D. R. Zerbino, E. Birney, Velvet: algorithms for de novo short read assembly using de Bruijn graphs, Genome Research 18 (5) (2008) 821–829.

[9] T. Onodera, K. Sadakane, T. Shibuya, Detecting superbubbles in assembly graphs, in: WABI, 2013, pp. 338–348.

[10] W. Sung, K. Sadakane, T. Shibuya, A. Belorkar, I. Pyrogova, An O(m log m)-time algorithm for detecting superbubbles, IEEE/ACM Trans. Comput. Biology Bioinform. 12 (4) (2015) 770–777.

[11] R. L. R. Thomas H. Cormen, Charles E. Leiserson, C. Stein, Introduction to Algorithms, MIT Press, Cambridge, MA., 2001.

[12] R. Tarjan, Edge-disjoint spanning trees and depth-first search, Acta Informatica 6 (2) (1976) 171–185.

[13] D. Harel, R. Tarjan, Fast algorithms for finding nearest common ancestors, SIAM Journal on Computing 13(2) (1984) 338–355.

[14] J. Fischer, V. Heun, Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE, in: M. Lewenstein, G. Valiente (Eds.), Combinatorial Pattern Matching, 17th Annual Symposium, CPM 2006, Barcelona, Spain, July 5-7, 2006, Proceedings, Vol. 4009 of Lecture Notes in Computer Science, Springer, 2006, pp. 36–48.
[15] S. Durocher, A simple linear-space data structure for constant-time range minimum query, in: A. Brodnik, A. Lopez-Ortiz, V. Raman, A. Viola (Eds.), Space-Efficient Data Structures, Streams, and Algorithms, Vol. 8066 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2013, pp. 48–60.

[16] S. Nurk, A. Bankevich, D. Antipov, A. A. Gurevich, A. Korobeynikov, A. Lapidus, A. D. Prjibelski, A. Pyshkin, A. Sirotkin, Y. Sirotkin, R. Stepanauskas, S. R. Clingenpeel, T. Woyke, J. S. McLean, R. Lasken, G. Tesler, M. A. Alekseyev, P. A. Pevzner, Assembling single-cell genomes and mini-metagenomes from chimeric MDA products, Journal of Computational Biology 20 (10) (2013) 714–737.