Linear-time superbubble identification algorithm for genome assembly

Ljiljana Brankovic [a,b], Costas S. Iliopoulos [b], Ritu Kundu [b], Manal Mohamed [b], Solon P. Pissis [b,*], Fatima Vayani [b]

[a] School of Electrical Engineering and Computer Science, The University of Newcastle, Newcastle NSW 2308, Australia
[b] Department of Informatics, King's College London, London WC2R 2LS, United Kingdom

A R T I C L E   I N F O

A B S T R A C T

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 utilise 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 $\mathcal{O}(n + m)$-time algorithm to detect all superbubbles in a directed acyclic graph with n vertices and m (directed) edges, improving the best-known $\mathcal{O}(m \log m)$-time algorithm by Sung et al.

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, and RNA molecules from a particular sample, 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 vertex in an overlap graph and each detected overlap as a directed edge between the vertices corresponding to overlapping reads. These methods have proved their use through numerous *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 *de Bruijn graph* [7], each $k - 1$ nucleotide long prefix and suffix of the k-mers is represented as a vertex and each k-mer

* Corresponding author.
E-mail addresses: ljiljana.brankovic@newcastle.edu.au (L. Brankovic), costas.iliopoulos@kcl.ac.uk (C.S. Iliopoulos), ritu.kundu@kcl.ac.uk (R. Kundu), manal.mohamed@kcl.ac.uk (M. Mohamed), solon.pissis@kcl.ac.uk (S.P. Pissis), fatima.vayani@kcl.ac.uk (F. Vayani).

is represented as a directed edge between its prefix and suffix vertices. The marginal information contained by a k-mer is its last nucleotide. The sequence of those final nucleotides is called the sequence of the vertex. 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 have been 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]. In particular, a *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 generalisation of a bubble, the so-called superbubble, was proposed as an important subgraph class for analysing assembly graphs [9]. A *superbubble* is defined as a minimal subgraph H in the de Bruijn graph with exactly one start vertex s and one end vertex t such that: (1) H is a directed, acyclic, single-source (s), single-sink (t) graph (2) there is no edge from a vertex not in H going to a vertex in $H\backslash\{s\}$ and (3) there is no edge from a vertex in $H\backslash\{t\}$ going to a vertex 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 $\mathcal{O}(nm)$-time algorithm to detect superbubbles, where n is the number of vertices and m is the number of edges in the graph [9]. Very recently, Sung et al. gave an improved $\mathcal{O}(m\log m)$-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 these 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 vertices.

Our contribution. 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 $\mathcal{O}(m\log m)$ time [10], which dominates the time bound of the algorithm. In this paper, we propose a new $\mathcal{O}(n+m)$-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 $\mathcal{O}(n+m)$-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. (See [9].) Let $G = (V, E)$ be a directed graph. For any ordered pair of distinct vertices s and t, $\langle s, t \rangle$ 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 vertex in U other than t forms a pair with s that satisfies the conditions above;

vertices s and t, and $U\backslash\{s, t\}$ used in the above definition are the superbubble's *entrance*, *exit* and *interior*, respectively.

We note that a superbubble $\langle s, t \rangle$ in the above definition is equivalent to a single-source, single-sink, directed acyclic 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 $U\backslash\{s, t\}$, 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. *(See [9].) Any vertex can be the entrance (respectively exit) of at most one superbubble.*

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

Lemma 2. *(See [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 $\langle p, c \rangle$ is a superbubble in G.
2) For any superbubble $\langle s, t \rangle$ in G, there must exist some parent p of t such that p has exactly one child t.
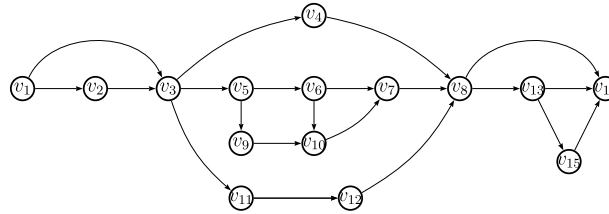
Fig. 1. A graph G with set of vertices $V = \{v_1, v_2, \cdots, v_{15}\}$. Note that G has as a single source v_1 and as a single sink v_{14}.
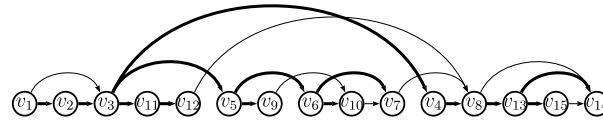
Fig. 2. Vertices of Fig. 1 in topological order, where $ordD[v_1] = 1$, $ordD[v_2] = 2$, $ordD[v_3] = 3$, $ordD[v_4] = 11$, $ordD[v_5] = 6$, $ordD[v_6] = 8$, $ordD[v_7] = 10$, $ordD[v_8] = 12$, $ordD[v_9] = 7$, $ordD[v_{10}] = 9$, $ordD[v_{11}] = 4$, $ordD[v_{12}] = 5$, $ordD[v_{13}] = 13$, $ordD[v_{14}] = 15$ and $ordD[v_{15}] = 14$.

In this paper we start by showing 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 $G = (V, E)$ with exactly one source (the vertex with in-degree 0) and exactly one sink (vertex with out-degree 0). If G has more than one source then a new source vertex r' is added to V and an edge from r' to each existing source is added to E. The same is done if G has more than one sink; in this case, a new sink vertex t' is added to V and an edge from each existing sink to t' is added to E. If such preprocessing is done, then among the superbubbles reported by the algorithm, only those which do not start at r' and do not end at t' represent the superbubbles in the original graph. For the sake of simplicity, for the rest of this paper and in all the propositions, lemmas and theorems that follow, we use G to denote a directed acyclic graph with exactly one source and exactly one sink, and we use n and m to denote the number of its vertices and edges respectively, that is, for $G = (V, E)$ we have $n = |V|$ and $m = |E|$.

A *topological ordering ordD* of G maps each vertex to an integer between 1 and n, such that $ordD[x] < ordD[y]$ holds for all edges $(x, y) \in E$. There exists a classical linear-time algorithm for computing the topological ordering of a directed acyclic graph [11,12]. In its recursive form, the algorithm visits an unvisited vertex of the graph, finds its unvisited neighbour, say v, and performs another topological sort starting from v. The algorithm *returns* if the current vertex does not have unvisited neighbours. Algorithm TOPOLOGICALSORT, given below, is a simplified version that takes as input a single-source, single-sink directed acyclic graph, and produces a topological ordering of vertices. For the graph G in Fig. 1, TOPOLOGICALSORT produces an ordering given in Fig. 2.

TOPOLOGICALSORT(G)

```
1   order ← n
2   for each vertex v ∈ V do
3       state[v] ← unvisited
4   RECURSIVETOPOLOGICALSORT(G, source)
```

RECURSIVETOPOLOGICALSORT(G, v)

```
1   state[v] ← visited
2   for each vertex w adjacent to v do
3       if state[w] = unvisited then
4           RECURSIVETOPOLOGICALSORT(G, w)
5   ordD[v] ← order
6   order ← order − 1
```

Proposition 1. *For any topological ordering ordD of vertices in graph G, if vertex u is reachable from v, that is, if there is a path from v to u, then $ordD[v] < ordD[u]$.*

j	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	v_1	v_2	v_3	v_{11}	v_{12}	v_5	v_9	v_6	v_{10}	v_7	v_4	v_8	v_{13}	v_{15}	v_{14}
entrance	s_1		s_2	s_3		s_4						s_5	s_6		
exit			t_1		t_2				t_3	t_4		t_5			t_6

Fig. 3. Candidates list for Fig. 1, *candidates* $= \{v_1(\text{entrance}),\ v_3(\text{exit}),\ v_3(\text{entrance}),\ v_{11}(\text{entrance}),\ v_{12}(\text{exit}),\ v_5(\text{entrance}),\ v_{10}(\text{exit}),\ v_7(\text{exit}),\ v_8(\text{exit}),$ $v_8(\text{entrance}),\ v_{13}(\text{entrance}),\ v_{14}(\text{exit})\}$. Note that both v_3 and v_8 appear twice in the list.

Proof. If the path from v to u is of length 1, i.e., there is an edge (v, u), then by the definition of topological ordering we have $ordD[v] < ordD[u]$. Otherwise, we denote the path from v to u of length k, $k > 1$, as $v, x_1, \dots, x_{k-1}, u$. Then by the definition of topological ordering we have $ordD[v] < ordD[x_1] < \cdots < ordD[u]$. Transitively, we have $ordD[v] < ordD[u]$. □

Importantly, in this paper we do not consider all topological orderings of graph G but only those obtained by algorithm TopologicalSort. Note that this algorithm finds a directed spanning tree T of G rooted at the *source*, which contains a path from the *source* to any vertex in G. The directed spanning tree T of G obtained by algorithm TopologicalSort is presented by bold edges in Fig. 2. It may be worth mentioning that a directed rooted tree is also known as *arborescence*.

We next present another important property of topological ordering obtained by algorithm TopologicalSort.

Proposition 2. *Let ordD and T be a topological ordering and a directed rooted spanning tree of graph G obtained by algorithm* TopologicalSort. *If there is a path in T from a vertex v to a vertex u, then, for each vertex w such that $ordD[v] < ordD[w] < ordD[u]$, there is a path from v to w.*

Proof. Recall that T contains a path from the root to each vertex of the tree; this is also true for each subtree of T. Furthermore, if there is a path from v to u in T, then u is contained in a subtree of T rooted at v, and each w such that $ordD[v] < ordD[w] < ordD[u]$ is also contained in the subtree rooted at v (but not in the subtree rooted at u). Therefore, there is a path from v to w, for each w such that $ordD[v] < ordD[w] < ordD[u]$. □

We next show that in an ordering obtained by TopologicalSort, a vertex has the topological ordering between the orderings of the entrance and the exit of a superbubble if and only if it belongs to the superbubble.

Lemma 4. *Let graph G contain a superbubble $\langle s, t \rangle$. Then a topological ordering obtained by* TopologicalSort *has the following properties.*

1. *For all x such that $x \in U \setminus \{s, t\}$, $ordD[s] < ordD[x] < ordD[t]$.*
2. *For all y such that $y \notin U$, $ordD[y] < ordD[s]$ or $ordD[y] > ordD[t]$.*

Proof. Recall that U is the set of vertices forming a superbubble (see Definition 1).

1. Since there is a path from the start s of the superbubble to all $x \in U \setminus \{s\}$, by Proposition 1 we have $ordD[s] < ordD[x]$ for all x such that $x \in U \setminus \{s\}$. Similarly, since there is a path from all $x \in U \setminus \{t\}$ to the end t of the superbubble, by Proposition 1 we have $ordD[x] < ordD[t]$ for all x such that $x \in U \setminus \{t\}$. Therefore, for all x such that $x \in U \setminus \{s, t\}$, $ordD[s] < ordD[x] < ordD[t]$.
2. Suppose the opposite, that is, suppose that there exists some $y \notin U$ such that $ordD[s] < ordD[y] < ordD[t]$. Since the superbubble $\langle s, t \rangle$ is itself a single-source, single-sink subgraph of G, any directed spanning tree of G rooted at the *source*, will contain a path from s to t. Then by Proposition 2 there also exists a path from s to y in T and thus also in G. However, by the definition of the superbubble, the only vertices reachable from s without going through t are the internal vertices of the superbubble — a contradiction. Therefore, for all y such that $y \notin U$, either $ordD[y] < ordD[s]$ or $ordD[y] > ordD[t]$. □

Algorithm SuperBubble starts by topologically ordering the vertices of graph G and then checks each vertex in V, in topological order, to identify whether it is an exit or an entrance candidate (or both). According to Lemmas 2 and 3, a vertex 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). There are at most $2n$ candidates, thus the cost of constructing a doubly-linked list of all the candidates is linear in n. The elements of the candidates list are ordered according to $ordD$, and each candidate is labelled as an exit or an entrance candidate. Note that if a vertex 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 (Fig. 3).

Algorithm SUPERBUBBLE processes the candidates list of graph G in decreasing topological order (backwards). Let $v'_1, v'_2, \ldots, v'_\ell$ be the list of candidates. The algorithm performs the following:

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

For clarity of presentation, we next provide a list and a short description of subroutines and arrays used by algorithm SUPERBUBBLE and subroutine REPORTSUPERBUBBLE. Before that, it is worth mentioning that *candidates* is a doubly-linked list of entrance and exit candidates; specifically, an element of the list is a vertex along with a label specifying if it is an entrance or an exit candidate. For the sake of simplicity of the following routines, we use a vertex and its corresponding candidate (element in the candidates list) interchangeably. This does not add to the complexity of the algorithm as we can use an auxiliary array v, where $v[i]$ stores a pointer to the corresponding element v_i in *candidates* so as to provide a constant-time conversion from a vertex to the corresponding candidate.

1. ENTRANCE(v) takes as input a vertex v and outputs TRUE if v is an entrance candidate, that is, if it satisfies Lemma 3, and FALSE otherwise.
2. EXIT(v) takes as input a vertex v and outputs TRUE if v is an exit candidate, that is, if it satisfies Lemma 2, and FALSE otherwise.
3. INSERTENTRANCE(v) takes as input a vertex v, inserts it at the end of *candidates* and labels it as *entrance*.
4. INSERTEXIT(v) takes as input a vertex v, inserts it at the end of *candidates* and labels it as *exit*.
5. HEAD(*candidates*) and TAIL(*candidates*) return the first and the last element in *candidates*, respectively.
6. DELETETAIL(*candidates*) deletes the last element in *candidates*.
7. NEXT(v) returns the candidate following v in *candidates*.

In addition to the above subroutines, the main algorithm also explicitly makes use of the following arrays.

1. The array *ordD* stores the topological order of the vertices.
2. The array *previousEntrance* stores the previous entrance candidate s for each vertex v. Formally, *previousEntrance*$[v] = s$ where s is an entrance candidate such that $ordD[s] < ordD[v]$; and there does not exist another entrance candidate s' such that $ordD[s] < ordD[s'] < ordD[v]$.
3. The array *alternativeEntrance* is used to reduce the number of *entrance* − *exit* pairs that need to be considered as possible superbubbles. Array *alternativeEntrance* is further detailed in Section 4.1.

Note that subroutine REPORTSUPERBUBBLE is called for each exit candidate in decreasing order either by algorithm SU-PERBUBBLE or through a recursive call to identify a nested superbubble. A call to subroutine REPORTSUPERBUBBLE(*start, exit*) checks the possible entrance candidates between *start* and *exit*, starting with the nearest previous entrance candidate (to *exit*). This task is accomplished with the help of subroutine 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 either a "−1" which means that no superbubble ends at t, or an alternative entrance candidate for a superbubble that could end at t. For the graph in Fig. 1, the algorithm detects and reports five superbubbles: $\langle v_8, v_{14} \rangle$, $\langle v_3, v_8 \rangle$, $\langle v_5, v_7 \rangle$, $\langle v_{11}, v_{12} \rangle$ and $\langle v_1, v_3 \rangle$. Here, both $\langle v_5, v_7 \rangle$ and $\langle v_{11}, v_{12} \rangle$ are nested superbubbles.

```
SUPERBUBBLE(G)
    1  TOPOLOGICALSORT(G)
    2  prevEnt ← NULL
    3  for each vertex v in topological order do
    4      alternativeEntrance[v] ← NULL
    5      previousEntrance[v] ← prevEnt
    6      if EXIT(v) then
    7          INSERTEXIT(v)
    8      if ENTRANCE(v) then
    9          INSERTENTRANCE(v)
   10          prevEnt ← v
   11  while candidates is not empty do
   12      if ENTRANCE(TAIL(candidates)) then
   13          DELETETAIL(candidates)
   14      else REPORTSUPERBUBBLE(HEAD(candidates),TAIL(candidates))
```

REPORTSUPERBUBBLE(*start, exit*)
```
 1   ▷ Report the superbubble ending at exit (if any)
 2   if (start = NULL) | (exit = NULL) | (ordD[start] ≥ ordD[exit]) then
 3       DELETETAIL(candidates)
 4       return
 5   s ← previousEntrance[exit]
 6   while (ordD[s] ≥ ordD[start]) do
 7       valid ←VALIDATESUPERBUBBLE(s, exit)
 8       if (valid = s) | (valid = alternativeEntrance[s]) | (valid = −1) then
 9           break
10       alternativeEntrance[s] ← valid
11       s ← valid
12   DELETETAIL(candidates)
13   if (valid = s) then
14       REPORT(⟨s, exit⟩)
15       while (TAIL(candidates) is not s) do
16           if EXIT(TAIL(candidates)) then
17               ▷ Check for nested superbubbles
18               REPORTSUPERBUBBLE(NEXT(s),TAIL(candidates))
19           else DELETETAIL(candidates)
20   return
```

Remark 1. It is also possible to design the algorithm so as to move forward in topological order instead of backwards.

For graph G in Fig. 1, algorithm SUPERBUBBLE(G) makes exactly three calls to subroutine REPORTSUPERBUBBLE:

1. REPORTSUPERBUBBLE(v_1, v_{14}): First, it checks the exit candidate v_{14} against the nearest previous entrance candidate, i.e. vertex v_{13}. Subroutine VALIDATESUPERBUBBLE(v_{13}, v_{14}) returns v_8 as an alternative entrance candidate. The new candidate is then checked and the superbubble $⟨v_8, v_{14}⟩$ is reported.
2. REPORTSUPERBUBBLE(v_1, v_8): First, it checks the exit candidate v_8 against the nearest previous entrance candidate, i.e. vertex v_5. Subroutine VALIDATESUPERBUBBLE(v_5, v_8) returns v_3 as an alternative entrance candidate. The new candidate is then checked and the superbubble $⟨v_3, v_8⟩$ is reported. Additionally, two recursive calls are made:
   (a) REPORTSUPERBUBBLE(v_{11}, v_7): First, it validates $⟨v_5, v_7⟩$ and reports it. Then, it makes a recursive call to subroutine REPORTSUPERBUBBLE(v_{10}, v_{10}) which terminates without reporting any superbubble.
   (b) REPORTSUPERBUBBLE(v_{11}, v_{12}): validates $⟨v_{11}, v_{12}⟩$ and reports it.
3. REPORTSUPERBUBBLE(v_1, v_3): validates $⟨v_1, v_3⟩$ and reports it.

4. Validating a superbubble

In this section, we describe subroutine VALIDATESUPERBUBBLE. 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 $A[1..n]$ for subsequent queries of the form: "Given indices i, j, what is the minimum value of $A[i..j]$?". The problem has been studied intensively for decades and several $⟨O(n), O(1)⟩$-RMQ data structures have been proposed, many of which depend on the equivalence between the Range Minimum Query and the Lowest Common Ancestor problems [13–15].

In order to check whether a superbubble candidate $⟨s, t⟩$ is a superbubble or not, we propose to utilise the range min/max query problem as follows:

- For a given graph $G = (V, E)$ and for each vertex $v \in V$ with topological order $ordD[v]$, calculate the topological orderings of the parent and the child of v that are topologically furthest from v.

$$OutParent[ordD[v]] = \min(\{ordD[u_i] \mid (u_i, v) \in E\}),$$

$$OutChild[ordD[v]] = \max(\{ordD[u_i] \mid (v, u_i) \in E\}).$$

- For an integer array A and indices i and j we define RANGEMIN(A, i, j) and RANGEMAX(A, i, j) to return the minimum and maximum value of $A[i..j]$, respectively.
  Then for a given superbubble candidate $⟨s, t⟩$, where s and t are an entrance and an exit candidate respectively (satisfying Lemmas 1 and 2), if $⟨s, t⟩$ is a superbubble then the following two conditions are valid

$$\text{RANGEMIN}(OutParent, ordD[s] + 1, ordD[t]) = ordD[s],$$

$$\text{RANGEMAX}(OutChild, ordD[s], ordD[t] − 1) = ordD[t].$$

j	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	v_1	v_2	v_3	v_{11}	v_{12}	v_5	v_9	v_6	v_{10}	v_7	v_4	v_8	v_{13}	v_{15}	v_{14}
OutParent[j]	–	1	1	3	4	3	6	6	7	8	3	5	12	13	12
OutChild[j]	3	3	11	5	12	8	9	10	10	12	12	15	15	15	–

Fig. 4. *OutParent* and *OutChild* arrays for the graph in Fig. 1.

For example, Fig. 4 represents both *OutParent* and *OutChild* arrays computed for graph G in Fig. 1. Furthermore, a candidate $\langle v_5, v_8 \rangle$ is not a superbubble as RANGEMIN($OutParent, ordD[v_5] + 1, ordD[v_8]$) $= 3 \neq 6 = ordD[v_5]$.

It should be clear that after an $\mathcal{O}(n + m)$-time preprocessing, validating a superbubble requires $\mathcal{O}(1)$ time which is the cost for range max/min query. Subroutine VALIDATESUPERBUBBLE(*startVertex, endVertex*) is designed to return an appropriate entrance candidate for a superbubble ending at *endVertex* (if any), as follows.

VALIDATESUPERBUBBLE(*startVertex, endVertex*)

1   *start* ← $ordD[startVertex]$
2   *end* ← $ordD[endVertex]$
3   *outchild* ←RANGEMAX($OutChild, start, end − 1$)
4   *outparent* ←RANGEMIN($OutParent, start + 1, end$)
5   **if** *outchild* $\neq$ *end* **then**
6   **return** -1
7   **if** *outparent* $=$ *start* **then**
8   **return** *startVertex*
9   **elseif** ENTRANCE(VERTEX(*outparent*)) **then**
10   **return** VERTEX(*outparent*)
11   **else return** *previousEntrance*[VERTEX(*outparent*)]

Note that subroutine VALIDATESUPERBUBBLE utilises subroutine ENTRANCE and the array *previousEntrance* defined in Section 3, as well as subroutine VERTEX that takes as input an integer i and outputs vertex v such that $ordD[i] = v$.

An important observation is that a subsequent call to subroutine VALIDATESUPERBUBBLE, for a given entrance candidate, returns alternative entrance candidates in strictly non-decreasing topological order as proved by Lemma 5.

Lemma 5. *Let t be the alternative entrance candidate returned by subroutine VALIDATESUPERBUBBLE(s, e). Then for any exit candidate e' such that $ordD[s] < ordD[e'] < ordD[e]$, the order of the alternative entrance candidate t' returned by subroutine VALIDATESUPERBUBBLE(s, e') will be greater than or equal to the order of t.*

Proof. Recall that the alternative entrance t returned by the subroutine VALIDATESUPERBUBBLE(s, e') is either a vertex with topological order *outparent*, or the *previousEntrance* of this vertex.

Since *outparent* $=$ RANGEMIN($OutParent, ordD[s] + 1, ordD[e]$), *outparent'* $=$ RANGEMIN($OutParent, ordD[s] + 1, ordD[e']$) and $ordD[s] < ordD[e'] < ordD[e]$, we have *outparent* $\leq$ *outparent'*. Therefore, $ordD(t) \leq ordD(t')$. □

4.1. Validation and alternativeEntrance

In case the validation of the candidate pair (t_0, e) fails, subroutine VALIDATESUPERBUBBLE(t_0, e) returns either "−1" or an alternative candidate t_1 which might be an entrance of a superbubble ending at e. This alternative candidate t_1 is either a vertex u_1, if u_1 is an entrance candidate, or the previous entrance candidate of u_1 such that

$$ordD[u_1] = OutParent[ordD[v_1]]$$
$$= \text{RANGEMIN}(OutParent, ordD[t_0] + 1, ordD[e]),$$

where v_1 is some vertex between t_0 and e in the topological ordering.

Suppose t_1 is also not a valid entrance of the superbubble ending at e. Then there must be a vertex v_2, $ordD[t_1] < ordD[v_2] < ordD[t_0]$, with some parent u_2, such that $ordD[u_2] = OutParent[ordD[v_2]]$. Then the alternative entrance is some t_2, which is either a vertex u_2 or its previous entrance and thus $ordD[t_2] < ordD[t_1]$. A series of such failed validations produces a sequence $t_1, t_2, \ldots$ of failed alternative entrance candidates.

An important observation here is that any entrance t_i, for $i \geq 1$, from such a sequence is an invalid entrance not only for the superbubble ending at e but also for all those ending at any other exit vertex e' such that $ordD[t_{i-1}] < ordD[e'] < ordD[e]$ and $t_i = $ VALIDATESUPERBUBBLE(t_{i-1}, e'). This is the case because the vertex v_i which causes the alternative entrance t_i to fail is such that $ordD[t_i] < ordD[v_i] < ordD[t_{i-1}]$ for $i \geq 1$. Therefore, v_i does not depend on the exit e but rather on the previous failed candidate entrance.

This is where array *alternativeEntrance* plays an important role. Storing *alternativeEntrance*[t_{i-1}] $= t_i$ for $i \geq 1$ enables us to skip this sequence at a later stage if t_i is returned by subroutine VALIDATESUPERBUBBLE(t_{i-1}, e').

5. Algorithm analysis

In this section, we analyse the correctness and the running time of the proposed algorithm SUPERBUBBLE. For simplicity, in the following lemma we will slightly abuse the terminology and refer to $\langle s, t \rangle$ as a *superbubble* if it satisfies the first three conditions given in Definition 1, and as *minimal superbubble* if it also satisfies the last condition in the same definition.

Lemma 6. *For a given exit candidate e, let s be the entrance candidate such that superbubble $\langle s, e \rangle$ is reported by subroutine* VALIDATESUPERBUBBLE(s, e). *Then $\langle s, e \rangle$ is a minimal superbubble.*

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

In the first case, $ordD[e] = ordD[e'] + 1$ implies that e is the only child of e' and e' is the only parent of e, which, by Lemma 2 makes $\langle e', e \rangle$ a superbubble.

In the second case, where there is at least one vertex v such that $ordD[e'] < ordD[v] < ordD[e]$, we also argue that $\langle e', e \rangle$ must be a superbubble. Indeed, $\langle e', e \rangle$ satisfies the following conditions:

1. **Reachability:** Since $\langle s, e \rangle$ is a superbubble, e is reachable from s; since $\langle s, e' \rangle$ is also assumed to be a superbubble, any path from s to e must go through e', therefore e is reachable from e'.
2. **Matching:** The only vertices reachable from e' without going through e are those whose topological order is between $ordD(e')$ and $ordD(e)$. Indeed, since $\langle s, e \rangle$ and $\langle s, e' \rangle$ are superbubbles, all these vertices are reachable from s through e', and no vertices with topological order greater than $ordD(e)$ are reachable from e' without going through e. Similarly, there are no edges between vertices with topological order less than $ordD(e')$ and those with the topological order between $ordD(e')$ and $ordD(e)$. Therefore, the only vertices from which e is reachable without going through e' are those whose topological order is between $ordD(e')$ and $ordD(e)$.
3. **Acyclicity:** Since $\langle s, e \rangle$ is a superbubble it is acyclic; since $\langle e', e \rangle$ is a subgraph of $\langle s, e \rangle$, it is also acyclic.

In both cases, since for each exit candidate the entrance candidates are checked in reverse topological order, subroutine VALIDATESUPERBUBBLE would have been called on $\langle e', e \rangle$ first, and would have reported $\langle e', e \rangle$ instead of $\langle s, e \rangle$. Therefore, $\langle s, e \rangle$ is a minimal superbubble. □

Lemma 7. *For the given entrance and exit candidates s and e, respectively, subroutine* VALIDATESUPERBUBBLE *reports $\langle s, t \rangle$ if and only if $\langle s, t \rangle$ is a superbubble.*

Proof. We prove the lemma by showing that if $\langle s, t \rangle$ is a superbubble then subroutine VALIDATESUPERBUBBLE reports it, and if VALIDATESUPERBUBBLE reports $\langle s, t \rangle$ then $\langle s, t \rangle$ is a superbubble.

1. We start by showing that if $\langle s, t \rangle$ is a superbubble then subroutine VALIDATESUPERBUBBLE reports it. Indeed, by Lemma 4, all the vertices with topological orderings between s and t belong to the superbubble $\langle s, t \rangle$. Therefore, the minimum *OutParent* is s and the maximum *OutChild* is t and thus subroutine VALIDATESUPERBUBBLE reports $\langle s, t \rangle$.
2. We next show that if subroutine VALIDATESUPERBUBBLE reports $\langle s, t \rangle$ then $\langle s, t \rangle$ is a superbubble. Let *start* and *end* be two integers, such that $ordD[s] = start$ and $ordD[t] = end$. The graph G as defined, has a single source r and a single sink r'; this implies that any vertex $v \in V$ is reachable from r and, at the same time, can reach r'. This is also true for s, t and for any vertex v such that $ordD[s] < ordD[v] < ordD[t]$.
   First, we 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 *OutParent*[*end*] < *start*, or there exists a vertex v such that $start < ordD[v] < end$, *OutParent*[v] < *start* and there exists a path $r \rightsquigarrow v \rightsquigarrow t$, which is a contradiction.
   Similarly, we can show that every vertex v such that $start < ordD[v] < end$ satisfies the **matching** criterion of the superbubble.
   The **acyclicity** criterion is guaranteed by the acyclicity of G and the **minimality** is satisfied by the design of subroutine REPORTSUPERBUBBLE which assigns each exit of a superbubble to the nearest entrance, and by the correctness of Lemma 6. □

Lemma 8. *For a given exit candidate e, let t be the alternative entrance candidate returned by subroutine* VALIDATESUPERBUBBLE(s, e). *Then any entrance candidate between t and e cannot be a valid entrance for the superbubble ending at e.*

Proof. By contradiction, assume that s' is an entrance candidate between t and e such that $\langle s', e \rangle$ is a superbubble. If s' had been between s and e, it would have already been reported, as SUPERBUBBLE checks entrance candidates in reverse topological order starting from e. Therefore, s' is between t and s, such that $ordD[t] < ordD[s'] < ordD[s] < ordD[e]$. Let

outparent = RangeMin(*OutParent*, *ordD*[*s*] + 1, *ordD*[*e*]). Then, vertex at *outparent* is between *t* and *s'*, otherwise subroutine ValidateSuperBubble(*s*, *e*) would have returned *s'* (instead of *t*). Therefore, *ordD*[*t*] ≤ *outparent* < *ordD*[*s'*].

Let *outparent'* = RangeMin(*OutParent*, *ordD*[*s'*] + 1, *ordD*[*e*]). Then *outparent'* ≤ *outparent*. This implies that *outparent'* ≤ *outparent* < *ordD*[*s'*]. However, for ⟨*s'*, *e*⟩ to be a valid superbubble, *outparent'* should have been equal to *ordD*[*s'*]. Hence, the assumption is wrong and thus, it is proved that there cannot be an entrance candidate, between *t* and *e*, which is a valid entrance for the superbubble ending at *e*.   □

Lemma 9. *For the given entrance and exit candidates s and e_1, respectively, let alternativeEntrance[s] be set to t_1 which later gets reset to t_2 such that $t_2 \neq t_1$, while considering s with another exit candidate e_2. Then no entrance candidate between s and e_2 can reset alternativeEntrance[s] to t_1 again.*

Proof. Let e_3 be an exit candidate between s and e_2 such that subroutine ValidateSuperBubble(s, e_3) returns t_3. Then by Lemma 5, $ordD[t_1] \leq ordD[t_2] \leq ordD[t_3]$. Since $t_1 \neq t_2$, we have $ordD[t_1] < ordD[t_2] \leq ordD[t_3]$. Therefore, $t_1 < t_3$ and *alternativeEntrance*[*s*] cannot be reset to the same value t_1 again.   □

Theorem 1. *Algorithm* SuperBubble *reports all superbubbles, and only superbubbles, in graph G in decreasing topological order of their exit vertices in $\mathcal{O}(n + m)$ time.*

Proof. Consider an execution of algorithm SuperBubble. Let superbubbles ⟨s_1, t_1⟩, · · · , ⟨s_k, t_k⟩ be the successive superbubbles reported just after the execution of Line 14 of subroutine ReportSuperBubble, where $ordD(t_1) > ordD(t_2) > \cdots > ordD(t_k)$.

1. First, we show that each ⟨s_i, t_i⟩ reported by the algorithm in Line 14 is a superbubble. This is proved by the correctness of Lemma 7.

2. Second, no superbubble is missed out by the algorithm as proved by the following. Subroutine ReportSuperBubble is called for each exit candidate in decreasing order. The entrance candidate for the superbubble (if any) ending at *exit* will only be between *start* and *exit*, where *start* is either the head of the candidates list (when subroutine ReportSuperBubble is called from algorithm SuperBubble) or next candidate of the entrance of an outer superbubble (when called through a recursive call to identify a nested superbubble). A call to subroutine ReportSuperBubble(*start, exit*) checks the possible entrance candidates between *start* and *exit*, starting with the nearest previous entrance candidate (to *exit*). Subroutine ValidateSuperBubble either successfully validates an entrance candidate, or returns a "−1", or returns an alternative entrance candidate. From Lemma 8, there cannot be any valid entrance between this alternative entrance and *exit*. If this alternative entrance starts a sequence of entrances already checked for some exit candidate previously (as depicted by *alternativeEntrance*), then all entrances of that sequence will be skipped, otherwise this alternative entrance will be tested. However, as mentioned in Section 4.1, none of the entrance candidates in the skipped sequence can be valid. Therefore, for each exit candidate, every potential entrance candidate is checked for validity, and those which are not considered are not valid.

3. Third, the running time of SuperBubble is $\mathcal{O}(n + m)$. Indeed, the running time of the TopologicalSort and computing the candidates list is $\mathcal{O}(n + m)$. Furthermore, all list operations cost constant time each, and sum up to a linear cost of $\mathcal{O}(n)$, as there are at most $2n$ candidates in the list. Finally, each call for subroutine ValidateSuperBubble costs $\mathcal{O}(1)$. The total number of times ValidateSuperBubble is called is $\mathcal{O}(n + m)$. This is because subroutine ValidateSuperBubble is called once for each exit candidate in Line 7 of subroutine ReportSuperBubble, and the total number of such calls is bounded by $\mathcal{O}(n)$. Additionally, it is called every time a new *alternativeEntrance* sequence is generated by subroutine ValidateSuperBubble. It follows from Lemma 9 that once an *alternativeEntrance* sequence is reset, it cannot be generated again by subsequent calls to subroutine ValidateSuperBubble. This resetting of *alternativeEntrance* for each entrance candidate (Line 10) thus enables avoiding repeated checks of the same sequences of entrance candidates. Resetting is done every time an edge is considered for the first time between a vertex (in between an entrance candidate *startVertex* and an exit candidate *endVertex*) and its topologically furthest parent (whose order is less than that of *startVertex*). Thus, the total number of times *alternativeEntrance* will be reset (for all the entrance candidates) is bounded by $\mathcal{O}(m)$. Therefore, the total running time for reporting all superbubbles in graph G is $\mathcal{O}(n + m)$.   □

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 vertices and m is the number of edges, improving the best-known $\mathcal{O}(m \log m)$-time algorithm for this problem [10]. It is also interesting to note that in this type of graph, that is, constructed from sequences over a fixed-sized alphabet, the out-degree of each vertex is bounded by the size of the alphabet (four for DNA alphabet); therefore, the time complexity of the proposed algorithm is essentially linear in n.

Our immediate goal is to practically evaluate our algorithm and compare its implementation to 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. Klenerman, C. Barnes, M. Osborne, Patent US20077232656, 2007.

[4] S. Batzoglou, Algorithmic challenges in mammalian genome sequence assembly, in: Encyclopedia of Genomics, Proteomics and Bioinformatics, John Wiley and Sons, Hoboken, New Jersey, 2005.

[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 Res. 18 (5) (2008) 810–820.

[6] P.A. Pevzner, H. Tang, M.S. Waterman, An Eulerian path approach to DNA fragment assembly, Proc. Natl. Acad. Sci. USA 98 (17) (2001) 9748–9753.

[7] N.G. de Bruijn, A combinatorial problem, K. Ned. Akad. Wet. 49 (1946) 758–764.

[8] D.R. Zerbino, E. Birney, Velvet: algorithms for de novo short read assembly using de Bruijn graphs, Genome Res. 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 Inform. 6 (2) (1976) 171–185.

[13] D. Harel, R. Tarjan, Fast algorithms for finding nearest common ancestors, SIAM J. Comput. 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, Proceedings of the 17th Annual Symposium, CPM 2006, Barcelona, Spain, July 5–7, 2006, in: Lecture Notes in Computer Science, vol. 4009, Springer, 2006, pp. 36–48.

[15] S. Durocher, A simple linear-space data structure for constant–time range minimum query, in: A. Brodnik, A. López-Ortiz, V. Raman, A. Viola (Eds.), Space-Efficient Data Structures, Streams, and Algorithms, in: Lecture Notes in Computer Science, vol. 8066, 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, J. Comput. Biol. 20 (10) (2013) 714–737.