Detecting Superbubbles in Assembly Graphs

Taku Onodera[1,a)] Kunihiko Sadakane[2,b)] Tetsuo Shibuya[1,c)]

Abstract: We introduce a new concept of a subgraph class called a superbubble for analyzing assembly graphs, and propose an efficient algorithm for detecting it. Most assembly algorithms utilize assembly graphs like the de Bruijn graph or the overlap graph constructed from reads. From these graphs, many assembly algorithms first detect simple local graph structures (motifs), such as tips and bubbles, mainly to find sequencing errors. These motifs are easy to detect, but they are sometimes too simple to deal with more complex errors. The superbubble is an extension of the bubble, which is also important for analyzing assembly graphs. Though superbubbles are much more complex than ordinary bubbles, we show that they can be efficiently enumerated. We propose an average-case linear time algorithm (*i.e.*, $O(n + m)$ for a graph with n vertices and m edges) for graphs with a reasonable model, though the worst-case time complexity of our algorithm is quadratic (*i.e.*, $O(n(n + m))$). Moreover, the algorithm is practically very fast: Our experiments show that our algorithm runs in reasonable time with a single CPU core even against a very large graph of a whole human genome.

1. Introduction

The sequencing technologies have evolved dramatically in the past 25 years, and nowadays many next-generation sequencers (NGSs) can sequence a human genome-size genome in only a few hours with very small costs. But still there is no sequencing technology that can sequence the entire genome at a time without breaking the genome into millions or billions of short reads. Thus assembling these reads into a whole genome has been one of the most important computational problems in molecular biology, and quite a few algorithms have been proposed for the problem [5], [9], [14] despite the computational difficulty of the problem [10].

Most assembly algorithms construct some graph in their first stage. They are categorized into two types depending on the types of the graph. Many old-time assemblers utilize a graph called the *overlap graph*, in which a vertex corresponds to a read and an edge corresponds to a pair of reads that have an enough-length overlap [1], [3], [11]. More recent algorithms often utilize a graph called the *de Bruijn graph*, in which an edge corresponds to a k-mer that exists in reads and a vertex corresponds to the shared $(k - 1)$-mer between the adjacent k-mers [4], [6], [8], [13], [15], [16], [17]. The de Bruijn graph is said to be more suitable for NGS short reads of large depth.

The next step of most sequencing algorithms after constructed the graph is to simplify the obtained graph by decomposing a maximal unbranched sequence of edges (which is called a *unipath*) into one single edge [4], [8], [15] (Fig. 1). The obtained graph is called a *unipath graph*. After obtained the unipath graph, many sequencing algorithms next detect simple typical motif structures caused by errors to detect errors: The most common motifs are tips, bubbles, and cross links [4], [6], [15], [17] (Fig. 2).

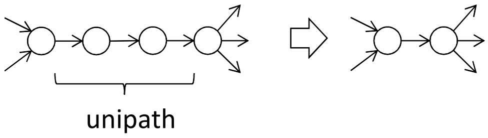
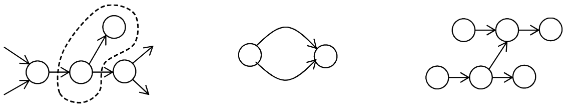


Fig. 1 Construction of a unipath graph.



(1) A tip (2) A bubble (3) A cross link

Fig. 2 Assembly graph simple motifs.

A tip (Fig. 2 (1)) is a low-frequency edge whose end (or start) vertex has no outgoing (resp. incoming) edges, which goes out from (resp. comes into) a high-frequency vertex[*1]. This motif often appears in case there are some error(s) around the end of a read. A bubble (Fig. 2 (2)) consists of multiple edges (with the same direction) between a pair of vertices, which is often caused by error(s) somewhere in the middle of a read. A cross link (Fig. 2 (3)) is a low-frequency edge that lies between high-frequency vertices. This appears when a substring of a read accidentally becomes (by error) the same substring that appears in a different region. All of these motifs are easy to find (obviously in linear time) due to their simplicity.

But we should consider much more complex structures if

[1] Human Genome Center, Institute of Medical Science, University of Tokyo 4-6-1 Shirokanedai, Minato-ku, Tokyo 108-8639, Japan.

[2] National Institute of Informatics, 2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan.

a) tk-ono@hgc.jp

b) sada@nii.ac.jp

c) tshibuya@hgc.jp

[*1] We say 'low/high'-frequency vertices/edges for vertices/edges that correspond to few/many reads.

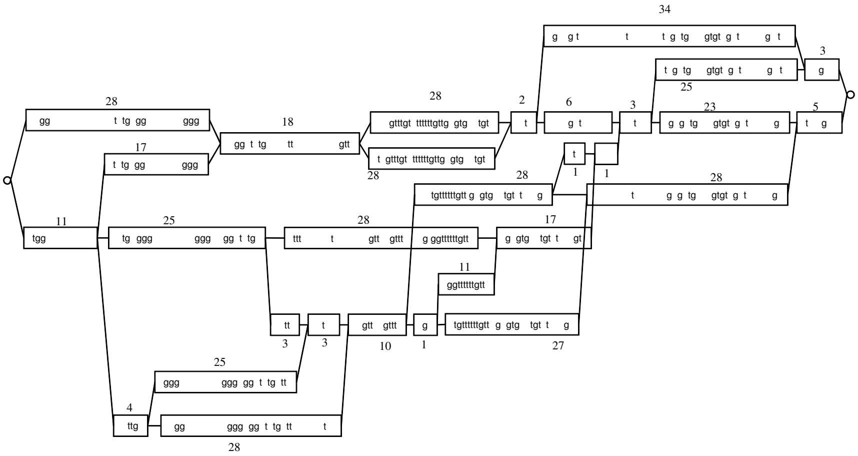


Fig. 3 A superbubble: A very complicated structure caused by errors or repeats. All the edges are labeled with sequences (vertices are not shown). The gaps in the labels are inserted manually in the figure to show alignment between edge labels that start at different offsets from the entrance of the superbubble.

input reads are erroneous (as in the case of the third generation sequencers), have many repeats (as in many large-scale genomes/meta-genomes), or have many mutations (as in cancer genomes). Fig. 3 shows an example of a subgraph of a unipath graph obtained from actual whole human genome reads (the same set of reads used in the experiments in section 4). In this subgraph, paths from the leftmost vertex branch to many paths but they converge into the rightmost single vertex in the end, and there are no cycles in this subgraph, *i.e.*, the subgraph forms a directed acyclic graph (DAG). The vertices between the leftmost vertex and the rightmost vertex has no outgoing/incoming edges to/from external vertices (*i.e.*, vertices not in this subgraph). An important point is that all the paths have similar labels with similar lengths.*2 We call this kind of a subgraph a *superbubble*, as it can be considered as an extension of an ordinary simple bubble (more detailed definition of superbubbles will be given in section 2). Superbubbles are complicated, but it is apparent that many of them are formed as a result of errors, inexact repeats, diploid/polyploid genomes, or frequent mutations. Thus detection of superbubbles should be very important, and it should be useful if we can detect them efficiently. For example, further time-consuming complicated algorithms (e.g., optimal alignment, paired-end read analyses, etc) are applicable against the superbubbles, even if they are too complicated to use against the entire graph.

In the followings, we will give detailed definition of the superbubbles in section 2, and show an efficient algorithm for finding superbubbles in section 3. We will show that the algorithm runs in average-case linear time against graphs with a reasonable model, though the worst-case time complexity is quadratic. In section 4, we will show that the superbubbles can be efficiently enumerated in reasonable time with a small machine, through large-scale experiments against reads from a whole human genome.

2. Preliminaries

2.1 Superbubble

Here, we formally define superbubbles and show some properties of them which are necessary in the rest of the paper.

Definition 1. *Let $G = (V, E)$ be a directed graph. If an ordered pair of distinct vertices (s, t) satisfies the following:*

reachability *t is reachable from s;*

matching *the set of vertices reachable from s without passing**3 *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 in the above condition;*

minimality *no vertex in U other than t forms a pair with s that satisfies the conditions above,*

then we say that the subgraph in the description of the acyclicity condition is a ***superbubble*** *and s, t and $U \setminus \{s, t\}$ are this superbubble's* ***entrance***, ***exit*** *and* ***interior*** *respectively. For any pair of vertices (s, t) that satisfies the above conditions, we denote the superbubble as $\langle s, t \rangle$.*

To take full advantage of the notation $\langle s, t \rangle$, we first need to confirm that if $(s_1, t_1) \neq (s_2, t_2)$ then $\langle s_1, t_1 \rangle \neq \langle s_2, t_2 \rangle$. The following remark ensures it.

Remark 1. *There is a one-to-one correspondence between the vertex pairs satisfying the conditions in Definition 1 and superbubbles.*

Proof. Because of the acyclicity condition, the vertices of a superbubble can be topologically sorted, *i.e.*, they can be ordered in such a way that if v is reachable from u then $u < v$. Due to the matching condition, s (resp. t) is the minimum (resp. maximum) ordered vertex. □

Now we observe a proposition which clarifies the situation and motivates linear time enumeration of superbubbles.

Proposition 1. *Any vertex can be the entrance (resp. exit) of at*

*2 The experiments in section 4 will show that the path label lengths of a superbubble are only at most 5% different in more than 85% of the detected superbubbles.

*3 Passing through a vertex means that visiting and then leaving it, not just visiting or leaving alone.

most one superbubble.

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

Proof. We prove the proposition by *reductio ad absurdum*. Suppose $\langle s, t_1 \rangle$ and $\langle s, t_2 \rangle$ are distinct superbubbles. If t_2 is a vertex in $\langle s, t_1 \rangle$, then t_2 is in the interior of $\langle s, t_1 \rangle$ but this contradicts to the minimality condition for $\langle s, t_1 \rangle$. Similarly, t_1 being a vertex in $\langle s, t_2 \rangle$ also results in a contradiction.

Suppose, on the other hand, that t_2 is not a vertex in $\langle s, t_1 \rangle$. There is a path from s to t_2. By removing cycles from t_2 to t_2 if necessary, this path can be taken in such a way that t_2 appears only at the last step and at this time, all vertices in the path are in $\langle s, t_2 \rangle$. On the other hand, the vertex just before the first vertex on the path that is not in $\langle s, t_1 \rangle$ is t_1. In particular this means that t_1 is in $\langle s, t_2 \rangle$ but this leads to contradiction by the first half of the argument. □

Corollary 1. *There are $O(n)$ superbubbles in a graph with n vertices.*

Before closing this subsection, let us point out, without proof, yet another property of superbubbles that is not directly necessary for this work but worth mentioning to grasp the picture.

Claim 1. *If two distinct superbubbles share a vertex, either one's exit is the other's entrance or one is included in the other's interior.*

2.2 Construction of a Unipath Graph

Given a set $\mathcal{R}$ of reads, we first construct the de Bruijn graph [13]. Let $T = T[1, m]$ be a read of length m in $\mathcal{R}$. The k-mers of T are length-k substrings of T, that is, $T[i, i + k - 1]$ for $i = 1, 2, \ldots, m - k + 1$. Let K denote the multiset of k-mers of all reads in $\mathcal{R}$, and K_d denote the set of (distinct) k-mers that appear at least d times in K. A k-mer in K_d is called a *solid* k-mer.

The de Bruijn graph $G = (V, E)$ of $\mathcal{R}$ is defined as follows. The vertex set V is the set of $(k-1)$-mers defined as $V = \{T[1, k-1] \mid T[1, k] \in K_d\} \cup \{T[2, k] \mid T[1, k] \in K_d\}$. The edge set E is defined as $\{(u, v) \mid \exists T[1, k] \in K_d, u = T[1, k-1], v = T[2, k]\}$. The edge label of (u, v) is $T[k]$ if $u = T[1, k-1], v = T[2, k]$. Typical values of k and d are $k = 28$, $d = 3$.

We use the succinct de Bruijn graph [2], which is a compressed representation of the de Bruijn graph of $\mathcal{R}$. For a set of m solid k-mers, the succinct de Bruijn graph uses $4m + o(m)$ bits to encode the graph, and supports the following operations.

- *outdeg*(v)/*indeg*(v) returns the number of outgoing/incoming edges from/to vertex v in $O(1)$ time, respectively.
- *outgoing*(v, c) returns the vertex w pointed to by the outgoing edge of vertex v with edge label c in $O(1)$ time. If no such vertex exists, it returns -1.
- *incoming*(v, c) returns the vertex $w = T[1, k-1]$ such that there is an edge from w and v and $T[1] = c$ in $O(k)$ time. If no such vertex exists, it returns -1.

From a de Bruijn graph $G = (V, E)$, we construct a unipath graph $G' = (V', E')$ as follows. The vertex set V' is a subset of V such that any vertex in V' has more than one outgoing edges or more than one incoming edges. The edge set E' is the multiset of all pairs (u, v) such that $u, v \in V'$ and there is a path $u, x_1, x_2, \ldots, x_\ell, v$ in G and outdegrees and indegrees of $x_1, x_2, \ldots, x_\ell$ are all one. The edge label of (u, v) is the concatenation of edge labels of $(u, x_1), (x_1, x_2), \ldots, (x_{\ell-1}, x_\ell), (x_\ell, v)$ in G. The length of the edge label is $\ell + 1$.

In addition to the data structure of the succinct de Bruijn graph, we use a bit vector $B[1, m]$ where $m = |E|$ is the number of edges in G to represent the unipath graph G'. We set $B[v] = 1$ if and only if the vertex v of G is also a vertex of G'. The outdegree and the indegree of v in G' is equal to those of v in G. To find the vertex *outgoing*(v, c) in G', we first compute $w =$ *outgoing*(v, c) in G. Then we repeatedly traverse the unique outgoing edge of w until $B[w] = 1$. The resulting vertex is the answer. The unipath graph is constructed in linear time from the succinct de Bruijn graph because each of the *outdeg*, *indeg*, and *outgoing* operations takes constant time. Figure 4 shows an example.

3. Algorithm

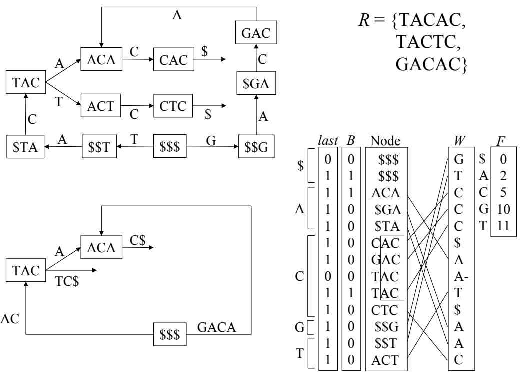
Here, we explain how to enumerate all superbubbles in a given graph. As we have seen in subsection 2.1, each vertex can be the entrance of at most one superbubble. Therefore, once we have a way to check if a vertex s has another vertex t s.t. (s, t) is the entrance/exit pair, then we can find all superbubbles just by iterating this procedure for all $s \in V$. Below, we focus our attention on this reduced problem.

3.1 Description

The algorithm is based on the standard topological sorting. It takes a directed graph $G = (V, E)$ and $s \in V$ as inputs, and returns $t \in V$ s.t. (s, t) is an entrance/exit pair of a superbubble if any. It proceeds by visiting vertices one by one maintaining the dynamic set S of vertices it can visit the next time. Initially, S is set to be $\{s\}$. It also maintains a label for each vertex. The label visited means that the vertex has already been visited. The label seen means that the vertex has at least one visited parent . At each step, the algorithm picks out an arbitrary vertex v from S labeling it as visited and label each child as seen. If all the parents of a child are visited, it pushes the child into S. In visiting vertices, the algorithm aborts anytime when it finds a vertex with no child, which means a tip, or a parent of s, which means a cycle because any vertex visited is a descendent of s. After visiting a vertex, the algorithm tests if it is going to visit the exit at the next step as follows. First it checks if S consists of one vertex, say t, and no vertex other than t is labelled as seen. If not, the test is negative. Otherwise, the algorithm further checks if the edge (t, s) exists or not. If it does, the algorithm aborts because it just found a path from s to s, a cycle. Otherwise, the algorithm returns t. The algorithm aborts if S runs out.

3.2 Correctness

A vertex can be pushed into S at most once because it happens when all its parents are visited and once visited a vertex never cease to being so. Thus, the algorithm can pick out a vertex from S at most n times and in particular it halts. Below, we prove the correctness of the returned value, which reduces to the followings: a) if the input vertex is the entrance of some superbubble,



	last	*B*	Node	*W*
$	0	0	$$$	G
	1	1	$$$	T
A	1	1	ACA	C
	1	0	$GA	C
	1	0	$TA	C
C	1	0	CAC	$
	1	0	GAC	A
	0	0	TAC	A-
	1	1	TAC	T
	1	0	CTC	$
G	1	0	$$G	A
T	1	0	$$T	A
	1	0	ACT	C

	F
$	0
A	2
C	5
G	10
T	11

Fig. 4 Top right: The input set $\mathcal{R}$, top left: The de Bruijn graph of $\mathcal{R}$ with $k = 3, d = 1$, bottom left: the unipath graph, bottom right: the succinct de Bruijn graph and the unipath graph. Non-branching nodes are removed. We store only *last*, *B*, *W* and *F*.

```
Require: directed graph $G = (V, E)$, $s \in V$
Ensure: returns $t$ s.t. $(s, t)$ is an entrance/exit pair of a superbubble if any
 1: push $s$ into $S$
 2: repeat
 3:     pick out an arbitrary $v \in S$
 4:     label $v$ as visited
 5:     if $v$ does not have a child then
 6:         abort // tip
 7:     for $u$ in $v$'s children do
 8:         if $u = s$ then
 9:             abort // cycle including $s$
10:         label $u$ as seen
11:         if all of $u$'s parents are visited then
12:             push $u$ into $S$
13:     if only one vertex $t$ is left in $S$ and no other vertex is seen then
14:         if edge $(t, s)$ does not exist then
15:             return $t$
16:         else
17:             abort // cycle including $s$
18: until $|S| = 0$
```

Fig. 5 Pseudocode of an algorithm to find the corresponding exit of an potential entrance

then the algorithm returns the corresponding exit; b) if the algorithm returns a vertex, it is the exit of a superbubble and the input vertex is the corresponding entrance.

First, we observe an invariant. Let V_{seen} be the set of vertices labelled as seen and V_{visited} be the set of vertices labelled as visited. Let V_{to} be the set of vertices that are reachable from s without passing through any element of V_{seen} and let V_{from} be the set of vertices from which at least one element of $V_{\text{visited}} \cup S$ can be reachable without passing through s.

Lemma 1. *After the algorithm visits a vertex, i.e., after the line 12 of the pseudocode in Figure 5 is executed,* $V_{\text{to}} = V_{\text{visited}} \cup V_{\text{seen}}$ *and* $V_{\text{from}} = V_{\text{visited}} \cup S$. *In particular, if the algorithm returns t, then* (s, t) *satisfies the matching condition.*

Proof. We prove the first half by mathematical induction. After the first visit, V_{visited}, V_{seen} and S consist of s, s's children and s's children with indegree 1 respectively and the lemma holds. Suppose the lemma holds up to the visit to some vertex. During the visit to the next vertex, say v,

(1) v is removed from S and its label is changed from seen to visited;
(2) all children of v are labelled as seen;
(3) the children of v whose parents are all visited are added to S.

Consequently, both V_{to} and $V_{\text{visited}} \cup V_{\text{seen}}$ acquire the vertices reachable from v without passing through any element of V_{seen}, i.e., the children of v. Therefore, $V_{\text{to}} = V_{\text{visited}} \cup V_{\text{seen}}$ still holds. On the other hand, $V_{\text{visited}} \cup S$ acquires the vertices newly added to S, i.e., the children of v whose parents are all labelled as visited. Now these vertices are also in V_{from} because $V_{\text{from}} \supseteq V_{\text{visited}} \cup S$ by definition. Furthermore, they are the only vertices V_{from} acquires because the parents of them were already in V_{from} after the previous visit by the induction hypothesis. Therefore, $V_{\text{from}} = V_{\text{visited}} \cup S$ also stays true.

Next, we prove the last half. If the algorithm returns t, after the last visit, $V_{\text{to}} = V_{\text{from}}$ because $S = V_{\text{seen}}$ due to the first half. On the other hand, at this time, V_{to} consists of the vertices reachable from s without passing through t because $V_{\text{seen}} = \{t\}$. Therefore, it suffices to show that V_{from} consists of the vertices from which t is reachable without passing through s. This is true because after every visit, from any vertex in V_{visited} at least one vertex in V_{seen} is reachable without passing through s, a fact which can be proven easily by mathematical induction again. □

Next, we prove a). Let t be the exit corresponding to s. Because of the matching condition of (s, t), the algorithm never aborts due to a tip or running out of S at least up to the point when t is pushed into S, no matter if t is pushed into S at all. Similarly, the algorithm never aborts due to a cycle up to the same point because of the acyclicity condition of (s, t). On the other hand, if t is indeed pushed into S, then t must be the only vertex seen and all other vertices of $\langle s, t\rangle$ must be visited due to the matching condition of (s, t) and the lemma. Therefore, the only possibilities left are that the algorithm outputs t or some other vertex in $\langle s, t\rangle$. But the second case never happens because a vertex, say v, other than t in $\langle s, t\rangle$ is output, then the pair (s, v) satisfies the reachability, matching (due to the lemma) and acyclicity conditions, which contradicts to the minimality condition of (s, t).

Last, we prove b). Suppose the algorithm returns a vertex t. Obviously, t is reachable from s. The matching condition holds because of the lemma. The alleged superbubble does not contain cycles including s because otherwise the algorithm must have aborted. And it does not contain cycles not including s because otherwise the first vertex visited in the cycle has a parent in the cycle. This means the parent has been visited earlier, which contradicts the way the child was chosen. Thus, the acyclicity condition holds. The minimality condition holds because otherwise, there is a vertex v s.t. (s, v) is an entrance/exit pair and because of a) the algorithm must have returned v, instead of t.

3.3 Analysis

In the worst case, each execution of the algorithm takes $\Theta(n + m)$-time and in total the calculation of all superbubbles takes $\Theta(n(n + m))$-time. Below, we show that, under a reasonable model, the algorithm takes constant time on average and thus all superbubbles can be found in $\Theta(n)$-time in total.

As we will see in the next section, although there are tens of thousands of superbubbles in practical unipath graphs, the entire graph is so large that its size is orders of magnitude greater than the total size of superbubbles. Thus, most of the time spent in the iterated executions of the algorithm is dedicated for traversing regions that are far away from any superbubbles. Therefore, it is reasonable to reduce the analysis of the algorithm to the evaluation of the time spent until the traversal of a non-superbubble region is aborted. In such a case, if a vertex is not pushed into S when it is labelled as seen, then it is very unlikely to be visited afterwards. In other words, once the algorithm comes across a vertex of indegree greater than 1, then it almost never proceeds to traverse its descendants. With these observations in mind, we model the way the tree of visited vertices grows in the algorithm by the following probabilistic tree generation process. It starts from the root. Each vertex is good with probability p. A good vertex corresponds to a vertex of indegree 1. If a vertex is good, it spawns i children with probability p_i. The theory of Galton-Watson branching processes [7] tells that the expected number of vertices of depth i is $\Theta(r^i)$ where $r := p \sum_i ip_i$, i.e., the expected number of children of each vertex. Therefore, if $r < 1$ the expected size of the tree is $\Theta(\frac{1}{1-r})$, a constant. For the unipath graph we constructed from human genome data, r was about 0.77 where p and p_i were determined as the proportion of vertices with particular in/out-degree within all vertices.

4. Experiment

4.1 Procedures

We first constructed the succinct de Bruijn graph with parameter $k = 27$ and $d = 3$ for the read set SRX016231, which was derived by sequencing a human individual by an Illumina sequencer. The length of each read is 100bp and the coverage is about 40. Next, we constructed the unipath graph as described in subsection 2.2. The resulting unipath graph consists of 107,154,751 vertices and 210,207,840 edges. Last, we found all superbubbles in the unipath graph by the algorithm in section 3.

4.2 Results

Table 1 is the histogram of the size of superbubbles where the size of a superbubble means the number of vertices in it. The superbubbles of size 2 are omitted because they are ordinary bubbles. The superbubble of Fig. 3 is of size 20 and this histogram tells, among other things, that there are hundreds of equally or more complex superbubbles. On the other hand, what matters the most for the application to genome assembly problem is whether superbubbles really capture erroneous or repeat/mutation abundant regions, which topological complexity alone does not necessarily suggest. One way to assess the relevance of a superbubble in this regard is to compare the length of paths in it where length of an edge is the length of the sequence represented by the edge. Note that topologically close paths can have a variety of lengths because each edge can be originated from a unipath. But among 23,078 superbubbles of size equal to or greater than 5 we found, 19,926 (86.3%) of them have the longest/shortest path length ratio smaller than 1.05. Therefore, superbubbles like that of Fig. 3 are indeed typical.

Table 1 Histogram of the size of superbubbles

size	3-9	10-19	20-29	30-39	40-49	50-59	60-
#S.B.	71663	4295	347	69	21	8	3

In terms of the computation time, it took 742.1 seconds for a Xeon 3.0GHz CPU to enumerate all superbubbles including ordinary bubbles. The number of vertices visited was 126,537,254.

5. Concluding Remarks

We introduced the concept of superbubbles in assembly graphs, and proposed an efficient algorithm for detecting them. But many tasks remain as future work. It is an open problem whether it is possible to detect superbubbles in worst-case linear time. Developing methods for categorizing the detected superbubbles (e.g., errors, repeats, mutations, and polyploids), and methods for fixing errors in superbubbles are important future tasks. It is also interesting to extend our algorithm for other bubble-like structures (e.g. the bulge structure [12]).

Acknowledgments

KS and TS are supported in part by KAKENHI 23240002. This research was supported by JST, ERATO, Kawarabayashi Large Graph Project. The super-computing resource was provided in part by Human Genome Center, the Institute of Medical Science, the University of Tokyo.

References

[1] S. Batzoglou, D. B. Jaffe, K. Stanley, J. Butler, S. Gnerre, E. Mauceli, B. Berger, J. P. Mesirov, and E. S. Lander. Arachne: a whole-genome shotgun assembler. *Genome Research*, 12:177–189, 2002.

[2] A. Bowe, T. Onodera, K. Sadakane, and T. Shibuya. Succinct de bruijn graphs. *Proc. 12th Workshop on Algorithms in Bioinformatics (WABI)*, pages 225–235, 2012.

[3] X. Huang and S. P. Yang. Generating a genome assembly with pcap. *Current Protocols in Bioinformatics*, Unit 11.3, 2005.

[4] B. Jackson, M. Regennitter, X. Yang, P.S. Schnable, and S. Aluru. Parallel de novo assembly of large genomes from high-throughput short reads. *Proc. 24th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–10, 2010.

[5] M. Kasahara and S. Morishita. *Large-Scale Genome Sequence Processing*. Imperial College Press, 2006.

[6] R. Li, H. Zhu, J. Ruan, W. Qjan, X. Fang, Z. Shi, Y. Li, S. Li, G. Shan, K. Kristiansen, H Yang, and J. Wang. De novo assembly of human genomes with massively parallel short read sequencing. *Genome Research*, 20:265–272, 2010.

[7] R. Lyons and Y. Peres. *Probability on Trees and Networks*. Cambridge University Press., 2012. In preparation. Current version available at `http://mypage.iu.edu/~rdlyons/`.

[8] I. MacCallum, D. Przybylski, S. Gnerre, J. Burton, I. Shlyakhter,

A. Gnirke, J. Malek, K. McKernan, S. Ranade, T. P. Shea, L. Williams, S. Young, C. Nusbaum, and D. B. Jaffe. Allpaths 2: small genomes assembled accurately and with high continuity from short paired reads. *Genome Biology*, 10(R103), 2009.

[9] J. R. Miller, S. Koren, and G. Sutton. Assembly algorithms for next-generation sequencing data. *Genomics*, 95:315–327, 2010.

[10] E. W. Myers. Toward simplifying and accurately formulating fragment assembly. *Journal of Comutational Biology*, 2:275–290, 1995.

[11] E. W. Myers, G. G. Sutton, A. L. Delcher, I. M. Dew, D. P. Fasulo, M. J. Flanigan, S. A. Kravitz, C. M. Mobarry, K. H. J. Reinert, K. A. Remington, E. L. Anson, R. A. Bolanos, H. Chou, C. M. Jordan, A. L. Halpern, S. Lonardi, E. M. Beasley, R. C. Brandon, L. Chen, P. J. Dunn, Z. Lai, Y. Liang, D. R. Nusskern, M. Zhan, Q. Zhang, X. Zheng, G. M. Rubin, M. D. Adams, and J. C. Venter. A whole-genome assembly of drosophila. *Science*, 287:2196–2204, 2000.

[12] S. Nurk, A. Bankevich, D. Antipov, A. Gurevich, A. Korobeynikov, A. Lapidus, A. Prjibelsky, A. Pyshkin, A. Sirotkin, Y. Sirotkin, R. Stepanauskas, J. McLean, R. Laskin, R. Stepanasukas, J. McLean, R. Laskin, S. R. Clingenpeel, T. Woyke, G. Tesler, M. A. Alekseyev, and P. A. Pevzner. Assembling genomes and mini-metagenomes from highly chimeric reads. *Proc. 17th International Conference on Research in Computational Molecular Biology (RECOMB 2013)*, pages 158–170, 2013.

[13] P. A. Pevzner, H. Tang, and M. S. Waterman. An eulerian path approach to dna fragment assembly. *Proceedings of the National Academy of Sciences*, 98:9748–9753, 2001.

[14] M. Pop. Genome assembly reborn: recent computational challenges. *Briefings in Bioinformatics*, 10(4):354–366, 2009.

[15] M. Sahli and T. Shibuya. Arapan-s: a fast and highly accurate whole-genome assembly software for viruses and small genomes. *BMC Research Notes*, 5(243), 2012.

[16] J. T. Simpson, K. Wong, S. D. Jackman, J. E. Schein, and S. J. Jones. Abyss: a parallel assembler for short read sequence data. *Genome Research*, 19:1117–1123, 2009.

[17] D. R. Zerbino and E. Birney. Velvet: algorithms for de novo short read assembly using de bruijn graphs. *Genome Research*, 18:821–829, 2008.