Internal Report CS Bioinformatics Track 14-03 June 2014

Leiden University

Computer Science

Bioinformatics Track

Probubble: variant calling through the alignment of overlap based de-novo assembly graphs

Jasper Linthorst

MASTER'S THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

TU Delft MSc Thesis

June 3rd 2014
Pages 1– 12

Probubble: variant calling through the alignment of overlap-based de-novo assembly graphs

Jasper Linthorst

Delft Bioinformatics Lab*, Delft University of Technology, Delft, The Netherlands

Defended on June 3rd 2014

ABSTRACT

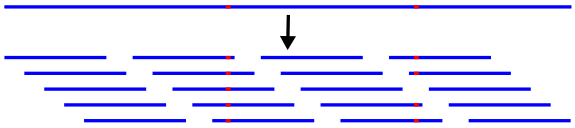
The comparison of high-throughput sequencing data through an imperfect reference genome is limiting the full potential of many sequencing studies. Therefore, the ability to detect high quality genetic variants without the intervention of a reference genome could cause a paradigm shift in the way in which many sequencing studies are currently interpreted. Different solutions are already available, but all have their own limitations. They either don't scale up to larger numbers of samples or are structurally restricted by the application of a de Bruijn graph. Here we present Probubble, an algorithm that is aimed at detecting variations between multiple samples by aligning overlap-based assembly graphs. On a simulated dataset, it is shown that the method works better than the de Bruijn graph based alternative. Furthermore, it is shown that in case of a highly divergent reference genome, the direct comparison of two samples through Probubble also results in improved variant calls over a mapping-based approach.

Contact: jasper.linthorst@gmail.com

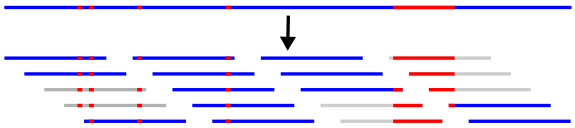
1 INTRODUCTION

Genetic variation for a great deal influences the phenotypical differences between cells and entire organisms [1]. In order to explain these differences, the capacity to detect genetic variation at the level of the actual DNA sequence is of major importance.

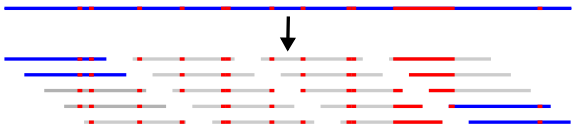
Nowadays, the most common way to detect genetic variation between two or more samples is to align the reads originating from a high throughput sequencing experiment to a reference genome. By allowing a limited amount of mismatches when aligning these reads, variations with respect to a reference genome can then be distinguished from errors by inspecting the pile-up of reads. Such a mapping-based approach might work well in cases where the reference sequence is closely related to the genome of the samples at hand. However, it becomes problematic when there is no reference genome available, or when the divergence from the reference sequence increases. For example, with increasing sequence divergence, structural variations are introduced. These variations can introduce novel genomic sequence to the genome that is absent from, or highly mutated with respect to, the reference sequence. As a consequence, reads that originated from these regions can no longer be properly aligned to the reference genome, with the result that variations in these regions cannot be detected (see Figure 1). In humans, these structural variations make up a considerable amount of the total variation [14]. In [8] and [10] it has also been shown, that large amounts of novel sequence can be discovered when all reads of a whole genome sequencing (WGS) experiment are considered. To counter this problem, mapping-based approaches resort to local realignment and de-novo assembly methods when read mapping fails [6]. However, only the partially mapped reads are considered for this purpose, by which larger variations, spanning unmapped reads are left out.



(a) Schematical representation of obtaining sequence reads from a sample's genome. Variations in the sample's genome with respect to the reference genome are indicated in red.



(b) A sample's genome is more diverged from the reference genome. Certain reads exceed the number of allowed mismatches (2 in this example) for alignment to the reference genome. Mappable reads are indicated in blue, unmappable reads are indicated in grey.



(c) A sample's genome is even more diverged from the reference genome, reads originating from the entire region become unmappable, thus variants cannot be detected in this region.

Fig. 1: Divergence from a reference sequence results in unmappable reads. In a mapping-based approach these reads cannot be used for variant detection.

De-novo reconstruction of genomes can eliminate the need for a mapping-based comparison as it reconstructs the original genome just from the set of sequence reads (i.e. without the use of a reference genome) using only the overlap information between the reads. When multiple samples are assembled simultaneously [8], variations between the samples emerge as bubble structures in the resulting assembly-graphs. In this way, variations between samples can be detected without the need to align individual reads to a reference sequence.

*Delft Bioinformatics Lab: http://bioinformatics.tudelft.nl

Various alternative approaches to variant calling have been introduced over the past years [8, 10, 18] that do not depend on directly aligning reads to a reference genome. In [8] complete multi-sample (or colored) de Bruijn graphs are constructed, which are subsequently analyzed for bubble structures in order to detect variations. In [10] and [18] similar reference-free variant detection methods are implemented (known as contrast assembly in Fermi [10] and graph concordance in SGA [18]), that are based on a pairwise comparison of two read-sets, but using an overlap based de-novo assembly graph.

In [17] (Ch. 5, pg 98) it has been shown that, when compared to a de Bruijn graph, string graphs [12] (based on overlapping reads) yield better results with respect to variant detection in both sensitivity and precision. This follows from the fact that a de Bruijn graph does not reflect an optimal representation of the underlying set of reads, i.e. reads are broken down into k-mers, by which the long-range information that is contained in the reads is lost. String graphs however, can be considered as a lossless and even compressed representation of the individual reads. The efficiency gains associated with the use of a de Bruijn graph allow the use of Cortex to detect variations between more than two samples simultaneously. This is a feature that makes it very useful with respect to studying cohorts of samples, something that is currently not easily possible using the methods as implemented in Fermi and SGA: the creation of true multi-sample string graphs, for larger genomes, quickly exceeds the limits of what is computationally feasible in terms of memory usage. An important underlying reason for that is that the creation of a multi-sample string graph always depends on a single run of the assembler at which point all sequence reads need to be kept in memory. This automatically limits the total number of samples that can simultaneously be assembled into a string graph, which is easily surpassed when considering large genomes and current hardware.

Here we introduce 'Probubble', a method that combines the use of string graphs for variant calling with a scalable method for obtaining multi-sample variation graphs. The proposed method is linear in time and sub-linear in space with respect to the number of samples. This is accomplished by an algorithm that enables the construction of a variation aware graph that integrates multiple single-sample string graphs obtained by individual de-novo assemblies. The difficulty that needs to be overcome, is that in string graphs, unlike de Bruijn graphs, nodes can differ in length and sequence. Since different de Bruijn graphs essentially contain the same set of nodes, the merging process of different de Bruijn graphs, as described by Cortex [8], is much more straightforward: it involves the simple addition of the edges from one graph to the other. In string graphs this problem is more complex and can best be seen as a graph alignment problem [3].

The algorithm presented here essentially performs a graph alignment of two string graphs while allowing mutations to the structure of the input graphs. Exact matches between the sequence defined on the nodes in the graphs is used for this purpose. The algorithm is divided into three main steps: preprocessing of the input graphs, followed by a global and then a local merge step. The final variant calling is performed by a bubble detection algorithm which currently only detects 'simple' bubbles. In order to assess the positioning of variants (i.e. with respect to genes), it is possible to merge the graph with a linear reference sequence, so that its coordinate system can be used.

2 METHODS

2.1 String graphs

Within a string graph the nodes in the graph correspond to unitigs [12]. Unitigs are formed by assembling unique unambiguously overlapping sequence reads, such that repetitive sequences within the genome are collapsed into a single node. From this definition it follows that (1) the sequence on every node in the graph cannot be shorter than the length of the shortest read used to generate the string graph, and (2) that every node in the graph defines a unique sequence. Therefore every sequence with a length larger than the longest read that was used to construct the graph must be unique within the graph. The edges in these graphs correspond to overlaps between unitigs. Unitigs can have overlapping sequence on the 5' part (tail) and on the 3' part (head). Since unitigs can originate from either one of the two strands of DNA, 4 possible valid overlaps are distinguished (see Figure 2).

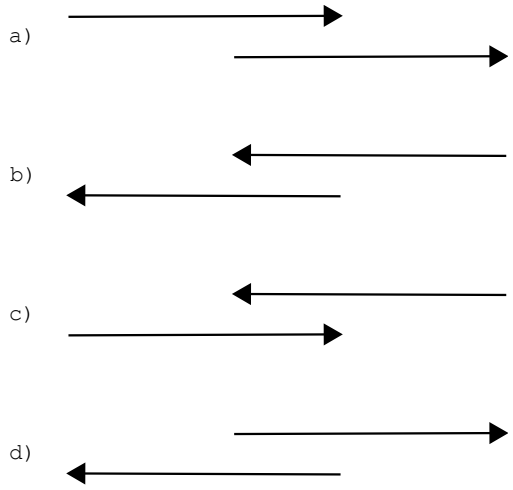
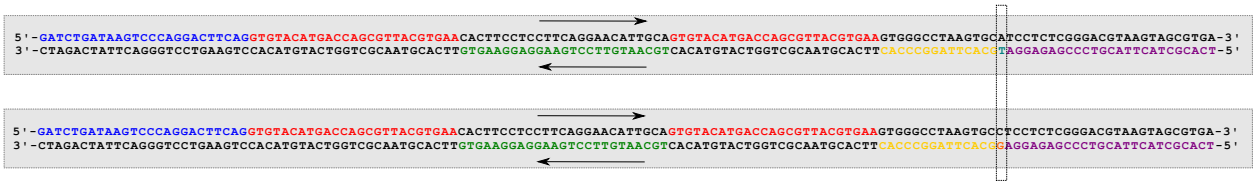


Fig. 2: **a** head-to-tail **b** tail-to-head **c** head-to-head **d** tail-to-tail

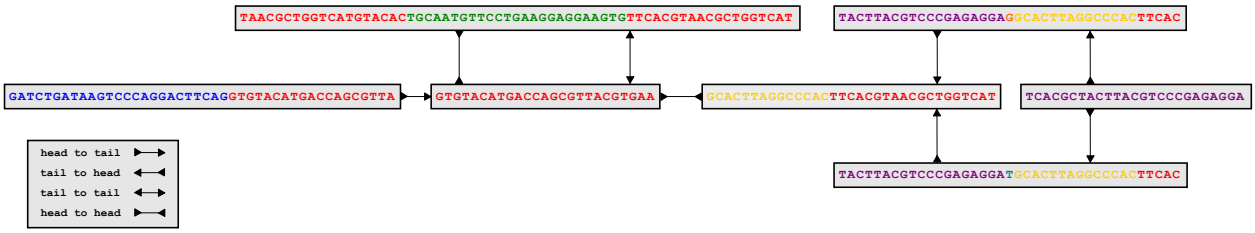
The strand orientation of different unitigs can therefore be derived from the annotations on the edges that connect them. Head-to-tail and tail-to-head overlap (Figure 2a and 2b) can only occur when both unitigs originate from the same strand and head-to-head and tail-to-tail can only occur when unitigs originate from opposite strands (Figure 2c and 2d).

Since all unambiguously overlapping reads have been merged into unitigs, the remaining overlaps form joins and splits within the bidirectional graph. The lengths of the corresponding overlaps are modeled on the edges. An example of a string graph representation built up of reads originating from a simulated diploid sample can be seen in Figure 3.

In order to merge, or rather align, two string graphs, a method for mapping nodes between two graphs is needed. Exact matches between the sequences modeled on these nodes can be used for this purpose. For various reasons (genomic variation between and within samples, variations in read-depth, sequencing errors etc.) exact matches between complete unitigs of two string graphs do not occur



(a) Both strands of two chromosomes of a simulated diploid sample.



(b) String graph representation of 20bp reads originating from the two simulated chromosomes.

Fig. 3: String graph representation of reads originating from a simulated sample. Unitigs are written from 5' to 3' and originate from different strands of the underlying chromosomes. Repetitive sequence is indicated in red, other colors are used to distinguish between different parts of the underlying sequence. A heterozygous SNP generates a 'simple' bubble in the graph.

often. Therefore, it is necessary to find substring matches between unitigs originating from both graphs. Suffix- and LCP arrays (see Appendix 1 and 2) can be used to detect common substrings that occur in both graphs in an efficient and scalable way.

2.2 Variation graphs

Assemblers like SGA and Fermi produce string graph data structures as an intermediate result of the assembly process. String graph generation is based on a parameter $k_{assembly}$, indicating the minimum length of a valid overlap. Transitive edges (or reducable edges) are avoided by combining unambiguous overlapping reads into unitigs. Steps that typically precede the output of a string graph are the generation of indices and error correction. Steps that typically follow after the generation of a string graph, are graph cleaning and scaffolding. In the scaffolding step, the paired-end information, if available, is used to determine a path through the assembly graph, in order to output longer contigs. The graph cleaning step is used to get rid of the abundance of edges that are introduced by errors that could not be corrected for. Throughout the work presented here, cleaned string graphs produced by Fermi were used as an input for Probubble. In order to align (or merge) two string graphs, graphs have to be converted to variation graphs in a number of preprocessing steps.

For every edge in a string graph the overlapping sequence is defined on both the source and the target node (see Figure 4a). This causes a considerable amount of duplicated sequence with respect to the underlying genome, and makes it unclear which part of the underlying genomic sequence is covered by which unitig (see colored sequence in Figure 3). It is important to remove these redundant sequences from the graph, since the alignment algorithm depends on unique exact matches between two graphs. If these sequences would not be removed, no unique matches could be found in the overlapping parts of unitigs, making the process of merging and bubble detection later on much harder.

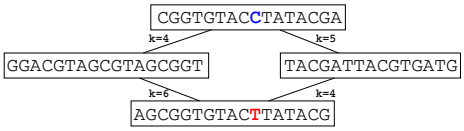
To reduce the amount of redundant sequence in the graph, unitig nodes are broken at the positions where overlapping sequence started and ended (Figure 4b). Next, the resulting unitig sequences can be reconnected so that the paths through the graph eventually spell the same sequences, but without any overlap defined on the edges. All unused 'tip' nodes can then be removed from the graph to get rid of most of the redundant sequence (Figure 4c).

By doing so, the resulting graph is no longer a strict representation of the original string graph, since the overlap information is lost. However, it does still represent all possible paths through the original graph and thus still contains all bubble structures caused by within sample variations.

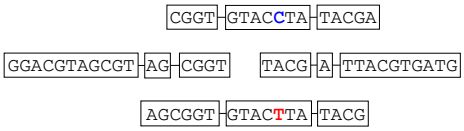
After breaking and reconnecting the nodes in the graph, all neighboring nodes have a similar alignment (the overlap for every edge is the same, namely 0). In order to further reduce the amount of redundant sequence, bubbles and repeat structures are simplified by merging common pre- and suffixes. This is done by adding the sequence defined on all neighboring nodes on one end of a unitig, to a prefix tree (also called a Trie). The leaves of the prefix Tree are then reconnected to the rest of the graph (see Figure 5a to 5e). This is done for every node in the graph until no more paths can be merged. Paths are only merged when the neighbors are not interconnected and there are no other edges leaving the node at the same end.

2.3 Global merge

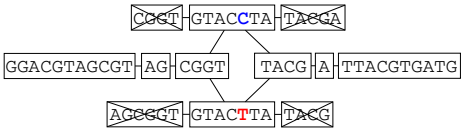
After preprocessing two string graphs, their resulting variation graphs can be merged. The process of merging starts by finding the



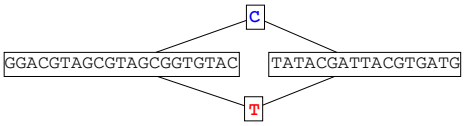
(a) A ‘simple’ bubble structure in a string graph (ignoring strand orientation). Edges between unitigs indicate sequence overlap, a parameter k indicates the number of overlapping base pairs.



(b) Nodes are broken at the start and endpoint of every overlap.



(c) Broken nodes are reconnected corresponding to the overlap that was defined on them before.

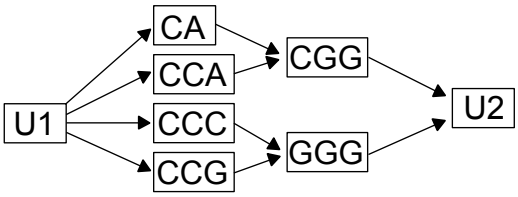


(d) Redundant sequence is further reduced by merging common pre- and suffixes in a prefix tree, see Figure 5a to 5e.

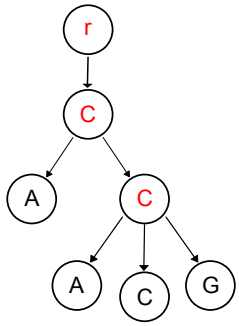
Fig. 4: Preprocessing steps performed on a ‘simple’ bubble structure.

longest common substrings (LCS) between the two graphs, as can be derived from the constructed suffix- and LCP array (see Appendix 1 and 2). During the global merge, all unitigs are compared to each other until no more common substrings can be found that are longer than the predefined threshold k_{global}. Since the size of the matching sequence between the two graphs can be seen as an indication of how unique a match is, matches are processed in order of length. After a longest common substring S is detected the originating nodes are ‘broken’ (see Figure 4b) and the parts corresponding to S are merged (see Figure 6). Subsequently, S and all substrings of S are masked within the suffix array, such that the next longest common substring cannot be a substring of S. The process of masking values is simple and can be achieved in linear time, without recomputing the entire LCP array.

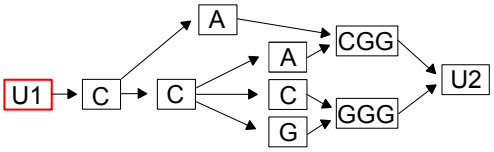
This process is repeated until no more exact matching substrings of a length longer than k_{global} exist within the two graphs. Since in a string graph all repetitive sequences longer than the read length are collapsed into unitigs, we know that every sequence longer than the length of the underlying reads has to be unique within the graph. Therefore, the value of k_{global} should be picked in such a way that it is larger than the largest read length of the underlying



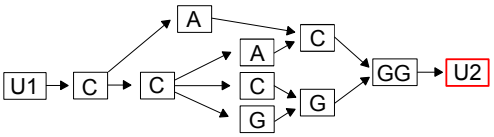
(a) Bubble structure with redundant sequence enclosed between Unitig1 (U1) and Unitig 2 (U2).



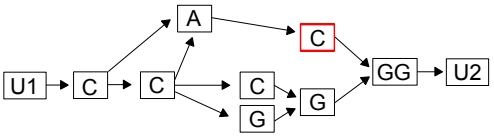
(b) The 3’ neighbors of U1 (‘CA’,‘CCA’,‘CCC’,‘CCG’) are added to a prefix tree. Nodes merging paths are indicated in red.



(c) Bubble structure obtained by reconnecting the leaves of the prefix tree.



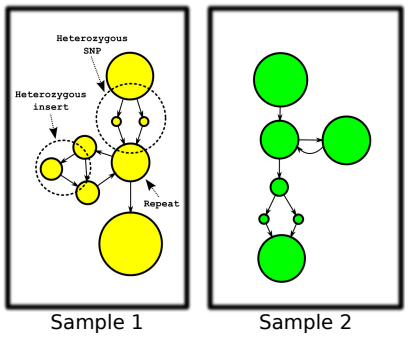
(d) Bubble structure obtained after fitting a prefix tree on U2.



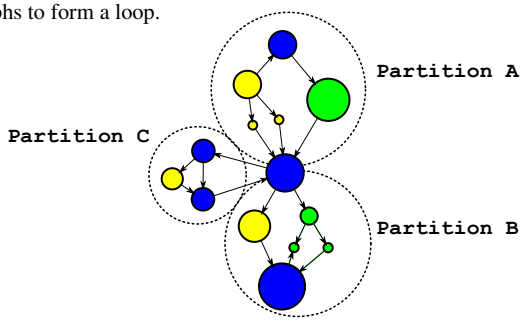
(e) Bubble structure obtained after fitting a prefix tree on the newly created ‘C’ node. Final result, no more pre- or suffixes can be merged.

Fig. 5

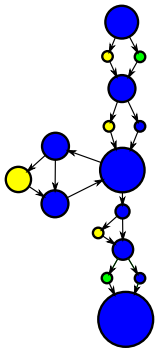
reads, to make sure that only unique matches are found and the risk of misalignment is minimized. Nodes in the resulting graph can now be divided into one out of three different nodes: nodes specific to sample one, nodes specific to sample two and nodes that are common to both sample one and sample two.



(a) The preprocessed variation graphs of sample 1 (yellow) and sample 2 (green). Heterozygous variations within sample 1 and form different bubble 'simple' structures. A repeat node causes both graphs to form a loop.



(b) Exact matching stretches of sequence, larger than k_{global}, are detected and merged into blue nodes. Graph partitioning is performed to extract subgraphs for further local merging.



(c) Local merging resulted in two additional 'simple' bubble structures, exposing two homozygous variations between sample 1 and sample 2. An indel and a SNP.

Fig. 6: Schematic overview of the different graphs resulting from 6a) assembly, 6b) global merging and 6c) local merging. Blue nodes represent 'merged' nodes and contain sequence shared between sample 1 and sample 2. The size of the nodes indicates the length of the associated sequence.

2.4 Local merge

After performing a global merge of the two graphs, a graph is obtained in which large unmerged bubble-like structures still occur (see Figure 6). The local merge step is aimed at collapsing these structures in order to obtain smaller bubbles that are interpretable as individual variations. To do this, the graph is segmented into multiple smaller graph structures. This segmentation is obtained by doing a breadth-first search from every merged node in the graph. During this search the orientation of nodes relative to the node from which the search was initiated is kept. The search is stopped when another merged vertex is detected. All traversed unmerged nodes are then considered as a subgraph that are identified by the node that was used to start the search.

Next, for every subgraph a specific suffix array and corresponding LCP array are constructed and the longest common substring for every partition is extracted. Since repeats can cause subgraphs to be overlapping, the exact matches are first added to a list and ordered by length. Then matches are sequentially processed from the largest to the shortest exact match and graphs are aligned in a similar way as during the global merge. When a node is encountered that was already part of a previously processed match, the alignment is skipped.

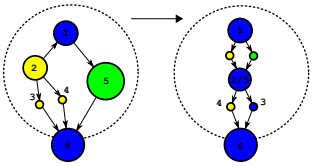
After merging, the graphs are partitioned again. This process is repeated until no more exact matching substrings are found that are longer than the predefined k_{local}. Theoretically, a value of 1 could be used for k_{local}, but to prevent small-scale misalignments and the resulting formation of complex bubbles in highly variable and/or repeat-rich parts of the genome, a slightly bigger value (eg. 5) was found to give better results. After this process, non-matching nodes contain the actual variations between the two graphs and are characterized by bubble-like structures (see Figure 7)).

Finally, a bubble-smoothing step is performed that is similar to the process described under preprocessing where a prefix tree is fitted in order to merge similar pre- and suffixes. This makes sure that the number of alleles contained in each 'simple' bubble is limited.

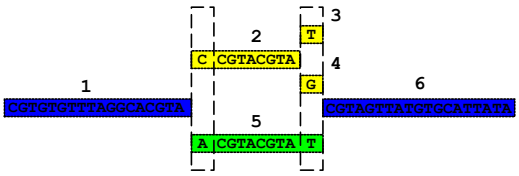
2.5 Bubble detection

The resulting graph after the local merge step contains different bubble structures for different sequence variations. SNPs and indels form different bubbles (see Figure 9). Sequence variations that are in close proximity of other variations or repeat regions within the same sample can cause more complicated bubbles. For example, in the case of a diploid organism, when two heterozygous SNPs are located in close proximity of one another the sequence in between the two variants is contained in the bubble structure. When merging these 'big' bubble structures with a different sample that does not have this combination of alleles on either chromosome, a more complicated bubble occurs (see Figure 8). Similar, bur far more complex bubble structures can occur when simultaneously assembling multiple samples, or when assembling variations in repetitive regions. These bubble structures can become very complex, and therefore hard to reduce to a comprehensive set of individual variant calls.

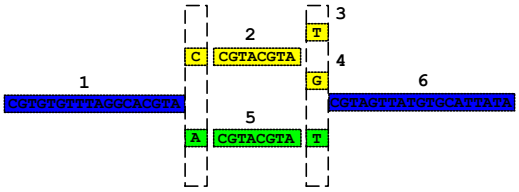
The actual variant calling as implemented here is therefore limited to the detection of 'simple' bubbles, as shown in Figure 9. 'Simple' bubbles are defined as bubbles that can be detected by traversing edges in one specific direction, for a maximum of two 'hops', as seen from the merged node. When two or more paths cross each other within these two 'hops', a bubble is detected. Previously mentioned 'big' bubbles do not always generate 'complex' bubble



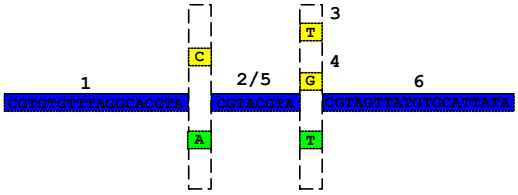
(a) High-level representation of the local merge steps described from 7b to 7e.



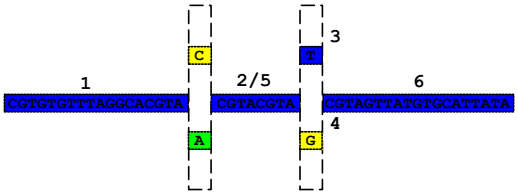
(b) Suffix and LCP array are constructed for unmerged nodes 2,3,4 and 5.



(c) The longest common substring is found between node 2 and node 5. Nodes are broken so that the exact matching parts can be aligned.



(d) A merged node '2/5' is created. The node is reconnected to all the neighbors of nodes 2 and 5.



(e) The graph is partitioned again. No more substrings longer than k_{local} can be detected. Prefix tree fitting (see Figure 5) eventually simplifies the second bubble structure by merging nodes 3 and 4.

Fig. 7: Steps involved in the local merge of partition A (see Figure 6b).

structures in the merged graph. In these cases, they can be detected, and subsequently the two branches of the bubble are aligned against each other using the Needleman-Wunsch alignment [13]. The following alignment is then translated into a set of variant calls.

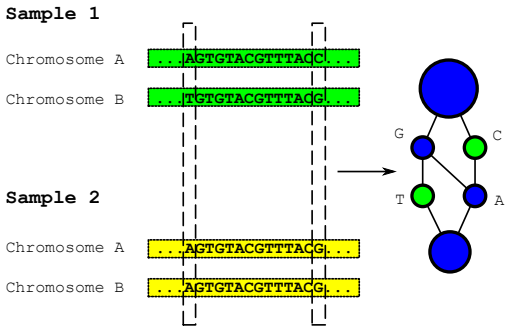


Fig. 8: Example of a more complicated bubble caused by side-by-side heterozygous mutations.

Bubbles do not always correspond to sequence variation. In the case of a longer inexact repeat sequence, a bubble structure can be formed within the same sample, without actually representing a heterozygous variation. However, when samples are then merged, it is expected that the same bubble structure will form in the next sample, resulting in a merged bubble structure for which all branches are seen in both samples. The bubble detection algorithm that was implemented here, does recognize these bubbles, and does not call variants in these cases.

2.6 Multiple samples

The graphs obtained by preprocessing and merging can be used for subsequent merge steps. In this way multi-sample graphs can be obtained in a modular way. Within these multi-sample variation graphs, bubble structures can be detected. These bubble structures can then be analyzed to characterize the kind of variation and to determine the genotype (see Figure 9). By merging a graph with a reference sequence (if available), the coordinate system defined by the reference sequence can be used to assess the location of variants with respect to their vicinity to for example genes.

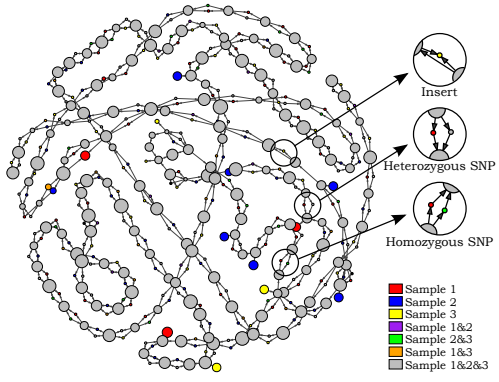


Fig. 9: Example of a multi-sample variation graph. Bubbles correspond to between and within sample variations. Colored tips in the graph correspond to errors, or variations for which only one valid overlap was observed.

Since merging or aligning two graphs is a heuristic process, the end result is not guaranteed to be optimal. The merging process is more likely to produce an optimal alignment of two graphs when the amount of sequence variation, or distance, between the two underlying samples is limited. In order to produce an optimally aligned multi-sample variation graph it is therefore advised to merge samples in a specific order. Methods, as described in [19], can be used to obtain a hierarchical clustering of samples without using a reference sequence to compare them. The obtained hierarchical clustering can then in turn be used in order to derive the optimal merging strategy. In this work, no further considerations are given to the order in which samples are merged, since variations in the datasets used here were uniform randomly distributed throughout the genome, and these effects are only expected to be of influence when merging large numbers of samples that contain some form of population structure.

2.7 Implementation details

A concept of the algorithm was implemented under the name Probubble. The actual program is split-up into three stages: preprocessing, merging and bubble detection. In the preprocessing stage, the cleaned output of Fermi [10] is loaded into a common graph structure (iGraph, [5]) and the steps, as outlined under preprocessing, are performed. The merge stage takes the input of two preprocessed graphs (or previously merged variation graphs) and outputs the resulting merged graph. A linear reference sequence can also be merged with a variation graph. In the bubble detection stage, 'simple' bubbles (as shown in Figure 9) are detected by traversing the graph. If a reference sequence was merged with the variation graph, the coordinate system of the reference sequence is used in order to position variants. Resulting variant calls are written to a VCF-file.

3 RESULTS

To assess the performance of Probubble, genomes were simulated based on 2.5MB of chromosome 22 of the human genome (hg19), in which different variations were introduced. Variations were subdivided into single nucleotide polymorphisms (SNPs) and indels. The proportion of SNPs and indels was set at a fixed rate of 90% and 10% respectively [4]. The simulated indel lengths were sampled from a Zipfian distribution with $\alpha = 1.65$ [2]. The total number of variations was determined by a mutation rate parameter *mu*, indicating the fraction of mutations with respect to the length of the initial template genome. A value of 0.001 was used as a typical value for the mutation rate parameter [4], indicating an average of one mutation in every 1000 base pairs. From these simulated genomes, Illumina 100bp paired-end reads, with a mean insert-size of 500 base pairs, were simulated using dwgsim [1]. These read sets were then assembled into string graphs and cleaned using Fermi [10].

The construction of the de-novo assembly graphs and the read simulation, prior to the actual merging algorithm, influences the success of graph-based variant detection. Two important parameters that influence the resulting string graphs are the read-depth coverage and the read-length overlap. For the conducted experiments, typical values of 40x coverage and an overlap of 50bp were used.

[1] http://sourceforge.net/apps/mediawiki/dnaa/index.php?title=Main_Page

3.1 Comparison of mapping and assembly-based callers

A first experiment was devised in order to compare Probubble to a mapping-based approach and the de Bruijn graph based reference-free variant caller Cortex. For the mapping-based approach the haplotype caller, as implemented in the Genome Analysis Toolkit (GATK) [6] was used in combination with the BWA-mem read aligner [11]. This variant caller can best be seen as a hybrid variant caller, since it locally performs de-novo assembly of partially aligned reads in order to be able to detect longer inserts. Cortex was used with default settings for quality filtering and two k-mer graphs for $k = 31$ and $k = 61$.

To compare the performance of the variant callers, simulated datasets were construfcted with an increasing mutation rate. Variants were subsequently called using the different algorithms. For GATK this means that reads were aligned to the reference sequence and subsequent variant calls were made by analyzing the pile-up of reads. For Cortex four de Bruijn graphs were constructed, two for the reference sequence (with $k = 31$ and one for $k = 61$), and two for the readset. Cortex then merges the resulting De Bruijn graphs (with corresponding values for k) and then detects bubbles in both graphs to call variants. Probubble merges a single de-novo assembly graph with the linear reference sequence in order to detect bubbles and call variations. The performance of the different algorithms was assessed based on the recall and precision measure, indicating the number of correctly detected SNPs and indels (see Figure 10).

It can be seen that the mapping-based approach outperforms both assembly-based methods and that Probubble performs better than Cortex when considering the variations of a single sample with respect to a reference sample. Investigation of the graphs shows that the better performance of GATK over Probubble can be mainly attributed to the fact that variations do not always form 'simple' bubbles in the assembly graphs, which mainly can be attributed to two reasons: 1) close by heterozygous variants and 2) formation of tip nodes. Variations that occur in close proximity of other heterozygous variants or repetitive areas can result in complex bubble structures, whereas high coverage tip nodes are created when suboptimal coverage limits the number of observed overlaps that satisfy the overlap threshold. With an increase in mutation rate, the fraction of complex bubbles also increases. That's why both reference-free based methods show a steep decline in performance when the mutation rate increases. The difference in performance between Cortex and Probubble has to be attributed to the fact that Cortex makes use of a less informative De Bruijn graph structure.

3.2 Most missed calls are heterozygous

It is important to note that the false negative, or missed variant calls of Probubble are mainly comprised of heterozygous variations. Considering the 2 to 1 ratio of the number of simulated heterozygous versus homozygous variants, one would expect to see a similar distribution of false negative variant calls. Figure 11 shows that false negatives are biased towards heterozygous variant calls. This supports the explanation that most variant calls are missed, because of either complex bubble formation or open bubbles due to lacks of overlapping reads. Which was commonly observed to happen with heterozygous variations (Section 2.5).

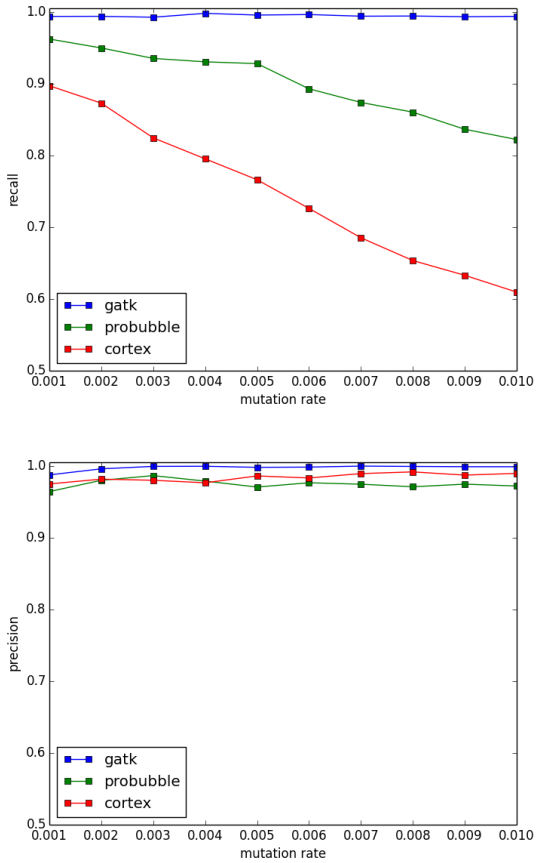


Fig. 10: Single run recall and precision measures of the different methods on a dataset with an increasing mutation rate.

3.3 Comparison of mapping- and assembly-based callers on longer indels

Further investigation of the missed calls shows that the GATK performs very well at shorter mutations. For longer indels, Cortex and Probubble outperform GATK's haplotype caller. We investigated the performance of the different methods on varying indel sizes, by sampling the indel sizes from a uniform distribution between 1 and 300, instead of the Zipfian distribution. The results of this experiment are shown in Figure 12.

Probubble and Cortex, show consequent performance on the entire range of indel sizes. GATK only performs well on short indels. Probubble performs slightly better than Cortex overall, with the exception of the precision in the 150bp-210bp range. This dip is caused by a failed Needleman-Wunsch alignment of a 'big' bubble, causing an increase in false positive variant calls. Multiple repetitions of the experiment would probably flatten out this observation.

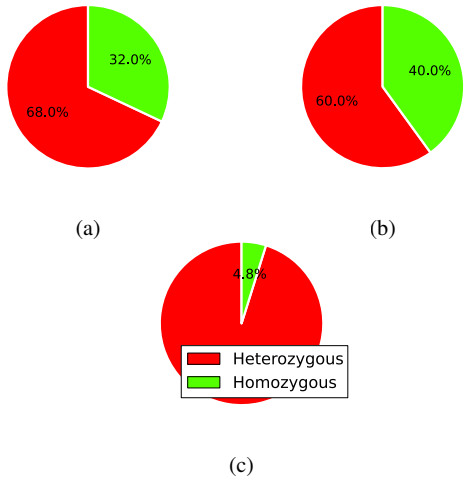


Fig. 11: Composition of missed calls by **a**: Cortex, **b**: GATK and **c**: Probubble. Only Probubble deviates from the expected 2/3 of missed heterozygous calls.

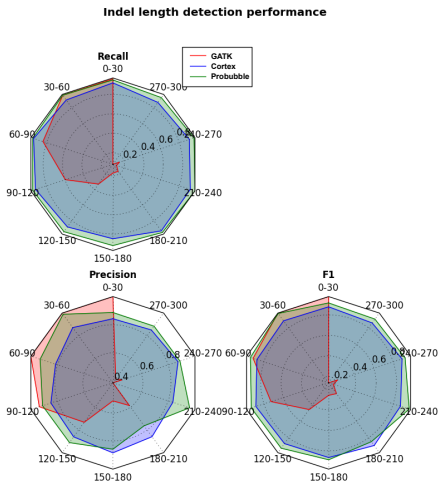


Fig. 12: Variant calling performance of the different methods, based on variations of increasing indel sizes.

3.4 Probubble outperforms GATK when dealing with distant reference genomes

The initial motivation for developing the reference-free variant caller was that mapping-based variant detection becomes problematic when there is no closely related reference genome available. Therefore we conducted a simulation experiment which mimics the absence of a closely related reference-genome. First a diverged haploid genome (R') was generated on the basis of the initial reference genome (R) with a mutation rate mu, see also Figure 13. This diverged reference genome (R') was then used to create two samples ($S0$ and $S1$) with a fixed mutation rate of 0.0005, such that

the distance between the two samples would approximate the typical mutation rate of 0.001 (see Figure 13).

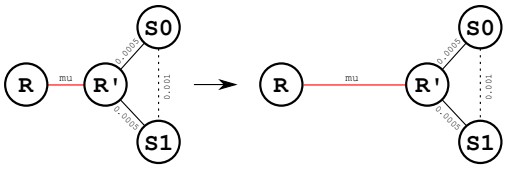


Fig. 13: Schematical representation of the setup of the experiment. The distance between the reference genome R and the diverged reference genome R' increases by incrementing the mutation rate parameter *mu*. Other distances are fixed.

Next, reads were sampled from *S0* and *S1*. These reads were then assembled and the resulting string graphs were merged using Probubble. The merged graph was then used to detect bubbles. Performance was validated directly against the diverged reference genome, R'.

For the mapping-based approach, the reads originating from *S0* and *S1* were aligned to the reference genome R. The insertions and deletions introduced by the creation of R' were tracked, so that calls made on R could still be traced to a location on R'. Then all calls that could be attributed to the creation of R' were filtered out. The remaining calls were expected to contain the actual variations between *S0* and *S1*, and only those were therefore evaluated in order to assess the performance.

To shorten the running time of the experiment, Cortex was not evaluated. Previous experiments already showed that Probubble outperformed Cortex and no changes with respect to this behaviour were expected here.

Figure 14 shows the recall and precision for this diverged reference genome experiment. The reference-free comparison indeed remains stable and the mapping-based performance declines when the mutation rate increases. An increasing mutation rate for the diverged genome causes that more and more reads originating from the diverged sequences *S0* and *S1* cannot be mapped back to the reference genome, because they are simply too different. As a result, the mapping-based approach can't call variants in these unmapped regions. Reference-free methods do not suffer from this as they compare the diverged genomes (*S0* and *S1*) directly to each other: the unmapped regions are now contained in the de-novo string graph and variants can be called when merging the graphs. Furthermore, the mapping based approach suffers from an increase in partially mapped reads, causing an increase in the number of false positive calls.

3.5 Merging multiple samples

Probubble is specifically designed to merge multiple graphs in a scalable way. Therefore we setup an experiment to test its behavior when merging an increasing amount of samples. For this, we simulated a series of samples with respect to a short template sequence (50000bp), each having a fixed mutation rate of 0.001. Samples were merged in a random sequential order. To assess the quality of merging two graphs, after each merge step we inspected

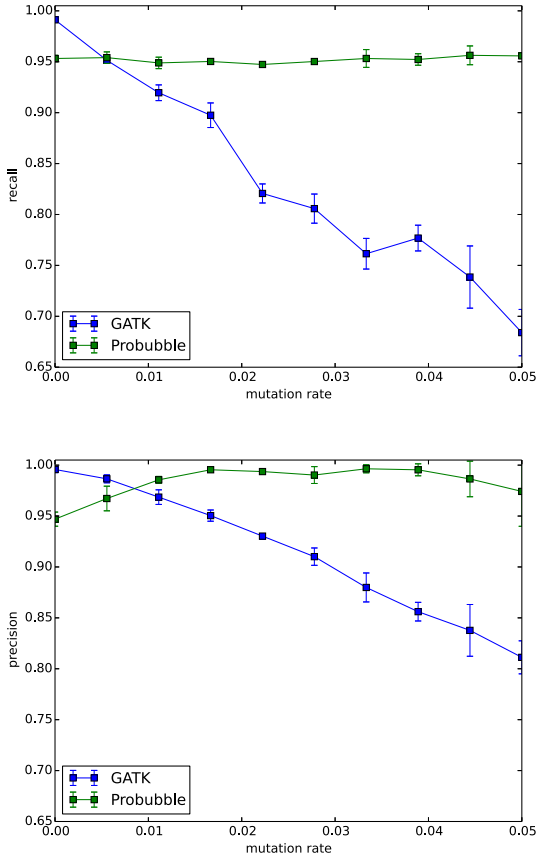


Fig. 14: Variant calling performance (averaged over three runs) based on the direct comparison of graphs using Probubble stays constant, while the performance of the mapping-based approach declines when the divergence of the reference increases.

the number of nodes that defined sequence that was observed in all merged samples. These nodes we'll define as 'common' nodes, since they define sequence that is common to all samples in the graph, i.e. the grey nodes in Figure 9.

Figure 15 shows the increase in 'common' nodes. From this figure one can observe that while more samples are merged in the same graph, the total number of common nodes increases. This is due to the fact that every time an additional sample is merged, the new graph needs to incorporate all the variations of the new sample (which in our simulation are independent from the previous). Thus, each variation contained in the graph to be merged, introduces a break-up of an existing merged node, creating a new bubble in the resulting merged graph. As a consequence, this (on average) creates an additional merged node for every variation that is added to the graph. We indeed observe that the number of common nodes increases linearly with the number of merged samples.

When misalignments start to occur it is expected that the linear increase in 'common' nodes will be converted into a decrease in

'common' nodes. Figure 15 shows that these effects start to show when +/-25 samples are merged. This shows that the total number of variations to be held within one graph is limited. Note that the mutation rate in our simulations were uniformly distributed for every sample and independent from the other samples, which is not a realistic case for real genomes. We expect most variations within a populations of samples to be common, thus dependent, and as a result we expect that the actual number of samples that can be merged in reality is much larger.

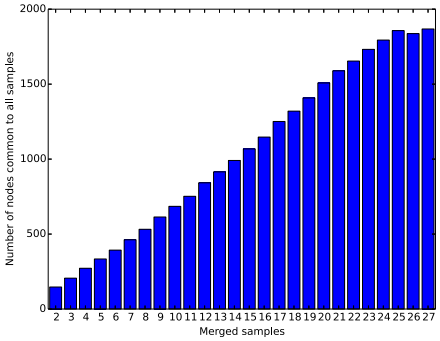


Fig. 15: Increase in the number of nodes that define sequence that is seen in all samples, while sequentially adding more samples.

4 DISCUSSION

We introduced a new approach to variant calling using overlap-based de-novo assembly graphs. We have shown that our method, called Probubble, outperforms the de Bruijn graph based alternative Cortex using its default settings when applied on simulated datasets. This is mainly attributed to the fact that a de Bruijn graph representation introduces more repetitive nodes in the assembly graph then strictly necessary when considering the entire set of reads as opposed to a string graph. These unnecessary repetitive elements in the de Bruijn graph increase the number of 'confounded bubbles' [9] with respect to a string graph representation, causing the method to be unable to detect all introduced variations.

We clearly showed the benefit of a reference-free variant calling approach (over a mapping-based version) when variations need to be detected in a genome for which no closely related reference genome is available. We showed that this already is the case for diverged genomes with a mutation rate of more than 0.01. Todays most powerful mapping-based variant caller, the GATK haplotype caller, cannot cope with such diverged samples since it is hampered too much by not being able to detect indel variations that are longer than the read length. Even the local de-novo assembly does not help, as it still depends on partially mapped reads that for large inserts map only at the flanks of the insert and are apparently not long enough to reconstruct the entire insert. All of this shows great promise for reference-free variant calling. As we showed that our method, Probubble, scales linear in time with respect to the number of comparable samples (as opposed to string graph based alternatives, like contrast assembly) and outperforms de Bruijn graph based approaches in terms of variant-calling, we have made an important step towards the detection of variants with a totally new approach.

It is important to note that only GATK used the paired-end information that was included in the simulated readset. We expect that both reference-free variant callers will improve when they include the paired-end information in their de-novo assembly. On the other hand, it is important to realize that de-novo based methods generally will need higher read coverages than mapping-based approaches, which currently will hamper their routine application.

The current implementation of Probubble is not capable of handling complex bubbles that arise when multiple different variants are in close proximity of each other (or at flanks of repeat regions) or in tip nodes (arising from limited coverage). More advanced preprocessing steps, such as aligning the branches of 'big' heterozygous bubbles, and better bubble detection algorithms will resolve this. However, it is still an open issue on how to convert complex bubbles to actual variant calls.

The current implementation also is only suitable for a rather small dataset. This might not grasp the complexities of full genomes completely. Therefore, the performance has to be re-assessed on larger datasets when there is a more efficient implementation of Probubble.

The reported precision and recall are biased by the composition of the dataset. That is, we simulated single nucleotide variations and indels but variations like inversions, translocations and copy-number variations were not simulated since they are not detectable by the variant callers considered here. Also, the simulated insertions consisted of random DNA sequence which reduces the probability that longer insertions cause repeat nodes in the graph (something that in reality is likely to happen with longer insertions [15]). Therefore, the results represented here with respect to the detection of longer indels might be over-optimistic.

Probubble assumes an equal copy-number for sequence matches between unitigs of two assembly graphs. This is not always the case, since sequence variation between samples can cause a stretch of sequence to be repetitive in one sample, but not in the other. With the current algorithm, these stretches of sequence are (falsely) mapped on top of each other, leading to additional repeat structures. When this situation can be detected beforehand, it can be used to, instead of creating them, resolve repeat nodes in the merged graph.

Much of the work presented here is not final, and no real data has been addressed, but beyond that, an important step has been made towards the reference-free comparison of complete genomes of large numbers of samples. Something that most definitely is a challenge for the very near future.

REFERENCES

[1]B. Alberts, D. Bray, J. Lewis, M. Raff, K. Roberts, and J. Watson. *Molecular Biology of the Cell*. Garland Science, New York, 5th edition, 2008.

[2]R. A. Cartwright. Problems and solutions for estimating indel rates and length distributions. *Molecular biology and evolution*, 26(2):473–480, November 28 2008.

[3]C. Clark and J. Kalita. A comparison of algorithms for the pairwise alignment of biological networks. *Bioinformatics*, May 2014.

[4]. G. P. Consortium. A map of human genome variation from population-scale sequencing. *Nature*, May 2011.

[5]G. Csardi and T. Nepusz. The igraph software package for complex network research. *InterJournal*, Complex Systems:1695, 2006.

[6]M. A. DePristo, E. Banks, R. Poplin, K. V. Garimella, J. R. Maguire, C. Hartl, A. A. Philippakis, G. del Angel, M. A. Rivas, M. Hanna, A. McKenna, T. J. Fennell, A. M. Kernytsky, A. Y. Sivachenko, K. Cibulskis, S. B. Gabriel, and D. A. . M. J. Daly. A framework for variation discovery and genotyping using next-generation dna sequencing data. *Nature Genetics*, 43(5):491–498, May 2011.

[7]J. Fischer. Inducing the lcp-array. *CoRR*, abs/1101.3448, 2011.

[8]Z. Iqbal, M. Caccamo, I. Turner, P. Flicek, and G. McVean. De novo assembly and genotyping of variants using colored de bruijn graphs. *Nature*, 44(2):226–32, January 8 2012.

[9]Z. Iqbal, M. Caccamo, I. Turner, P. Flicek, and G. McVean. De novo assembly and genotyping of variants using colored de bruijn graphs. *Supplement*, 2012.

[10]H. Li. Exploring single-sample snp and indel calling with whole-genome de novo assembly. *Bioinformatics*, 28(14):1838–1844, July 15 2012.

[11]H. Li and R. Durbin. Fast and accurate long-read alignment with burrows-wheeler transform. *Bioinformatics (Oxford, England)*, 26(5):589–595, Mar. 2010.

[12]E. W. Myers. The fragment assembly string graph. *Bioinformatics*, 21(2):79–85, January 2005.

[13]S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, Mar. 1970.

[14]A. W. Pang, J. R. MacDonald, D. Pinto, J. Wei, M. A. Rafiq, D. F. Conrad, H. Park, M. E. Hurles, C. Lee, J. C. Venter, E. F. Kirkness, S. Levy, L. Feuk, and S. W. Scherer. Research towards a comprehensive structural variation map of an individual human genome. *Genome Biology*, 11(5), May 2010.

[15]J. Sebat. Large-scale copy number polymorphism in the human genome. *Science*, 305(5683):525–528, July 2004.

[16]A. M. S. Shrestha, M. C. Frith, and P. Horton. A bioinformatician's guide to the forefront of suffix array construction algorithms. *Briefings in Bioinformatics*, 15(2):138–154, Mar 2014.

[17]J. T. Simpson. *Efficient sequence assembly and variant calling using compressed data structures*. PhD thesis, University of Cambridge, September 2012.

[18]J. T. Simpson and R. Durbin. Efficient de novo assembly of large genomes using compressed data structures. *Genome Research*, 22(3):549–556, Mar. 2012.

[19]K. Song, J. Ren, G. Reinert, M. Deng, M. S. Waterman, and F. Sun. New developments of alignment-free sequence comparison: measures, statistics and next-generation sequencing. *Briefings in Bioinformatics*, 15(3):343–353, May 2014.

Appendix

1 THE SUFFIX ARRAY

In order to quickly search large amounts of sequence data, there are two main indexing approaches: suffix trees and suffix arrays [16]. Because of recent advances in the efficient construction of suffix arrays using induced sorting techniques [16] and the relatively large memory footprint of suffix trees, suffix arrays are now found at the heart of many popular sequence-based bioinformatics applications. Suffix arrays can also be used in order to find the longest common substring in a set of sequences. In the case described here, these sequences originate from the unitigs of the two string graphs that are to be merged.

To be more precise: Let U_i denote the sequence that is defined on a unitig and let $\bar{U}_i$ be its reverse complement. All characters in the sequences originate from the ordered alphabet $\Sigma = \{A, C, G, T\}$. To index these sequences in a suffix array, all unitigs and their reverse complements are concatenated and separated by sentinels. The resulting text can be denoted as $T = [U_1\$, \bar{U}_1\$, ..., U_n\$, \bar{U}_n\$]$, where n is the total number of unitigs to be indexed and \$ is a unique sentinel character that is lexicographically lower than all other characters in Σ. Since the unitigs of the first graph are concatenated into T before the unitigs of the second graph, the sentinel at position β indicates the boundary between the graphs, where β is equal to two times the number of unitigs in the first graph. The substring of T ranging from i to j is denoted by $T_{i...j}$, for $1 \leq i \leq j \leq n$. The substring $T_{i...n}$ is called the i'th suffix of T and is denoted by S_i. Since sentinels are unique, the string S_i is terminated when a sentinel is encountered. The suffix array $SA[1, n]$ of T is a permutation of the integers in $[1, n]$, such that $S_{SA[i-1]} <_{lex} S_{SA[i]}$ for all $1 < i \leq n$. In other words, SA describes the lexicographical order of suffixes (see Figure 16).

2 THE LCP ARRAY

The longest common prefix array (LCP) can be seen as a supplementary data structure to the suffix array, and describes the length of the longest common prefix between $S_{SA[i]}$ and $S_{SA[i+1]}$ for LCP_i, where LCP_0 has a fixed value of -1 (see Figure 16). From the LCP array it is easy to derive the longest common substring that is found in both sets of unitigs. By looking at $argmax(LCP)$, to make sure that this substring is observed in both graphs, the following constraint $SA_{argmax(LCP)} > \beta \veebar SA_{argmax(LCP)-1} > \beta$ has to be true. Various linear time algorithms have recently been proposed [7] to efficiently calculate the LCP array alongside the construction of the suffix array. For the alignment of two assembly graphs the algorithm proposed here uses exact matching stretches of sequence between two graphs. The LCP array is used for this purpose. From this requirement it follows that it is not necessary to calculate every value of the LCP array, since the algorithm is constrained to finding the longest common substring that occurs in both graphs. So that suffixes that originated from the same graph do not need to be compared against each other. In the example outlined in Figure 16, this means that the calculation of the LCP for indices $1, 3, 9, 6, 7$ and 12 could be skipped.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T	**G**	**T**	**A**	**C**	G	$\$^0$	G	T	A	$\$^1$	C	**G**	**T**	**A**	**C**	A	$\$^2$	T	A	T	$\$^3$
SA	20	16	9	5	15	8	13	2	18	14	3	10	4	6	11	0	19	7	12	1	17
	$\$^3$	$\$^2$	$\$^1$	$\$^0$	A	A	A	A	A	C	C	C	G	G	G	G	T	T	T	T	T
					$\$^2$	$\$^1$	C	C	T	A	G	G	$\$^0$	T	T	T	$\$^3$	A	A	A	A
							A	G	$\$^3$	$\$^2$	$\$^0$	T		A	A	A		$\$^2$	C	C	T
							$\$^2$	$\$^0$				A		$\$^1$	C	C			A	G	$\$^3$
												C			A	G			$\$^2$	$\$^0$	
												A			$\$^2$	$\$^0$					
												$\$^2$									
LCP	-1	0	0	0	0	1	1	2	1	0	1	2	0	1	3	**4**	0	1	2	3	2

Fig. 16: Example of four unitig sequences (GTACG, GTA, CGTACA and TAT) originating from two string graphs (indicated with blue and red). Longest common valid substring starts at positions 0 and 11 in T and has a length of 4. For clarity, reverse complements of the sequences have been left out.