An algorithm is presented for fast construction of graphs of reads, where an edge between two reads indicates an approximate overlap between the reads. Since the algorithm finds approximate overlaps directly, it can process reads without error-correction preprocessing steps.

Extensions of the algorithm, such as construction graphs of overlapping pairs of reads, are discussed.

The algorithm can be used to construct graphs for assembly and for other related applications such as error correction. Preliminary experimental results indicate that the algorithm constructs graphs which lead to relatively long contigs even in the presence of sequencing errors.

Building approximate overlap graphs for DNA assembly using random-permutations-based search.

Roy R. Lederman[†]

Technical Report YALEU/DCS/TR-1470
December 18, 2012

Formating updated June 25, 2015.

[‡] Applied Mathematics Program, Yale University, New Haven CT 06511

Keywords: *DNA, alignment, assembly, random permutations.*

1 Introduction

One of the classical problems in the study of genomes is the problem of assembling many short reads into an accurate representation of a genome. Reviews of the assembly problem and the existing algorithms for assembly are available in [2, 8, 11, 7].

Assembly algorithms are based on two classes of graphs: graphs of reads used in overlap-layout-consensus algorithms (OLC), and de-Bruijn graphs (DBG) of short substrings of reads.

OLC algorithms [9] require graphs of reads which indicate which reads overlap. Finding overlaps between reads can be computationally expensive: the naive approach would require $O(N^2)$ comparisons and alignments between reads (where N is the number of reads). There are several algorithms for relatively fast calculation of graphs of reads when there are no errors in the reads [12, 13, 1, 3]. However, these algorithms may be vulnerable to sequencing errors. There are some extension for these algorithms, and algorithms which have been proposed for approximate overlaps of reads with errors [14].

DBGs are often used for assembly because they can be constructed relatively fast. Although there are fast algorithms for analyzing DBGs, DBGs may be more ambiguous and not read-coherent because they are based on short substrings of reads. In addition, DBGs can be large and may require considerable storage (RAM, disk space, and i/o operations). Like OLC algorithms which use graphs of exact overlaps, DBG algorithms are vulnerable to sequencing errors.

Several strategies have been proposed for dealing with the vulnerability of DBG and OLC algorithms to sequencing errors. One of the strategies is an error-correction procedure which locates and fixes some of the sequencing error before the algorithms are used. In some cases, these error-correction procedures are the most computationally expensive step in the assembly, and efficient parallelization of error-correction algorithms can be difficult.

In [4] we introduced a new approach to read alignment, based on sorted arrays of permuted strings. Here, we describe how permutations-based search can be use to rapidly build graphs of reads without any error-correction preprocessing. We describe a simple algorithm for analyzing graphs of reads, but note that more elaborate graph analysis algorithms may be useful in applications.

2 Preliminaries

2.1 The assembly problem

In this paper we consider the following scenario: There is some long string W, which represents the DNA which is being sequenced. The sequencing procedure produces a library of short reads of length M. These reads are contiguous substrings of W, taken from random locations in W. There are sequencing errors, which are modeled as random substitutions of characters in the strings with some error probability.

In sequencing, the output can be substrings of W and reverse complements of the substrings. To simplify the description, we ignore the possibility of reverse complements. The reverse complements can be added to the model by adding the reverse complement of each original read to the library of reads (as we did in the implementation described below).

Given a large library of such short reads, we would like to estimate W. In general, the problem need not have a unique solution even when there are no errors and even under some reasonable assumptions on the length of W : the reads may not cover all the characters of W, and there may be multiple valid assemblies. Errors further complicate the problem.

Here we consider an initial step of an assembly procedure, which requires the assembler to produce contiguous strings ("contigs") which contain long substrings of W ("corrected contigs") with a small number of errors.

2.2 Graphs of reads

The framework for assembly which we describe in this paper involves creating graphs of reads. Each read is a vertex in the graph, and the directed edges in the graph represent possible extensions of the reads: if the read X_i overlaps with X_j, and X_j extends beyond the end of X_i, then there is a directed edge from node i to node j.

Each overlap can has some shift: the first character in X_j can correspond to the tenth character in X_i. The value of the shift can be positive, negative, or zero. We use directed edges with positive values, a negative shift is represented by a directed edge in the opposite direction. For simplicity, we ignore the cases of shifts by zero, reads that contain each other and multiple possible shifts between reads, but these can be added.

When a correct graph is available, W can be estimated by traveling along paths in the graph (and correcting errors by "voting" for the correct character in each position). In practice, the graph may have false edges (ambiguity in extending the reads) and missing edges, therefore graph analysis algorithms are required. Here we focus on building graphs of reads with low ambiguity and few missing edges for libraries of reads that have sequencing errors.

2.3 Permutations-based search

Permutations-based search for biological applications is introduced in [4]. Given a library of strings $\{X_i\}_{i=1}^{N}$, all of equal length M, permutations-based searches allow to find the best matches for a query string Y of length M with high probability.

The basic idea in permutations-based search is to create several libraries of randomly permuted versions of $\{X_i\}_{i=1}^N$ and sort each of these permuted libraries lexicographically.

We create the random permutation $U^{(j)}$, and permute all the reference strings using this permutation. The library of permuted strings, $\{X_i^{(j)}\}_{i=1}^N$ is sorted lexicographically. We then permute the query string Y to get the permuted version $Y^{(j)}$. If we are "lucky", the mismatches in Y are all moved to the end of the permuted version, so that when look at the location in the sorted list where $Y^{(j)}$ should have appeared, we find a small subset of $\{X_i^{(j)}\}_{i=1}^N$ that contains the correct best match. The procedure is repeated several times to obtain any desired probability of error.

As shown in [4], the procedure is fast and accurate. There are several extensions that reduce the memory size and make the procedure faster and more flexible [4, 6, 5].

2.4 Types of neighbors

In this manuscript we use the term "neighbor" two different ways. In this subsection we define the different types of "neighbors" and the context in which they are used.

In the assembly problem, we assume that the reads are substrings of some long string. We assume that many of the reads represent overlapping segments of the long string. We refer to all the reads that overlap with the read Y as the "overlap-neighbors" of Y. In graphs of reads, the overlap-neighbors of Y are connected to Y by edges.

One of the steps in permutations-based search algorithms is to look for the lexicographical position of a permuted version of the query string in a sorted list of permuted reference strings. We refer to the strings around the lexicographical position of the permuted query as the "list-neighbors" of the permuted query string. This search operation is analogous to looking up a word in a dictionary, so the "list-neighbors" are analogous to the words on the page where the query should have appeared.

3 Algorithm

Assume a library of N reads, each of length M. We denote the ith read by X_i. We assume that the library contains the reverse complement of each of the reads, so for the read X_i there is another read X_j which is the reverse complement of X_i.

We denote a substring of X_i which begins at the kth character of X_i by $X_{i(k)}$. We denote the permuted version of $X_{i(k)}$ by $X_{i(k)}^{(z)}$, where (z) identifies the permutation used.

We assume some some method of comparing reads to validate that they are "possible" overlap neighbors and rank their compatibility. For example, a validation can require an overlap of at least 40 with no more than 5 mismatches. A score function can be of the form: $\alpha \times (\textit{number of matches}) - \beta \times (\textit{number of mismatches})$.

We define the parameter J, the number of permutations to use. We also define the parameter m_1, where $M - m_1 + 1$ is the minimum overlap that we require between reads.

We construct a directed graph, where each edge has a positive shift. We ignore the cases of overlaps with shift 0, reads which are contained in each other, and the possibility of multiple valid shifts, but these cases are handled by the algorithm, and can be handled more specifically according the desired specifications of the graph.

Step 1 of the algorithm creates sorted libraries of permuted reads for the search procedures.

Step 2 of the algorithm finds candidate "overlap-neighbors" with a positive shift for each of the reads.

Step 3 is a simplified graph analysis algorithm that proposed "contigs".

Algorithm 3.1

```
Step 1:  IndexReads
       For k=2 to $m_1$
           Create J permutations of length $m = M - k + 1$:  $\{U^{(m,j)}\}_{j=1}^{J}$.
           For j=1 to J
               Truncate the reads to length $m$.
               Create a library of permuted truncated reads.
               Sort the library lexicographically.
               Store the sorted library as index $Ar^{(m,j)}$.
           End For
       End For
```

Algorithm 3.2

```
Step 2: BuildGraph
       For i=1 to N (all reads)
           For k=2 to $m_1$ (all shifts)
               Set $m = M - k + 1$
               For j=1 to $J$ (all indexes for the shift, or a random subset)
                   Find the list-neighbors of $X_{i(k)}^{(m,j)}$ in $Ar^{(m,j)}$.
                   Validate the list-neighbors.
                   For each valid list neighbor $w$:
                       Add the edge $(i, w)$ with distance $k - 1$.
                   End For
               End For
           End For
```

```
        End For
```

Algorithm 3.3

```
Step 3: Analyze graph
        Calculate scores for all edges.
        For each vertex,
            keep only the outgoing edge with the highest score.
        End For
        For each path in the graph,
            calculate a consensus string and report as a proposed contig.
        End For
```

3.1 Reducing the number of indexes and string operations

The simplified description of the algorithm requires a large number of indexes. The number of indexes can be reduced significantly by choosing only a subset of the $m_1 - 1$ possible lengths (or even only a single length). In addition, as shown in [5], the J permutations for each possible length can be replaced with a small number of permutations, which are applied to shifted version of the reads. When these modification are implemented, only very few indexes are required and relatively few searches are required for each read.

Other methods used in permutations-based algorithms can also be applied to reduce the number of list-neighbor validations.

3.2 Using paired-end reads

To further reduce the number of validations and the number of invalid edges in the graph, we can use the pairs of the reads. Suppose that the read Y_1 has the paired read Y_2. We construct a list of possible overlap-neighbors for Y_1 and a list of overlap neighbors for Y_2. Suppose that Z_1 is an overlap-neighbor of Y_1 and that Z_2 is the pair of Z_1. If Z_2 is not an overlap neighbor of Y_2, we remove Z_1 from the list of overlap-neighbors of Y_1.

Many existing assembly algorithms ignore paired reads until late in the process. Using the information about pairs allows this algorithm to eliminate many of the invalid paths and can also accelerate the algorithm.

3.3 Indels

Permutations-based searches allow mismatch errors. However, the basic permutations-based search is vulnerable to indels. Some sequencing technologies produce mainly mismatches and are therefore very compatible with permutations-based searches. Some other technologies produce mostly homopolymer-length-error indels. In [6] we describe a framework that allows to use permutations-type algorithms for reads which have homopolymer-length-errors.

3.4 Analyzing the graph

The graph analysis described above can be replaced by better graph analysis algorithms. Existing graph analysis approaches can be applied to the graph created by the algorithm described here.

The construction of the graph can be done "on-demand" by searching for overlap-neighbors with parameters given by the assembly algorithm. For example, when a path terminates, the assembly algorithm may request additional searches to find additional overlap-neighbors using more loose search parameters. Similarly, a greedy algorithm can start at some read and request for valid extensions of the read recursively.

4 Implementation and results

We implemented a simple version of the algorithm in C. We tested the algorithm by simulating reads with simulated errors from reference genomes.

We used the implementation to create graphs of simulated reads. For each read, we kept only the outgoing edge associated with the "overlap-neighbor-candidate" that obtained the highest overlap score, and removed all other edges. The resulting graph had only a single outgoing edge for each of the reads.

To check the results, we then removed the false edges connecting reads which were not true overlap-neighbors in the original reference. We did not restore any of the lower-scoring edges which we previously removed, so a read that had a false outgoing edge was left with no outgoing edges.

Finally, we measured the length of longest correct contig in which each position of the reference appears.

Clearly, this analysis can allow regions to be covered by several contigs. However, since we kept at most one outgoing edge for every read, long contigs imply long paths in which ambiguities are correctly resolved consistently.

The reads were simulated from a bacterial reference genome and from a human chromosome reference. In the human chromosome, we used the GRCh37 reference and removed the nucleotides labeled with “N”, but did not remove other regions which are considered repetitive. The simulated reads were paired-end, 100 characters long, with an error rate of 1%. The reverse complement of each pair was added to the database. The comparison was performed on a cluster node with (2) E5620 CPUs and 48GB RAM. The program used only a single process.

For a bacterial reference DNA of E.coli of length 4.6×10^6 characters, we used 3×10^6 pairs of reads. 90% of the reference was covered by contigs of weighted average length $\approx 140 \times 10^3$ characters. 50% of the reference was covered by contigs of weighted average length $\approx 193 \times 10^3$.

For a single chromosome of human DNA (chromosome 22, 35 million characters, not diploid), we used 20×10^6 pairs of reads. 90% of the reference was covered by contigs of weighted average length $\approx 88 \times 10^3$ and 50% of the reference was covered by contigs of weighted average length $\approx 126 \times 10^3$.

The analysis of the bacterial DNA took 9 hours (single process) and the analysis of the human DNA took 85 hours (single process). Based on our experiments with more advanced versions of permutations-based search, we estimate that it is possible to build graphs for libraries of about $10^8 - 10^9$ pairs of uncorrected reads in significantly less than 1 hour per million pairs (likely as little as several minutes per million pairs). Much of the process can be parallelized.

This procedure does not require an error-correction procedure prior to assembly and may produce graphs with relatively few ambiguities.

4.1 Applications

The algorithm can be adapted for use in applications such as:

- De novo assembly.
- Error correction.
- Alignment in the presence of significant mutations or errors.
- Structural variation discovery, analysis of copy number variation.
- Alternative splicing / Gene expression / RNA-Seq.
- Assembly of different paths in populations, using reads from multiple individuals.
- Mixture sequencing (analysis or assembly of mixtures of different cells)
- Assembly around points of interest (“Targeted assembly” [10]).
- Discovery of artificial splicing points (in experiments where segments are spliced for sequencing).

5 Conclusions

An algorithm has been presented for the construction of graphs of reads for OLC algorithms. The algorithm allows substitution errors in the reads, and therefore does not require a computationally expensive error correction preprocessing steps. In fact, the algorithm can be used to perform error correction.

The handling substitutions also allows to analyze diploid samples and populations in the presence of SNPs (the desired policy for representing the SNPs should be determined by the validation procedure and graph-analysis algorithm).

The algorithm uses information from entire reads and can be extended to use information from both reads in the paired-end case. This allows the algorithm to eliminate many ambiguities in graphs, accelerates the graph construction, and reduces the storage requirements.

The current implementation uses a simple version of permutations-based algorithms. It is possible to implement a version that is much faster and more memory efficient.

Algorithms for analyzing the graphs are not the main subject of this paper. We demonstrated that simple graph analysis algorithms can produce good assembly results, but propose to use more elaborate algorithms in applications.

The preliminary results presented in this paper refer to simplistic cases. We plan to implement a faster, more efficient and more flexible version that can be compared to other algorithms directly using real data.

References

[1] Hieu Dinh and Sanguthevar Rajasekaran. A memory-efficient data structure representing exact-match overlap graphs with application for next-generation DNA assembly. *Bioinformatics*, 27(14):1901–1907, July 2011.

[2] Paul Flicek and Ewan Birney. Sense from sequence reads: methods for alignment and assembly. *Nat Meth*, 6(11s):S6–S12, November 2009.

[3] Giorgio Gonnella and Stefan Kurtz. Readjoiner: a fast and memory efficient string graph-based sequence assembler. *BMC Bioinformatics*, 13(1):82, 2012.

[4] Roy Lederman. A Nearest Neighbors Algorithm for Strings. Technical report, YALEU/DCS/TR1453, 2012.

[5] Roy Lederman. A random-permutations-based approach to read alignment. *In preparation*, 2012.

[6] Roy Lederman. Homopolymer Length Filters. Technical report, YALEU/DCS/TR1465, 2012.

[7] Zhenyu Li, Yanxiang Chen, Desheng Mu, Jianying Yuan, Yujian Shi, Hao Zhang, Jun Gan, Nan Li, Xuesong Hu, Binghang Liu, Bicheng Yang, and Wei Fan. Comparison of the two major classes of assembly algorithms: overlaplayoutconsensus and de-bruijn-graph. *Briefings in Functional Genomics*, 11(1):25–37, January 2012.

[8] Jason R. Miller, Sergey Koren, and Granger Sutton. Assembly algorithms for next-generation sequencing data. *Genomics*, 95(6):315–327, June 2010.

[9] Eugene W. Myers. The fragment assembly string graph. *Bioinformatics*, 21(suppl 2):ii79–ii85, January 2005.

[10] P. M. Peterlongo and R. M. Chikhi. Mapsembler, targeted and micro assembly of large NGS datasets on a desktop computer. *BMC bioinformatics*, 13(1):48, 2012.

[11] Michael C. Schatz, Arthur L. Delcher, and Steven L. Salzberg. Assembly of large genomes using second-generation sequencing. *Genome Research*, 20(9):1165–1173, September 2010.

[12] Jared T. Simpson and Richard Durbin. Efficient construction of an assembly string graph using the FM-index. *Bioinformatics*, 26(12):i367–i373, June 2010.

[13] Jared T. Simpson and Richard Durbin. Efficient de novo assembly of large genomes using compressed data structures. *Genome Research*, 22(3):549–556, March 2012.

[14] Niko Välimäki, Susana Ladra, and Veli Mäkinen. Approximate all-pairs suffix/prefix overlaps. *Special Issue: Combinatorial Pattern Matching (CPM 2010)*, 213(0):49–58, April 2012.