Bliss: A New Read Overlap Detection Algorithm

Sayali Davalbhakta *sayalid@gmail.com*
Department of Computer Engineering and IT
College of Engineering, Pune
Shivaji Nagar, 411005, India

Abstract

Many assemblers carry out heuristic based overlap detection to avoid string comparisons. But heuristics skips many true overlaps. Also, the order of the number of read pairs compared for overlaps is higher than the order of n. In the raw approach it would be n^2 where every read is compared to every other read. Some assemblers have used a hybrid approach to bring the order down from n^2. Here is an algorithm which works with 100% accuracy. As there is no heuristics involved, it is able to report all the overlaps in a given set of reads without actual string comparisons. It achieves this purely by querying the k-mer position data. Moreover, the number of read pairs compared is proportional to the number of reads present i.e. of the order of n.

Keywords: Overlap Detection, De Novo Assembly, Overlap-layout-consensus, OLC Assemblers, DNA Sequencing, Whole Genome Shotgun Assembly.

1. INTRODUCTION

As the sequencing techniques advance, the assemblers need to adapt to the trend. The EULER assembler introduced the use of de Bruijn graphs in assembly [1]. These assemblers suffer from heavy memory consumption [2]. The other approach conventionally taken by some de novo assemblers is the overlap-layout-consensus (OLC) approach [3]. The primary difference between the two approaches is that de Bruijn graph is a graph with k-mers as nodes and an edge corresponds to k-1 characters of overlap. Whereas the read overlap graph has reads as nodes and edges correspond to the overlap between reads. Velvet and SOAPdenovo are examples of de Bruijn based assemblers [2], [4]. De Bruijn graph based assemblers suffer from the problem of huge memory consumption. Several solutions have been devised by assemblers to tackle the memory consumption issue. ABySS takes a distributed approach towards building the de Bruijn graph [5], [6]. Some others try representing the de Bruijn graph as a sparse graph [7], [8]. Celera, ARACHNE are examples of OLC assemblers [9].

The OLC assemblers typically have overlap detection among one of the initial phases. The very basic approach would compare every read with every other read for overlap. Thus it will require N^2 comparisons. This has been optimized by assemblers. During the overlap detection phase, ARACHNE generates k-mers and stores along with the source read information [9]. It then sorts the k-mer table so that reads sharing k-mers are adjacent. Only the reads sharing some k-mers are considered for overlap detection. This is great improvement over the raw N^2 comparison approach.

This paper makes two significant advances over this approach to further improve performance. The only overlaps of interest are the ones that occur at the ends i.e. end of one read overlaps with the beginning of another read. Two reads having overlap which does not span till the end are of no use in read alignment. Such overlap is due to repeats in the original sequence. Bliss, the new algorithm proposed in this paper, segregates the k-mers in two different data structures. Assuming the k-mer positions in a read starting at zero, the $zero^{th}$ k-mers of all reads i.e. the ones occurring in the beginning of reads, are stored separately along with the source read. There is no

need to store the position separately because it is zero for all of them. The other k-mers at non-zero locations are stored in another data structure along with the source read and the position information. During overlap detection, the data structure of non-beginning k-mers is searched for reads sharing any of the k-mers at zeroth position. For n reads, at the most, there will be n beginning k-mers. A read is compared with only those reads that share its zeroth k-mer. Thus the number of read pairs compared is of the order of n.

Bliss detects overlap purely by searching the k-mer position data. The other advancement made by Bliss is by a factor of K. Although k-mers are generated by sliding a window of size K over the reads, during overlap detection there is no need to slide by a position of one. Since one shared k-mer means an overlap of K, the overlap detection can continue by advancing by K positions at each step. Thus it further improves the speed of overlap detection.

At this point of time, error correction is beyond the scope of this algorithm. It operates post error correction. In future when the quality of the sequencing data becomes near accurate, algorithm like Bliss will certainly a play key role in further enhancing the quality of assembly as it is able to trap all the real overlaps unlike the heuristics based approaches.

2. OVERLAP DETECTION ALGORITHM

Bliss takes a three phase approach towards overlap detection.

2.1 Phase I – k-mer Generation

In the first phase, k-mers are generated from each read by sliding a window of size K over the read. The zeroth k-mers, that is the ones occurring at the beginning of the reads, are put in a data structure Z along with the source read number. The other k-mers are added to another data structure T along with the source read number and the position in the read. K-mers are stored by their unique radix-4 number and the k-mer strings are discarded [10]. This brings in space saving and also substitutes string comparisons by number comparisons.

2.2 Phase II – Gather Read Pairs

In the second phase, read pairs which may possibly overlap are gathered in another data structure P. P holds the id of first read, the index at which overlap begins in the first read and the id of the second read. The overlap, if it is there, always begins at position zero for the second read hence the index is not stored separately for the second read.

Read pairs are gathered by iterating through the set Z. For each k-mer in the set Z, the set T is searched to extract reads that have this k-mer at some position in them. An entry is made in the set P of the id of the read corresponding to this shared k-mer in T, the corresponding position from T and the id of the read corresponding to this k-mer in Z.

If multiple entries of the same read pair are present in P, then they are processed in the ascending order of the position so that the largest overlap is tested for first.

2.3 Phase III – Filter out Read Pairs with Discontinuity in Overlap

For each entry in P, let r1 be the id of the first read i.e. the one that came from the set T, let p be the corresponding position and let r2 be the id of the second read i.e. from Z. Since the k-mer at index p in the first read, matches the zeroth k-mer in the second read, there is already an overlap of K characters between these reads. This overlap starts at position zero for the second read. We need to check if it continues till the last position in the first read. If not, this read pair is discarded. To check the continuity till the last character, we start at the last k-mer in the first read and the corresponding k-mer in the second read. If these k-mers match, we move backward by K positions in both the reads at each step until there are K or less characters left at the beginning of the second read. These characters are covered by the zeroth k-mer, hence there is no need to again match them here. So we just report overlap and break. Checking for continuity backwards starting from the end of the first read has a clear advantage of eliminating false overlaps quickly

without requiring any additional k-mer comparisons. This is another speed improvement in the process of overlap detection. The maximum number of k-mers compared i.e. before reporting overlap in this phase is (ceil(((l – p)/K)) – 1) where l is the length of r1.The last k-mer position is computed for the first read by subtracting K from the read length. If k-mers in both the reads are not identical at any step, there is discontinuity in overlap so we break and move to the next read pair

2.4 Algorithm

Input: R: {r} Set of reads
K: k-mer length.

Output: P: {(r_1, i, r_2)}. r_1 and r_2 overlap from i to (l – 1) in r_1 where l = length(r_1) and from 0 to (l – i - 1) in r_2.

Data Structures:

T: Set of 3-tuples (k_{mn}, r_m, n) k-mer k_{mn} occurs in read r_m at n^{th} position where n != 0
Z: Set of 2-tuples (r_m, k_{m0}) k_{m0} occurs in read r_m at 0^{th} position
P: Set of 3-tuples (r_1, i_1, r_2) Reads r_1 and r_2 share a k-mer at i^{th} position in r_1 and 0^{th} position in r_2. A possible overlap between r_1 and r_2 may begin here.

Phase I - Tokenization/K-mer generation

```
For each r_m in R,
{
   l_m = length (r_m);
   lastKmerIndex = ( l_m – K);
   for (i = 0, i <= lastKmerIndex, i++)
   {
      /* Get a substring of length K in r_m starting at i, Unique number is radix-4 representation of
      this string */
      k_mi = UniqueNumber (substring(r_m, i, K))
      If (i != 0)
      {
         Enter (k_mi, r_m, i) in T;
      }
      Else
      {
         Enter (r_m, k_mi) in Z;
      }
   }
}
```

Phase II – Gather read pairs with possible overlap occurring at the end

For each r_m in (r_m, k_{m0}) in Z,

- *Fetch from T, (k_{sv}, r_s, v) where $k_{sv} = k_{m0}$ and (r_s != r_m)*
- *Enter in P (r_s, v, r_m)*

For multiple entries of r_s and r_m, (r_s, x, r_m) order by x ascending so that the largest overlap is tested for first

Phase III – Filter read pairs with discontinuity in overlap

```
For each (r_s, v, r_m) in P,
{
   l_s = length (r_s);
   lastKmerIndex = (l_s – K);
   i = lastKmerIndex;
   j = (lastKmerIndex – v);

   While (j > 0)
```

```
{
    Search T for tuples (k_si, r_s, i), (k_mj, r_m, j) where k_si = k_mj
    If (found)
    {
        i = i – K;
        j = j – K;
    }
    Else  /* If (!found)  - Discontinuity in overlap */
    {
        Remove (r_s, v, r_m) from P;
    }
}  /* While */

If (j <= 0) /* Overlap detected, remove other entries for this read pair from P */
{
    For all q > v
        Remove from P (r_s, q, r_m);
}

} /* For Each */

Output P /* P contains all the overlapping read pairs */
```

3. DISCUSSIONS

Bliss, by segregating zeroth k-mers, greatly reduces the number of comparisons. It only considers full overlaps for both the reads. Partial overlaps are partly discarded in the first step by this segregation. The rest of them are removed in the third phase. The number of k-mers looked up for read pair (r_s, v, r_m) before reporting overlap are (ceil(((l – v)/K)) – 1), where *l* is the length of the first read r_s and the overlap begins at index v for the first read.

The data structures need to be indexed to facilitate faster fetch. Suggested indexes are: set T {(k_{mn}, r_m, n)} has two indexes: one on k and the other on (r, n), set P has index on (r_1, r_2, i) sorted ascending by i for multiple entries of the same read pair.

Bliss uses unique number representation of k-mers and discards k-mer strings. This further improves speed.

Consider the values given in Table 1.

K-mers	Occurring at position zero in reads - Z	Occurring at non-zero position in reads - T
k_0	r_1, r_{23}	$r_{13,8}$, $r_{12,5}$, $r_{13,22}$
k_1	…..	…..
k_2	…..	…..

Table 1: Example of Shared k-mers.

k-mer k_0 in Table 1 is at zeroth position in r_1, r_{23} and non-zero position for $r_{13, 8}$, $r_{12, 5}$, $r_{13, 22}$. For each row, we compare each read in set Z with each entry in set T. Thus, the pairs compared for k-mer k_0 are (r_1, r_{13}), (r_{23}, r_{13}) starting from 8th position in r_{13}, (r_1, r_{12}), (r_{23}, r_{12}) starting from 5th position in r_{12} and (r_1, r_{13}), (r_{23}, r_{13}) starting from 22nd position in r_{13}. Thus there are 6 comparisons done for k_0.

Figure 1 shows the k-mers compared for (r_s, v, r_m). In phase 3 of the algorithm, Bliss iterates for r_s through i = v + 2K, v + K and for r_m through j = 2K, K.

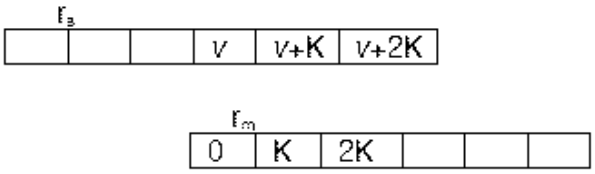

FIGURE 1. k-mers Compared For Detecting Overlap Between Reads r_s and r_m.

4. Number of Comparisons

Let N be the number of reads in the given set, L be average read length and K be k-mer length. Thus the total number of k-mers generated in the set is (L – K + 1) * N. The probability that read R_x is not compared with read R_1 is $(1-1/((L – K + 1) * N))^{(L - K)}$. Hence the probability that R_x will be compared with R_1 is $1 – [(1-1/((L – K + 1) * N))^{(L - K)}]$. Thus Expected number of reads that will be compared with R_1 is $(N -1) * [1 – [(1-1/((L – K + 1) * N))^{(L - K)}]]$. The total expected number of comparisons thus becomes $N * (N -1) * [1 – [(1-1/((L – K + 1) * N))^{(L - K)}]]$. In this computation, let the total expected number of comparisons be N * X. The X above approaches 1 when the real data numbers are put in. Thus the number of comparisons approaches N.

The test data had 100 (N) reads of length 30 (L) and K was 6. The expected number of comparisons becomes: 95. The E.Coli set had 400000 reads with average length of 230. The K was set to 18. Thus N = 400000, L = 230 and K = 18. Thus the expected number of read pairs compared becomes: 398121.

In reality, all k-mers are not distinct or else there will be no overlap among reads. Thus the number of distinct k-mers under consideration is less than ((L – K + 1) * N) which slightly increases the number of comparisons and makes it approach closer to N from the lower side. E.g. consider an ideal case where each read has its last k-mer overlapping with the zeroth k-mer of exactly one read. The larger is the number of k-mers overlapping, the less is the number of distinct k-mers present. Similarly the more is the number of reads sharing the last k-mer, the less is the number of k-mers. As an example let's take up a case where only the last k-mer overlaps and that too with only one read. This will reduce the number of distinct k-mers by N making it (L – K) * N. With these many k-mers in the set, the number of pairs compared for test data and E.Coli above become 99 and 399999 respectively i.e. the numbers are truly linear. This case we considered is the maximum number of k-mers present in the set with overlap present. In reality multiple k-mers will overlap for many read pairs and many k-mers will be present in more than two reads. Thus the number of distinct k-mers will further reduce.

The raw approach would involve N * (N-1) comparisons which becomes 9900 and 159999600000 for test data and E.Coli respectively. The approach taken by some assemblers where two reads sharing any k-mers are compared will involve $N * (N -1) * [1 – [(1-1/((L – K + 1) * N))^{(L – K + 1)}]^{(L – K + 1)})]$ comparisons. For test data and E.Coli, these numbers become 2190 and 85177107 respectively. Clearly Bliss involves the least number of comparisons among these three approaches and yet reports all true overlaps.

5. IMPLEMENTATION AND RESULTS

Bliss can be implemented with the data structures T, Z, P as in-memory data structures or persistent data structures. The indexing and data retrieval infrastructure can be implemented from scratch for fine tuning and better optimization or some existing infrastructure can be utilized. I chose persistent data structures. Also, instead of reinventing the wheel, I decided to exploit the indexing and data retrieval facilities provided by a relational data engine to implement the data access layer. To utilize the overlaps detected for contig generation, I supplied it with a basic contig generation algorithm. The contig generation algorithm halts when it cannot extend the contig further or if there are more than one paths avilable for extension.

I implemented Bliss using MySQL database. The first phase of k-mer generation involves importing the reads from .fa file into the database and then generating k-mers from the reads. The second phase computation is saved by defining a view on the sets T and Z. The third phase iterates over the view to check for continuity of overlap in both the reads. The read pairs for which continuity exists are inserted in the read pairs' table. The contig generation algorithm operates on this read pairs' table to form contigs.

Implementing indexing and data retrieval from scratch to fine-tune to the problem at hand may boost performance compared to relying on a relational data engine. However it trades performance with complexity and hence the development effort.

Bliss was tested on experimental data and on single-end E. Coli data. For E. Coli, on a 2 GB, quad-core machine with 32 bit OS, Bliss took approximately 6 hours to output contigs.

Some of the database level optimizations done are as mentioned here. These MySQL parameters were optimized: innodb_buffer_pool_size was set to 1024M, key_buffer_size to 64M, table_open_cache to 4000, table_definition_cache to 4000. To improve performance, minimal required indexes were defined. Inserts, updates, deletes were done in bulk wherever possible. Foreign key constraints were removed. Large tables were partitioned.

Another in-memory optimization done in the third phase was to fetch all the k-mers for the given pair of reads at the given indexes in one shot. The k-mers from one read were bitwise XORed with their counterparts in the other read. The results were then bitwise ORed to get a single number. If this number is zero, it means all the k-mers match and there is true overlap between the read pair else there is no overlap.

6. CONCLUSION AND FUTURE SCOPE

Bliss is a simple overlap detection algorithm. It uses a hybrid approach of k-mer generation followed by read overlap detection. It significantly reduces the number of comparisons for overlap detection by segregating zeroth k-mers from the others. It detects the overlap by searching the k-mer position data. While detecting overlap, it advances in steps of K in backward direction to speed up the process.

Since there is no heuristics involved Bliss offers 100% accuracy. All the overlaps in a given set of reads are correctly reported by Bliss.

The data structures can be in-memory or persistent and may use custom indexing and data retrieval infrastructure. If implemented on a relational data engine, several database level optimizations may be done for performance.

Bliss can be extended to include single bit error correction and multiple bit error trapping by analyzing the sequence of differing k-mers between two reads.

When run post error correction, bliss reports all the overlaps present in the given set of reads.

7. ACKNOWLEDGEMENTS

My sincere thanks to Dr. S.D. Bhide and Prof. S. P. Gosavi for their support and valuable feedback.

8. REFERENCES

[1] D. R. Zerbino and E. Birney. "Velvet: Algorithms for de novo short read assembly using de Bruijn graphs." (2008) *Genome Research*, 18:821–829

[2] Jeffrey J. Cook, Craig Zilles. "Characterizing and Optimizing the Memory Footprint of De Novo Short Read DNA Sequence Assembly" April, 2009.

[3] Flicek P, Birney E. "Sense from sequence reads: methods for alignment and assembly." *Nat Methods*. 2009 Nov;6(11 Suppl):S6-S12.

[4] Ruiqiang Li, Hongmei Zhu, Jue Ruan, et al. (2009, December). "De novo assembly of human genomes with massively parallel short read sequencing", *Genome Research*, [Online].Available: http://genome.cshlp.org/content/early/2009/12/16/gr.097261.109.full.pdf+html

[5] Birol I, Jackman SD, Nielsen CB, Qian JQ, Varhol R, Stazyk G, Morin RD, Zhao Y, Hirst M, Schein JE, Horsman DE, Connors JM, Gascoyne RD, Marra MA, Jones SJ. "De novo Transcriptome Assembly with ABySS" *Bioinformatics* (2009) 25 (21): 2872-2877.

[6] Jared T. Simpson, Kim Wong, Shaun D. Jackman, Jacqueline E. Schein, Steven J.M. Jones, Inanc Birol. "ABySS: A parallel assembler for short read sequence data", (2009) 19(6):1117-23.

[7] Chengxi Ye, Zhanshan (Sam) Ma, Charles H. Cannon, Mihai Pop, Douglas W. Yu. SparseAssembler: de novo Assembly with the Sparse de Bruijn Graph

[Online].Available:

http://arxiv.org/ftp/arxiv/papers/1106/1106.2603.pdf

[8] Chengxi Ye, Charles H. Cannon, Zhanshan (Sam) Ma, Douglas W. Yu, Mihai Pop. SparseAssembler2: Sparse k-mer Graph for Memory Efficient Genome Assembly. [Online].Available:

http://arxiv.org/ftp/arxiv/papers/1108/1108.3556.pdf

[9] Serafim Batzoglou, David B. Jaffe, Ken Stanley, et al. "ARACHNE: A Whole-Genome Shotgun Assembler" (2002) *Genome Research*, 12:177–189.

[10] Sayali Davalbhakta. "Throwing Away k-mer Strings" Submitted for publication.