IDBA - A Practical Iterative de Bruijn Graph De Novo Assembler

Yu Peng, Henry Leung, S.M. Yiu, Francis Y.L. Chin

Department of Computer Science, The University of Hong Kong
Pokfulam Road, Hong Kong
{ypeng, cmleung2,smyiu,chin}@cs.hku.hk

Abstract. The de Bruijn graph assembly approach breaks reads into *k*-mers before assembling them into contigs. The string graph approach forms contigs by connecting two reads with *k* or more overlapping nucleotides. Both approaches face the problem of false-positive vertices from erroneous reads, missing vertices due to non-uniform coverage and branching due to erroneous reads and repeat regions. A proper choice of *k* is crucial but for any single *k* there is always a trade-off: a small *k* favors the situation of erroneous reads and non-uniform coverage, and a large *k* favors short repeat regions.

We propose an iterative de Bruijn graph approach iterating from small to large *k* capturing merits of all values in between. With real and simulated data, our IDBA algorithm is superior to all existing algorithms by constructing longer contigs with similar accuracy and using less memory. The running time of IDBA is comparable with existing algorithms.

Availability: IDBA is available at http://www.cs.hku.hk/~alse/idba/

Keywords: De novo assembly, de Bruijn graph, string graph, mate-pair, high throughput short reads

1 Introduction

Despite tremendous research efforts, de novo assembly remains an incompletely solved problem. Although more reference genomes are known, de novo assembly remains a critical step in studying a genome. Applications such as detection of structural variations [1] cannot be done easily based on resequencing techniques and there are findings that show genome assembly based on resequencing may produce errors especially for species with high mutation rates [1].

With high throughput sequencing technologies (e.g. Ilumina Genome Analyzer and Applied Biosystems SOLiD), mate-pair short reads (35nt to 75nt) for a mammalian genome can be generated in a few weeks at a low cost. As short reads have different characteristics (i.e. shorter length, higher coverage, but relatively higher error rates) as compared to traditional Sanger reads, new assembly tools have emerged [2-11]. The first batch of tools (e.g. SSAKE [3], VCAKE [4], SHARCGS [5]) uses the "overlap-then-extend" idea but need to rely on data structures such as prefix trees, so they require lots of memory and run very slowly. The newer tools are divided into those

based on the de Bruijn graph (e.g. Velvet [7], Abyss [8], Euler-SR[2, 6], AllPaths[11]) and those based on the string graph (e.g. Edena [9]). Each of the two approaches has merits and limitations, and it is not clear which is better.

De Bruijn graph algorithms [7-8, 12-13] assemble reads by constructing a de Bruijn graph in which each vertex represents a length-k substring (k-mer) in a length-l read and connects vertex u to vertex v if u and v are *consecutive k-mers* in a read, i.e. the last (k – 1) nucleotides of the k-mer represented by u is the same as the first (k – 1) nucleotides of the k-mer represented by v. Intuitively, maximal paths of vertices without branches in the graph correspond to contigs to be outputted by algorithms.

String graph algorithms [9, 14] represent each read by a vertex and there is a directed edge from vertex u to vertex v if the suffix of at least x nucleotides of read u is the same as the prefix of read v. Similar to de Bruijn graph algorithms, string graph algorithms report maximal paths without branches as a contigs.

When the reads are error-free with high coverage, most tools work well. However, because of repeats, erroneous reads, and non-uniform coverage, their performances have not always been satisfactory. In this paper, we focus on three major problems: (1) *false positive vertices* (due to errors in reads); (2) *gap problem* (due to non-uniform or low coverage) and; (3) *branching problem* (due to repeats or errors in reads).

Three major problems:

(a) *False Positive Vertices*: Errors in reads introduce false positive vertices which make both graphs bigger and consume more memory; for example, for the human genome with 30x coverage, the memory requirement of Velvet [7] and Abyss[8] is more than 250G.

(b) *Gap problem*: Due to non-uniform or low coverage, reads may not be sampled for every position in the genome. For the de Bruijn graph, when all the (possible l – k) reads covering consecutive k-mers are missing, we may have short "dead-end" paths. The larger the k, the more serious is the gap problem. The same applies to the string graph if all the (possible l – x) reads following another read are missing.

(c) *Branching problem*: Those k-mers which connect with multiple k-mers due to repeat regions or erroneous reads introduce branches in the de Bruijn graph. Many algorithms [7-9] stop the contigs at branches and it is not possible to extend a contig without additional information. A small k will lead to more branches. The same branching problem occurs in string graph algorithms, and depending on the value of x (the number of overlapping nucleotides for two consecutive reads), the same read can be connected with multiple other reads.

Existing assembly algorithms:

Table 1 summarizes the major techniques used by existing algorithms to solve the above three problems. There are two methods for handling false positive vertices. (1) *Dead-end removal*: False positive vertices usually lead to short dead-end paths. Both de Bruijn and string algorithms (e.g. [7-8]) remove false positive vertices by removing these paths. However, due to the gap problem, some paths may be removed by mistake. (2) *Filtering*: De Bruijn graph algorithms remove false positive vertices if the corresponding k-mers appear no more than m times. However, some correct k-mers with low coverage might also be removed especially for large k for which the expected k-mer occurrence frequency is low. As for string graph algorithms, the

expected occurrence of each read is also low (1 or 2) and they rely on error correction which fall back to consider the multiplicity of *k*-mers [15] to correct errors in each read before forming contigs.

Problems	Techniques	de Bruijn graph	String graph
1) False positive vertices	(i) Dead-end removal	Yes	Yes
	(ii) Filtering	Yes (not effective if *k* is large)	Not applicable (relies on error correction algorithms)
2) Gap	No effective method (try to use a reasonable small *k* or *x*)	--	--
3) Branching	(i) Using read information	Yes	Not applicable (already use the whole read information)
	(ii) Bubble removal	Yes	Yes

Table 1: Major techniques used to handle the three problems.

There is no effective method to deal with the gap problem except all algorithms try to avoid gaps by using a large *k* (or *x* in string graphs).

Some de Bruijn graph algorithms [12] solve the branching problem by considering only those branches that are supported by reads. However, this method may easily lead to erroneous contigs [12] if the reads are erroneous especially when error rates are high. This method cannot be applied to string graph algorithms as they already consider the read information. The other technique is *bubble removal*, which is used by both approaches [7-9] and tries to merge similar paths of very similar vertices into *one path* as the small differences may only be due to SNPs or errors. However, the merging might be incorrect and this process increases the length of contigs at the expense of their accuracy.

	High Coverage Low Error Rate (100x, 0.5%)	Low Coverage Low Error Rate (30x, 1%)	High Coverage High Error Rate (100x, 2%)	Low Coverage High Error Rate (30x, 2%)
Edena (string Graph)	63256	5104	53491	147
Velvet (de Bruijn Graph)	61204	19284	32596	9424
Abyss (de Bruijn Graph)	58678	22109	50009	10992
IDBA (our algorithm)	63218	63218	59287	32612

Table 2: Performance (N50) of three existing assembly algorithms (Edena, Velvet, Abyss) against IDBA under different coverage and error rates for the simulated dataset using E.coli where read length is 75nt. The best results generated are used for comparison.

To summarize, string graph algorithms do not have an effective method to remove errors from reads and have the gap problem if *x* is set to a reasonable value to avoid the branching problem. However, string graph algorithms, which make use of the direct information in the whole read, perform very well in case of high coverage and low error rate. For other cases, de Bruijn graph algorithms may perform better. Our observations are confirmed by the N50 comparison of Edena [9] (currently one of the best string graph algorithms), Velvet [7] and Abyss [8] (the best de Bruijn graph algorithms) based on different coverage and error rates of the data as shown in Table 2 (more details on the comparison can be found in Section 4.)

The best existing assembly algorithms are Edena (string graph based), Velvet and Abyss (both de Bruijn graph based, differing in the exact details for handling dead-

ends and bubble removal). However, setting the correct parameter k in de Bruijn graph algorithms (or x in string graph algorithms) is crucial. Not only does the choice affect the filtering but also, for any single k (or x) value, there is a trade-off between the gap problem and the branching problem. In order to minimize the number of gaps, a smaller k (or x) should be used. But with a small k (or x), the branching problem becomes more serious. Existing algorithms usually pick a moderate value for k (or x) to balance between the two problems. None of the existing approaches try to take advantage of using different k (or x) values[1].

Our contributions:

We propose a new assembly algorithm (IDBA) based on the de Bruijn graph. The idea is simple but practical in that it alleviates the difficulties in setting a correct k and the filtering threshold m, gives good results, uses much less memory (many existing tools require huge amount of memory making them infeasible for large genomes) at the expense of a reasonable increase in running time. Instead of using a fixed k, our algorithm iterates from small to large k (k_{min} to k_{max}) capturing the merits of all values in between. The key step is to maintain an accumulated de Bruijn graph to carry useful information forward as k increases. Note that this is not the same as running the algorithm for many different k values independently as it is not clear how to combine contigs from different runs to get a better result. We show theoretically that the accumulated de Bruijn graph can capture good contigs and these contigs can be made longer as k increases. Based on experiments on simulated and real data, we show that IDBA can produce longer contigs (see Table 2 for the N50 comparison) with similar accuracy (very few wrong contigs and high coverage). More detailed results are presented in Section 4.

We are able to reduce the memory consumption by 70-80% as compared to existing algorithms which use a fixed k of moderate size. Because k is of moderate size, the algorithms cannot do filtering in the first step if the coverage is not high and thus create a big graph due to false positive vertices. However, we can start with a small k. With conservative but effective filtering in the first step (e.g. set $m = 1$), many false positive vertices are pruned. Although IDBA iterates through different k values, with implementation tricks, IDBA runs a lot faster than Abyss and is comparable with other existing algorithms.

Organization of the paper and remarks:

We organize the paper as follows. In Section 2, we introduce our algorithm IDBA and show the advantages of using small and large k values. Also, we provide key implementation details which help to reduce the memory consumption and running time. Section 3 compares the performance of IDBA with existing algorithms on both simulated and real data. We conclude the paper in Section 4.

We note that using mate-pair information to resolve repeats that are longer than reads is another important aspect of an assembly tool. In this paper, we mainly focus on short repeats, which account for the largest portion of repeats in genomes and

[1] The SHARCGS [5] algorithm uses fixed x values (the number of overlapping nucleotides) when extending a read, but they repeat the whole assembling procedure *independently* using a few different x values and combine the resulting contigs from different runs only.

cannot be resolved by mate-pair information easily as the error in the insert size may be even larger than the length of the repeat. We leave the problem of how to use mate-pair information for assembly for future study. So, in the last step of our assembly tool, which uses mate-pair information to connect the contigs, we simply follow Abyss [8]. Note also that, although our approach can be applied to the string graph with a range of x values, currently there is no effective way to remove errors from reads for string graphs, and so we focus on de Bruijn graphs.

2 Algorithm IDBA

In this section, we present our algorithm IDBA (Iterative De Bruijn Graph short read Assembler). Given a set of reads, we denote the de Bruijn graph for any fixed k as G_k. Instead of using only one fixed k, IDBA iterates on a range of k values from $k = k_{\min}$ to $k = k_{\max}$ and maintains an *accumulated de Bruijn graph* H_k at each iteration. At the first step when $k = k_{\min}$, H_k is equivalent to the graph G_k after deleting all vertices whose corresponding k-mers appear no more than m times (we set $m = 1$ or 2 in practice depending on coverage) in all reads. Theorem 1 shows that these k-mers are very likely to be false positives.

To construct H_{k+1} from H_k, we first construct potential contigs in H_k by identifying maximal paths $v_1, v_2, \ldots, v_p$ in which all vertices have in-degree and out-degree equal to 1 except v_1 and v_p which may have in-degree 0 and out-degree 0, respectively. We remove all reads that can be represented by potential contigs in H_k i.e. those reads that are substrings of a contig (as these reads cannot be used to resolve any branch). In the construction of H_{k+1}, we only consider the remaining reads and the potential contigs in H_k. We perform two steps to convert H_k to H_{k+1}. (1) For each edge (v_i, v_j) in H_k, we convert the edge into a vertex (representing a $(k+1)$-mer $x_{i1}\, x_{i2} \ldots x_{ik}\, x_{jk} = x_{i1}\, x_{j1} \ldots x_{jk}$). (2) We connect every two such vertices by an edge if the corresponding two consecutive $(k+1)$-mers have support from one of the remaining reads or potential contigs of H_k, i.e. the corresponding $(k+2)$-mer exists.

Note that in practice, we do not need to go from k to $k+1$; we can jump from k to $k+s$, in which case, for (1), we convert each path of length s in H_k into a vertex. In Theorem 4 in the Appendix, we show that by setting $s = 1$, we may get high quality contigs. The choice of s will represent a trade-off on the efficiency of the algorithm and the quality of the contigs.

For each H_k, we follow other algorithms [7] to remove dead-ends (potential contig shorter than $3k - 1$ with one end with 0 in-degree or out-degree, which represents a path in H_k of length at most $2k$). Note that removing a dead-end may create more dead-ends, the procedure will repeat until no more dead-ends exist in the graph. These dead-end contigs are likely to be false positives (Theorem 2). In fact, most of the remaining false positive vertices after the first filtering step can be removed as dead ends. After obtaining $H_{k_{\max}}$, we merge *bubbles* where bubbles are two paths representing two different contigs going from the same vertex v_1 to the same vertex v_p where these two contigs differ by only one nucleotide. This scenario is likely to be caused by an error or a SNP. Like other algorithms [7-9], we merge the two contigs

into one. Then, finally we use mate-pair information to connect the contigs as much as possible using a similar algorithm as Abyss[8] and report the final set of contigs.

Algorithm IDBA:

```
1   k <- k_min (k_min = 25 by default)
2   Filter out k-mers appearing <= m times
3   Construct H_{k_min}
4   Repeat
5       a) Remove dead-ends with length < 2k
6       b) Get all potential contigs
7       c) Remove reads represented by potential contigs
8       d) Construct H_{k+s} (s = 1 by default)
9   Stop if k >= k_max (k_max = 50 by default)
10  Remove dead-end with length shorter than 2k_max
11  Merge bubbles
12  Connect potential contigs in H_{k_max} using mate-pair information
13  Output all contigs
```

First, we note that the probability of removing a true positive vertex in our filtering step is very low (Theorem 3 in Appendix A.3 gives the analysis) as long as $k_{\min}$ and the filtering threshold m are set to a reasonable value (e.g. $m = 1$). For example, if 1.6×10^6 length-75 reads are sampled from a genome of length 4.1×10^6 (45x coverage) with error rate 1%, the probability of filtering out a true positive vertex in H_{25} is 1.14×10^{-9}, i.e. the expected number of false negative vertices is 0.0047 << 1 which is very small. Even for some cases the number is large, say 10, it is still relatively very small when compared with the genome size. Thus, for simplicity in analysis, we assume there is no false negative vertex in $H_{k\min}$. The filtering step can remove a large portion of the false positive vertices. Most of the remaining false positive vertices are removed in later steps by dead-ends The probability of removing a correct contig as a dead-end is also small (see Theorem 4 in Appendix A.3 for the exact calculation of the probabilities). The probability of determine a dead-end wrongly is 2.46×10^{-4} only when the above example is considered.

A contig that appears in G_k for a small k may not be a contig in $G_{k'}$ with $k' > k$ due to the gap problem. However, in our construction, if a contig c appears in H_k, there must be a contig c' in $H_{k'}$ containing c (Theorem 1). That is, the contig information is carried over from H_k to H_k'. As k increases, more branches can be resolved while the gaps solved by previous iterations will be preserved in the current H_k.

Theorem 1: Assume that $k_{\min} = k$ and $k < k'$. If there is a contig c in G_k of length at least $3k_{\max} – 1$ with all vertices are true positive. there must be a contig c' in $H_{k'}$ such that c' contains c.

*Proof***:** By induction on k. Let $k' = k + 1$ and $c = x_1x_2\ldots x_{p+k-1}$ be a contig in H_k represented by the path $p = (v_1, v_2, \ldots, v_p)$, all vertices $v_1, v_2, \ldots, v_p$ have in-degree and out-degree ≤ 1, it is easy to see that the path $p' = (v_1', v_2', \ldots, v_{p-1}')$ in H_{k+1} where each $(k+1)$-mer $v_i' = x_ix_{i+1}\ldots x_{i+k}$ also has in-degree and out-degree ≤ 1. As the length of the contig represented by path $p' \geq 3k_{\max} – 1$, there must be a contig including path p, i.e. c, in H_{k+1}. □

Corollary: $H_{k_{\max}}$ must contain all contigs in $G_{k_{\min}}$ of length at least $3k_{\max} - 1$ with all vertices are true positive.

In practice $H_{k_{\max}}$ always contains longer contigs than $G_{k_{\min}}$ by resolving branches at each iteration. By iterating the graph H_k towards larger k, we may get longer and longer contigs as some of the branches (e.g. length-k repeat region (Case 1) and error branches in H_{k+1} (Case 2)) may be resolved when using a larger k.

Case 1: Let $c_1 = s_1v_rs_2$ and $c_2 = s_3v_rs_4$ be two substrings in the genome where v_r is a common length-k substring representing a repeat region, s_1, s_2, s_3, s_4 are different substrings. c_1 and c_2 are represented by five contigs in H_k as the k-mer v_r has in-degree of 2 and out-degree of 2. If there are two correct reads containing v_r and its 2 neighboring nucleotides at both ends in c_1 and c_2 respectively, and there is no error read containing $s_1v_rs_4$ or $s_3v_rs_2$, then there must be two contigs, one containing c_1 and the other containing c_2 in H_{k+1}.

Case 2: Let c be a contig in H_k that stops before vertex u whose in-degree is 1 and out-degree is >1. Assume that among all branches of u, only u to v is correct. If there is a correct read containing u and its 2 pairs of neighboring nucleotides at both ends and there is no error read linking c with other branches, there will be a longer contig c' in H_{k+1} that contains c.

Theorem 2: If there is a contig c in G_k of length at least $3k_{\max} - 1$ with all vertices are true positive which satisfies case 1 or case 2 in $H_{k'}$, $k = k_{\min} \leq k' < k_{\max}$, there is a longer contig c' in $H_{k_{\max}}$ contains c.

In the algorithm, we increase the value of k by 1 at each iteration, i.e. $s = 1$. Theorem 5 in Appendix A.3 shows that for a better quality of the contigs, this is essential. On the other hand, as a trade-off between the efficiency of the algorithm and the quality of the contigs, it is possible to set $s > 1$, i.e. to increase the value of k by more than 1 at each iterative step.

2.1 Implementation details

The memory used by IDBA is only about 20-30% of that used by the other existing tools because 80% of false positive vertices are removed in the filtering step and IDBA uses a compact hash table to represent De Bruijn graph implicitly with each edge is represented by one bit only.

Although IDBA construct $H_{k_{\max}}$ from $H_{k_{\min}}$ step by step, the running time of IDBA is not directly proportional to the number of k values between $k_{\max}$ and $k_{\min}$. According to Theorem 3, a contig in H_k is also a contig in H_{k+1}, thus IDBA only needs to check whether a branch in H_k can be resolved in H_{k+1}. Since reads represented by a contig is removed in each iteration, the number of reads considered in each step is decreasing, e.g. half of the reads are removed when constructing $H_{k_{\min+1}}$. In practice, IDBA runs much faster than Abyss, and three times slower than Velvet.

3 Experimental Results

The genome of Escherichia coli (O157:H7 str. EC4115) from NCBI [16] is used for simulated experiments (the genome length is 5.6 M). Reads are randomly sampled uniformly with coverage 30x. In our experiments, we generated reads with error rates 1%, read length 75 and insert distance 250. Note that we have repeated the experiments using other coverage (e.g. 50x, 100x), error rates (e.g. 2%) and read length (e.g. 50). The results are similar, so we only show the result for 30x coverage with 1% error on length-75 reads. We also use a real data set, namely bacillus subtilis, to evaluate our algorithm. The length of the genome is 4.1M. The reads are sequenced using Solexa machine with coverage 45x, read length 75 and insert distance 400. The estimated error rate is about 1%.

3.1 Simulated data

We compare the performance of Velvet[2], Abyss, Edena and our algorithm IDBA on the simulated data based on different k values (or x values). For IDBA, we fix k_{min} = 25, m = 1 and compare the performance of IDBA with different k_{max}. For the other algorithms, defaults are used for other parameters. We also plot the upper bound that can be achieved by an ideal de Bruijn graph with no false positive or false negative vertices and edges.

We only consider valid contigs which are longer than 100 bps and can be aligned to the reference with 99.9% similarity. Figure 1(a) shows the comparison of the softwares based on N50. As we mentioned in Section 1, existing assembly algorithms have many false positive vertices and branching problems when k is small and they have many gaps when k is large. Thus these algorithms have the best performance (largest N50) for in-between values of k (the optimal k for Velvet, Edena and Abyss are 30, 40 and 40 respectively in this data set). However, since IDBA consider a range of k values, its performance is better than the others even when considering a range of 10 values for k (k_{min} = 25 and k_{max} = 35). Furthermore, when IDBA considers a larger range for k (k_{min} = 25 and k_{max} = 50), its performance is close to the upper bound. We have only 10 false positive contigs when setting k_{min} = 25 and k_{max} = 50 while Abyass, Velvet and Edena produce 66, 19 and 650 false positive contigs respectively.

	Time	Memory	k	Contigs				Coverage
				Number	N50	Max length	Wrong contigs (total len.)	
Velvet	155s	1641M	30	1369	19284	96905	19 (9813)	94.57%
Edena	957s	678M	40	4672	5104	46908	650 (72019)	97.22%
IDBA	371s	360M	25 – 50	1550	63218	217365	10 (3935)	97.50%
IDBA-pe	412s	360M	25 – 50	952	92207	217365	34 (108625)	94.58%
Abyss	1114s	1749M	40	1390	22109	87118	66 (34998)	95.05%
abyss-pe	1237s	1749M	40	484	59439	226626	186 (352437)	91.39%
upper-bound	--	--	50	1561	63218	217365	0 (0)	99.11%

Table 3. Statistics of optimal (w.r.t. N50) result of each algorithm for simulated data

[2] Velvet cannot handle k larger than 31; we implemented Velvet algorithm by ourselves for the comparison. Note that since Abyss performs better than Velvet, we did not implement the pair-end version of Velvet.

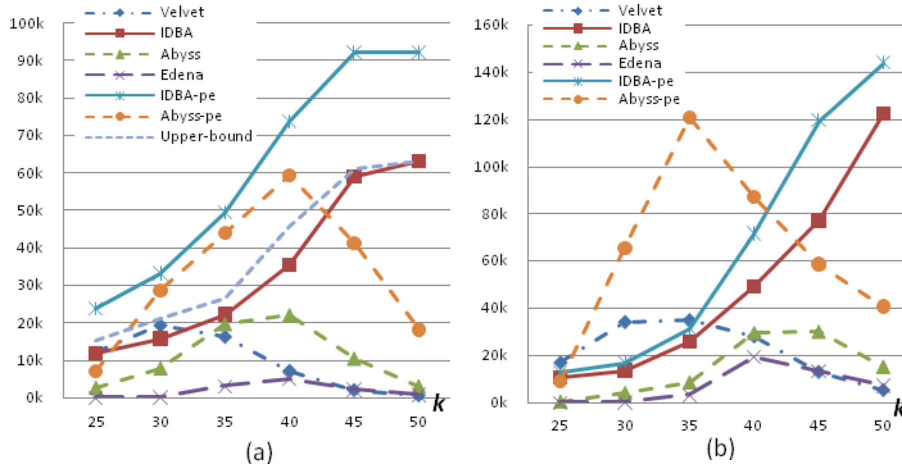

Fig. 1. (a) N50 for contigs produced by assembly algorithms with different *k*-values (*x*-values if the software is string graph based) on simulated data using E.coli as the reference genome where read length is 75nt, coverage is 30x and error rate is 1%. (IDBA-pe and abyss-pe are the results for using mate-pair information to extend the contigs while Edena does not use mate-pair information) (b) N50 for contigs produced by assembly algorithms with different *k* (or *x*) values on real data from bacillus subtilis where read length is 75nt, coverage is 45x and error rate is 1%.

For IDBA and Abyss, we also apply the pair end information to connect the resulting contigs to make them longer. The results are shown in the same graph (IDBA-pe and abyss-pe). Note that as *k* increases, the N50 may drop when applying the mate-pair procedure since more branches have been resolved incorrectly and some short contigs are removed as dead-ends. In fact, further research is required on how to use mate-pair information effectively for assembly. The pair-end version of Abyss has optimal result when *k* is 35 while IDBA has optimal result when *k* is 45.

Table 3 shows a comprehensive statistics on the performance of the algorithms on their optimal *k* values (w.r.t. N50). IDBA produced much longer contigs than all other algorithms. When mate-pair information is not available, the N50 of IDBA (63218) is about three times that of the next best algorithm (22109 by Abyss) and is the same as the upper bound. IDBA also produced the fewest number of wrong contigs (a contig which cannot be aligned to the reference genome with 99.9% similarity) and the total length of all wrong contigs is only about 4000nt which is half of the next best algorithm (Velvet). The coverage of IDBA is also the best among all algorithms. Since IDBA performs well on assembling single end reads, it outperforms other algorithms when considering mate-pair information. To conclude, IDBA outperforms other algorithms substantially and produce much longer contig with high accuracy.

3.2 Real data

Figure 1(b) shows the N50 of the contigs produced by Velvet, Abyss, Edena and our algorithms IDBA on the real reads from bacillus subtilis using different *k* values (*x*

values). Since the reads may not be uniformly sampled in the real data set, we usea smaller k_{min} (20nt) and keep $m = 1$ to run IDBA. For the other algorithms, we use their default parameters except for k. We do not have the reference genome to check if a contig is valid. We calculate the N50 for all reported contigs longer than 100bp. Note that the result may not be accurate, because some algorithms may produce longer but invalid contigs. The results are consistent with that of the simulated data. Velvet, Edena and Abyss get their best performance when k = 35, 40 and 45 respectively. IDBA can keep improving the result while k_{max} is increasing.

In this data set, pair end information is not so useful for IDBA because using read information can already solve most of the branches. When using k_{max} equal to 50, the pair end version IDBA is only slightly better than single end version. The performance of pair end version Abyss has similar performance as in simulated data. Its optimal k is 35, and the longest N50 it produces is 30% shorter than IDBA. In conclusion, IDBA produced the longest contigs among all algorithms. A detailed comparison is given in Table 4 in Appendix A.2.

3.3 Running time and memory consumption

Besides Abyss (12.8 minutes – 7 hours for simulated data and 10 minutes – 1.2 hours for real data depending on the value of k), the running time of other algorithms are more or less the same. Abyss runs much slower when k is small, probably due to its slow procedure for dealing with graphs with many false positive vertices. Velvet (120 – 190 seconds for simulated data and 80 – 100 seconds for real data) is the fastest among all algorithms. IDBA (150 – 270 seconds for simulated data and 190 – 320 seconds for real data) runs faster than Abyss and is about three times slower than Velvet. Refer to Figures 2 and 3 in the Appendix for details.

The memory consumption is about the same for different k values across the existing algorithms. Abyss and Velvet require about 2G bytes of memory for simulated data and 1G memory for real data. IDBA only requires about 400M and 300M respectively because 80% of false positive vertices are removed in the first filtering step with only 8 25-mers are removed incorrectly (matches with expected number 8.88 calculated in Theorem 3). So, the memory consumption of IDBA is only about 20 – 30% of the existing De Bruijn graph tools. Edena consumes less memory than Abyss and Velvet because the number of reads is small, but still double the amount used by IDBA. Refer to Tables 3 in Section 3.2 and Table 4 in the Appendix for details.

4 Conclusions

Our IDBA algorithm, based on de Bruijn graphs, can capture the merits of all k values in between $k_{\min}$ and $k_{\max}$ to achieve a good performance in producing long and correct contigs. Because the initial filtering step removes many false positive k-mers and the number of reads considered at each iterative step is reduced, the required memory and running time is much reduced. Though an accumulated de Bruijn graph is maintained at each iterative step, the running time is comparable with the existing algorithms. In

fact, this running time can be further reduced if, say, one or two k values are skipped at each iterative step. In practice, the quality of the result is only slightly affected by the skipping of values, in exchange for shorter running time.

Our next target is to investigate how to better use mate-pair information for resolving long repeats in order to produce even longer and more accurate contigs.

References

1. Wang, J., et al., *The diploid genome sequence of an Asian individual.* Nature, 2008. **456**(7218): p. 60-5.
2. Chaisson, M.J., D. Brinza, and P.A. Pevzner, *De novo fragment assembly with short mate-paired reads: Does the read length matter?* Genome Res, 2009. **19**(2): p. 336-46.
3. Warren, R.L., et al., *Assembling millions of short DNA sequences using SSAKE.* Bioinformatics, 2007. **23**(4): p. 500-1.
4. Jeck, W.R., et al., *Extending assembly of short DNA sequences to handle error.* Bioinformatics, 2007. **23**(21): p. 2942-4.
5. Dohm, J.C., et al., *SHARCGS, a fast and highly accurate short-read assembly algorithm for de novo genomic sequencing.* Genome Res, 2007. **17**(11): p. 1697-706.
6. Chaisson, M.J. and P.A. Pevzner, *Short read fragment assembly of bacterial genomes.* Genome Res, 2008. **18**(2): p. 324-30.
7. Zerbino, D.R. and E. Birney, *Velvet: algorithms for de novo short read assembly using de Bruijn graphs.* Genome Res, 2008. **18**(5): p. 821-9.
8. Simpson, J.T., et al., *ABySS: a parallel assembler for short read sequence data.* Genome Res, 2009. **19**(6): p. 1117-23.
9. Hernandez, D., et al., *De novo bacterial genome sequencing: millions of very short reads assembled on a desktop computer.* Genome Res, 2008. **18**(5): p. 802-9.
10. Chaisson, M., P. Pevzner, and H. Tang, *Fragment assembly with short reads.* Bioinformatics, 2004. **20**(13): p. 2067-74.
11. Butler, J., et al., *ALLPATHS: de novo assembly of whole-genome shotgun microreads.* Genome Res, 2008. **18**(5): p. 810-20.
12. Pevzner, P.A., H. Tang, and M.S. Waterman, *An Eulerian path approach to DNA fragment assembly.* Proc Natl Acad Sci U S A, 2001. **98**(17): p. 9748-53.
13. Idury, R.M. and M.S. Waterman, *A new algorithm for DNA sequence assembly.* J Comput Biol, 1995. **2**(2): p. 291-306.
14. Myers, E.W., *The fragment assembly string graph.* Bioinformatics, 2005. **21 Suppl 2**: p. ii79-85.
15. Chin, F.Y., et al., *Finding optimal threshold for correction error reads in DNA assembling.* BMC Bioinformatics, 2009. **10 Suppl 1**: p. S15.
16. http://www.ncbi.nlm.nih.gov/.

Appendix

A.1 Running Times of the Assembling Algorithms

Figure 2 and Figure 3 show the running time of IDBA and existing assembling algorithm, Velvet, Abyss and Edena on the simulated data set and the real data set. From the figures, we can see that IDBA has similar running time as other assembling algorithm except Abyss which takes a very long time when k is small due to a complicated method for removing dead-ends.

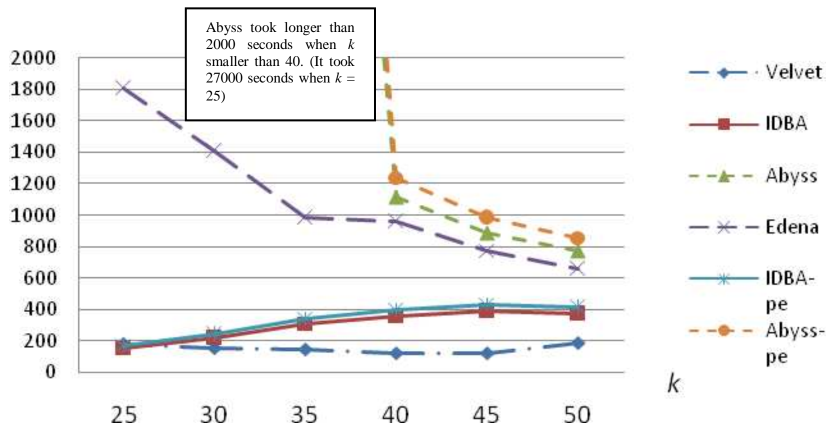

Fig. 2. Running time of assembly algorithms with different k (or x) values on simulated data.

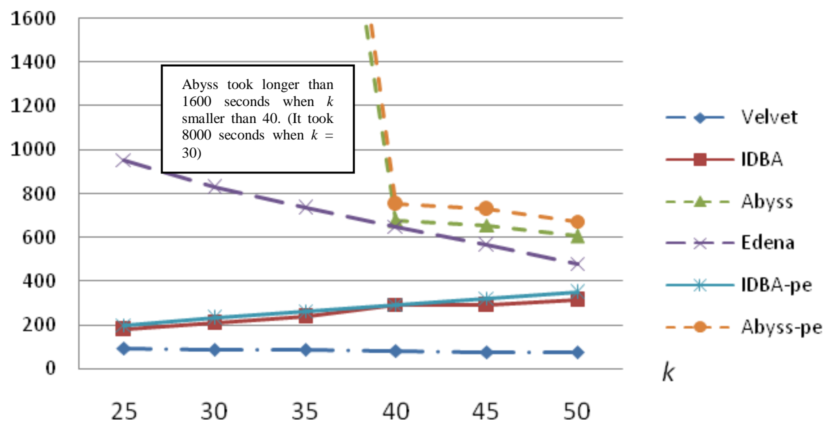

Fig. 3. Running time of assembly algorithms with different k (or x) values on real data.

A.2 Detailed comparison of the Assembly Algorithms for real data

In Table 4, we show comprehensive statistics on the performance of the algorithms on their optimal k value (w.r.t. N50) for the real dataset. IDBA produced much longer contigs than all other algorithms no matter whether the single-end or the pair-end version is used. The result is consistent with that of the simulated dataset.

	Time	Memory	k	Contigs Total No.	N50	Max length
Velvet	89s	893M	35	1369	35136	164023
Edena	649s	632M	40	4672	19423	66455
IDBA	314s	310M	25 – 50	1550	122574	602412
IDBA-pe	349s	310M	25 – 50	952	143987	602609
Abyss	729s	923M	40	1390	30081	134067
Abyss-pe	3766s	936M	35	484	120807	537397

Table 4. Statistics for the optimal (w.r.t. N50) result of each algorithm for real data.

A.3 Theorems and proofs

Theorem 3: Assume m is the filtering threshold, the probability that a $k_{\min}$-mer v in the genome (except the first $l - k_{\min}$ and last $l - k_{\min}$ $k_{\min}$-mer in the whole genome) does not appear in $H_{k\min}$ (false negative) when t length-l reads are uniformly sampled from a length-g genome with error rate e is at most $\sum_{i=0}^{m}\binom{t}{i}p^i(1-p)^{t-i}$ where $p = [(l-k_{\min}+1)/(g-l+1)]\cdot(1-e)^{k_{\min}}$.

Proof:

$$\begin{aligned}
&\Pr(v \text{ is sampled in a read})\\
&= \Pr(\text{read contains } v \text{ is sampled})\Pr(v \text{ is sampled} \mid \text{read contains } v \text{ is sampled})\\
&\quad + \Pr(\text{read does not contain } v \text{ is sampled})\Pr(v \text{ is sampled} \mid \text{read does not contain } v \text{ is sampled})\\
&\ge \Pr(\text{read contains } v \text{ is sampled})\Pr(v \text{ is sampled} \mid \text{read contains } v \text{ is sampled})\\
&\ge \frac{l-k_{\min}+1}{g-l+1}\cdot(1-e)^{k_{\min}}
\end{aligned}$$

The probability that a correct $k_{\min}$-mer v appears no more than m times is at most

$$\sum_{i=0}^{m}\binom{t}{i}p^i(1-p)^{t-i} \quad \text{where} \quad p = \frac{l-k_{\min}+1}{g-l+1}\cdot(1-e)^{k_{\min}} \qquad \square$$

Theorem 4: Assume that a contig c in H_k is treated as dead-end and removed. The probability that c is a correct contig is less than

$$2\left[1-\frac{l-k-2}{g-l+1}\cdot(1-e)^{k+3}\right]^t$$

Proof: A contig c in H_k is treated as dead-end only if c is of length less than $3k - 1$ and is not a dead end in H_{k-1}. Since all contigs in H_{k-1} are preserved in H_k, c is removed because (1) the length of c is at least $3(k - 1) - 1$ and shorter than $3k - 1$, or (2) c is shorter than $3(k - 1) - 1$ and one of its ends has 0 in-degree or out-degree in H_k. Thus c will not be treated as a dead-end if the two adjacent $(k+3)$-mers of c is sampled. By considering $k_{\min} = k + 3$ and $m = 0$ in Theorem 1, the probability that no read contains a particular $(k+3)$-mer is at most

$$\left[1-\frac{l-k-2}{g-l+1}\cdot(1-e)^{k+3}\right]$$

and the probability that no read contains a particular $(k+3)$-mer cover a particular end of c is at most

$$2\left[1-\frac{l-k-2}{g-l+1}\cdot(1-e)^{k+3}\right]^{t} \qquad \square$$

Theorem 5 Shows that $H_{k_{\max},1}$ has at most the same number of gaps as $H_{k_{\max},2}$. There are some cases that there is a longer contig in $H_{k_{\max},1}$ which is not in $H_{k_{\max},2}$. For example, consider a contig c in H_k stops before vertex u whose in-degree = 1 and out-degree >1 and all branches of u are shorter than $2k$ and only u to v is correct. If there is only one read contains u and its 2 pairs of neighboring nucleotides at both ends which has error in other positions and there is no error read linking c with other branches, there is a longer contig c' in $H_{k+2,1}$ that contains c which does not appear in $H_{k+2,2}$.

Theorem 5: Let $H_{k,s}$ denote the accumulated de Bruijn graph graph H_k at step s. If a $k_{\max}$-mer ($k_{\max+1}$-mer) in the genome appears in $H_{k_{\max},2}$, it also appears in $H_{k_{\max},1}$.

Proof: By induction on $k_{\max}$. Consider $k_{\max} = k_{\min} + 2$. Given a $k_{\max}$-mer (($k_{\max}$+1)-mer) v does not appear in $H_{k_{\max},1}$, let v' be the shortest substring of v of length-k which does not appear as a vertex in $H_{k,1}$ or an edge in $H_{k-1,1}$, $k_{\min} \leq k \leq k_{\max}$.

If $k = k_{\min}$, i.e. v' does not appear in $H_{k_{\min},2}$, v does not appear in $H_{k_{\max}+1,2}$.

If $k = k_{\min}+1$, there are two cases: (a) v' does not appear in $H_{k_{\min},1}$ as an edge or (b) v' is a vertex on a dead-end with length less than $2(k_{\min}+1)$ in $H_{k_{\min}+1,1}$. In case (a), since any ($k_{\min}$+2)-mer contains v' as substring does not appear in $H_{k_{\max},2}$, v does not appear in $H_{k_{\max},2}$. In case (b), v is a vertex on a dead-end with length less than $2(k_{\min}+2)$ in $H_{k_{\max},2}$ which will be removed.

If $k = k_{\min}+2$, there are two cases: (a) v does not appear in $H_{k_{\min}+1,1}$ as an edge or (b) v is a vertex on a dead-end with length less than $2(k_{\min}+2)$ in $H_{k_{\min}+2,1}$. In case (a), consider the path (v_1, v_2, v_3) in $H_{k_{\min},1}$ representing the ($k_{\min}$+2)-mer v. Since v does not appear in $H_{k_{\min}+1,1}$ as an edge, v_2 has >1 in-degree or out-degree and there is no read contains v' as subsring. Thus the in-degree and out-degree of v are 0 in $H_{k_{\max},2}$ and v will be removed as dead-end. In case (b), v is a vertex on a dead-end with length less than $2(k_{\min}+2)$ in $H_{k_{\max},2}$ which will be removed.

If $k = k_{min}+3$, i.e. v does not appear in $H_{k_{max},1}$ as an edge, the path (v_1, v_2, v_3, v_3) in $H_{k_{min},1}$ representing the ($k_{min}+3$)-mer v is not a potential contig and there is no read contains v as a substring. Thus v does not appear in $H_{k_{max},2}$ as an edge. □