IPSJ Transactions on Bioinformatics Vol.8 2–8 (Jan. 2015)

[DOI: 10.2197/ipsjtbio.8.2]

Original Paper

A Memory Efficient Short Read *De Novo* Assembly Algorithm

Yuki Endo[1,a)] Fubito Toyama[1] Chikafumi Chiba[2] Hiroshi Mori[1] Kenji Shoji[1]

Received: October 17, 2014, Accepted: October 29, 2014, Released: January 27, 2015

Abstract: Sequencing the whole genome of various species has many applications, not only in understanding biological systems, but also in medicine, pharmacy, and agriculture. In recent years, the emergence of high-throughput next generation sequencing technologies has dramatically reduced the time and costs for whole genome sequencing. These new technologies provide ultrahigh throughput with a lower per-unit data cost. However, the data are generated from very short fragments of DNA. Thus, it is very important to develop algorithms for merging these fragments. One method of merging these fragments without using a reference dataset is called *de novo* assembly. Many algorithms for *de novo* assembly have been proposed in recent years. Velvet and SOAPdenovo2 are well-known assembly algorithms, which have good performance in terms of memory and time consumption. However, memory consumption increases dramatically when the size of input fragments is larger. Therefore, it is necessary to develop an alternative algorithm with low memory usage. In this paper, we propose an algorithm for *de novo* assembly with lower memory. In our experiments using *E.coli* K-12 strain MG 1655 and human chromosome 14, the memory consumption of our proposed algorithm was less than that of other popular assemblers.

Keywords: bioinformatics, next generation sequencing, *de novo* assembly, de Bruijn graph

1. Introduction

Determining whole genome sequences of various species has many applications not only in understanding biological systems, but also in medicine, pharmacy, and agriculture. In recent years, the emergence of high-throughput next-generation sequencing (NGS) technologies has dramatically reduced the time and cost for whole genome sequencing. These new technologies provide ultrahigh throughput with a lower per-unit data cost. However, these technologies generate sequence data from many very small fragments (sometimes fewer than 100 base pairs) of DNA. These fragments are typically called reads. The precision of NGS is not perfect, and reads might include sequencing errors. Thus, developing algorithms for merging these reads is very important. Merging reads without reference data is called *de novo* assembly.

The *de novo* assembly algorithms can be classified into two approaches based on their features: overlap-layout-consensus (OLC) and de Bruijn graph. OLC relies on an overlap graph. Edena [1], MIRA [2], Celera [3], SSAKE [4], and VCAKE [5] assemblers are based on the OLC approach. In the OLC strategy, overlaps are found by all-against-all pair-wise comparison. Overlap graphs are constructed from pair-wise overlaps. In the overlap graphs, a vertex represents a read and an edge denotes an overlap between two connected vertices (reads). Consensus sequences are determined by tracing paths in the graph. On the other hand, Velvet [6], ABySS [7], ALLPATHS [8], and SOAPdenovo [9] assemblers are based on de Bruijn graph approach. In the de Bruijn graph, a vertex represents a sequence of k bases (k-mer), where k is any positive integer. An edge represents an overlap of k-1 bases. Thus, two connected vertices are denoted by a k-1 overlap between their vertices (k-mers). After the de Bruijn graph is constructed from reads obtained by NGS, contigs are determined by tracing paths in the graph. The de Bruijn graph approach is most widely applied to the short reads from Solexa and SOLiD platforms. In this approach, fixed-length (k-1) overlaps are found and redundant k-mers (subsequences) are compressed, making it suitable for assembling vast quantities of short reads. However, memory consumption increases dramatically when the size of input reads is extremely large (more than several gigabytes) and it is hard to use them for large-scale assemblies.

To overcome this problem, several algorithms [10], [11], [12], [13] have been proposed in recent years. These algorithms are also based on de Bruijn graph approach. In these algorithms, the data structures for representing the de Bruijn graphs are designed with small size. To realize the compact de Bruijn graph, succinct data structures [10], [11], Bloom filter [12] and FM-index [13] are used. However, the overall costs, including the costs for constructing the compact graph, are not discussed in detail because these papers focused on how to represent the compact de Bruijn graph. In general, the processes of constructing de Bruijn graph (such as k-mer counting) consume much memory and time. Therefore, developing an algorithm in consideration of overall costs is very important.

In this paper, we propose an algorithm for large scale *de novo*

[1] Graduate School of Engineering, Utsunomiya University, Utsunomiya, Tochigi 321–8585, Japan
[2] Faculty of Life and Environmental Sciences, University of Tsukuba, Tsukuba, Ibaraki 305–8572, Japan
[a)] 00.endo@gmail.com

assembly with low memory usage. Although our algorithm is based on de Bruijn graph approach in the same way as Velvet, edge information is not kept in the main memory. Thus, the amount of memory usage can be greatly reduced by our method. The maximum memory usage in overall assembly processes is evaluated and compared in our experiments. Therefore, in the proposed method, the overall costs for *de novo* assembly are taken into account. In addition, the data structure for representing de Bruijn graph is simple, and external storage for an assembly is not required. All required data for an assembly are stored in the main memory. In our experiments using the *E.coli* K-12 strain MG 1655, the average maximum memory consumption of the proposed method was approximately 13–19% of that of the popular assemblers. Moreover, in the experiments using human chromosome 14, the average amount of memory of our method was approximately 54–63% of that of the popular assemblers.

2. Assembly Algorithms with Low Memory Consumption

In this paper, we propose an algorithm for large scale *de novo* assembly with low memory usage. **Figure 1** shows the outline of our algorithm. First, all *k*-mers obtained by segmenting sequence reads are recorded. At the same time, the number of occurrences of each *k*-mer is also counted. Second, the de Bruijn graph is constructed using *k*-mers. Then, the graph is partitioned into subgraphs such that the subgraph has a simple path or a simple cycle. The simple path does not have repeating vertices or edges in the graph. Then subgraphs are connected to make a larger simple path. The data about the number of occurrences of a *k*-mer are used to make an informed selection of path connections. Finally, contigs are generated by tracing vertices in each of the connected graphs.

2.1 Extraction of *k*-mers

From all reads, *k*-mers are extracted. They are kept in a database in the memory as "*k*-mer integers." As shown in **Table 1**, a *k*-mer integer is a one-to-one numeric representation of each *k*-mer. Specifically, bases "A," "C," "G," and "T" correspond to the integers 0, 1, 2, and 3, respectively. Thus, a *k*-mer sequence is expressed as a quaternary numeral. For example, a 5-mer base sequence "ACGTA" is converted to the quaternary number 01230. This is converted to the decimal number 108. Thus, 108 is *k*-mer integer corresponding to "ACGTA". If a *k*-mer sequence was represented by a string, the required memory usage would be *k* bytes. Using the *k*-mer integer representation, the amount of memory for *k*-mer sequences can be reduced to one fourth of that of the required to represent the *k*-mer as a string. Using *k*-mer integers not only improves memory usage, but also processing time. Forward and reverse complement *k*-mer sequences are recorded as the same sequence in our method, which means that either of the two complementary sequences can be registered in the database kept in the main memory.

In this work, *k*-mer integers and the number of occurrences of the *k*-mers corresponding to the *k*-mer integer are kept in the main memory. In order to lower memory usage, other data (such as edges in the de Bruijn graph) are not kept in the main memory.

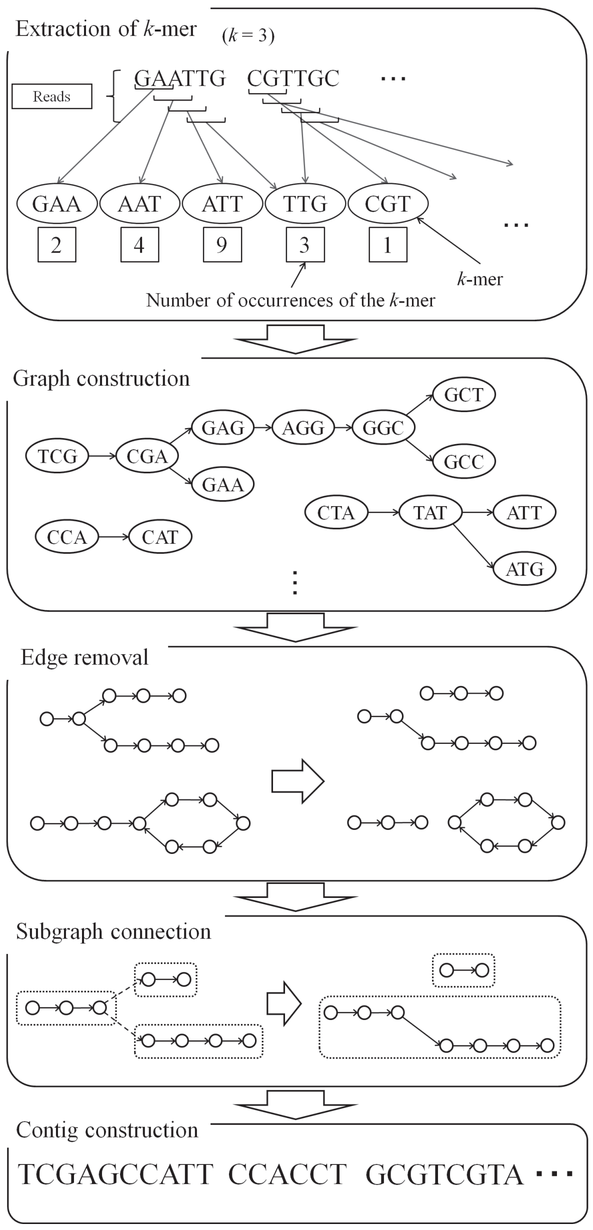


Fig. 1 Outline of the proposed method.

Therefore, since information about which *k*-mers are extracted from which reads is not stored, this information cannot be used in process of path tracing or graph construction. Although these features might affect the quality of the assembly, the amount of memory usage can be greatly reduced. The *k*-mer sequences in which the number of occurrences is small (less than a threshold) are not used in the graph construction because it is likely that such *k*-mer sequences contain sequencing errors. In our experiments, the threshold was set to 5. **Figure 2** shows the extraction of *k*-mers and the contents of the database in the proposed method.

2.2 Graph Construction

The de Bruijn graph is constructed using *k*-mers. In the de Bruijn graph, each vertex represents a *k*-mer. An edge represents an overlap of *k*-1 bases. Thus, two connected vertices denote a *k*-1 overlap between their vertices (*k*-mers). For example, there is an edge between the two vertices corresponding to "ACGTA" and "CGTAC." The direction of the edge is from "ACGTA" to

Table 1 *k*-mer sequence corresponding to *k*-mer integer (in case of 5-mer).

k-mer	Quaternary	Binary	*k*-mer integer (decimal)
AAAAA	0	0	0
AAAAC	1	1	1
AAAAG	2	10	2
AAAAT	3	11	3
AAACA	10	100	4
AAACC	11	101	5
AAACG	12	110	6
AAACT	13	111	7
AAAGA	21	1000	8
AAAGC	22	1001	9

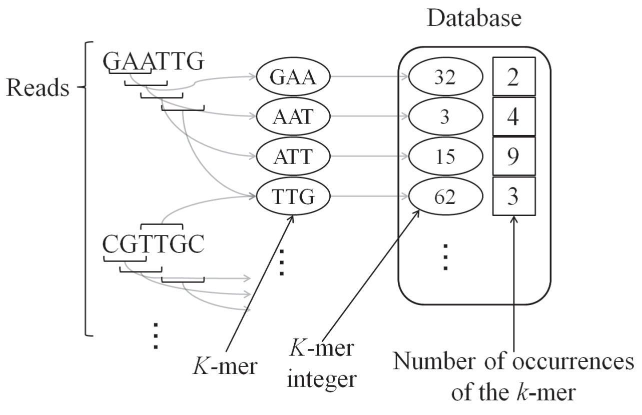


Fig. 2 Extraction of *k*-mer and the contents of database in the proposed method (in case of 3-mer).

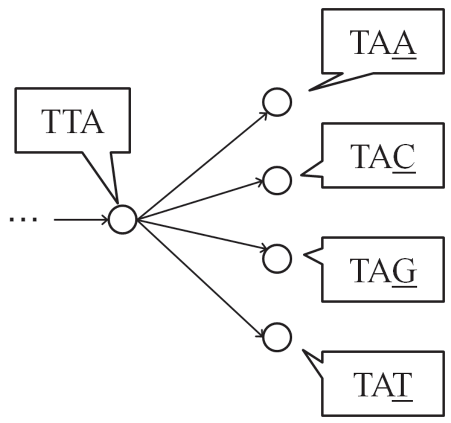


Fig. 3 Example of 4 types of *k*-mers which are connected to the current vertex (in case of 3-mer).

"CGTAC." The contigs are generated by tracing vertices in the de Bruijn. However, de Bruijn graph constructed from short reads has numerous branches and cycles. Therefore, it is difficult to find the paths from which contigs are constructed. The method for finding paths is described in Section 2.3. In the proposed method, if there is a *k*-1 overlap between *k*-mers (vertices), an edge is constructed between their vertices (*k*-mers). On the other hand, in popular *de novo* assemblers, an edge is constructed by a *k*-1 overlap between *k*-mers which are extracted from *k*+1-mer in a read. Therefore, the de Bruijn graph constructed by our method is a little different from others. However, the line graph constructed from *k*+1-mers by our method is the same as the graph generated from *k*-mers by popular de novo assemblers.

In conventional algorithms using de Bruijn graphs, when the graph is constructed, edge information about which vertices are connected to each other is also kept in main memory. Since there are many edges in the graph, keeping all the edge information consumes a huge amount of memory. In our method, the edge information is not kept in main memory. Thus, although the computational time for assembly is increased, memory usage can be reduced. The existence of the edge is calculated only when it is required. Specifically, the vertices that are connected by a directed edge from a vertex have only 4 types of *k*-mers because the *k*-mers, which are represented by the connected vertices, overlap by *k*-1 bases as shown in **Fig. 3**. Thus, the connected vertices (*k*-mer sequences) can be obtained by checking for four values of *k*-mer integers in the database. By using this method, although processing time to check the values of *k*-mer integer is increased, the memory for keeping edge connection data is not required. Only the data representing the vertices are kept in the database. Thus, the amount of memory can be greatly reduced in our method. Construction of the graph is finished by registering the *k*-mer integers from all *k*-mer sequences and the number of occurrences of each in our database.

2.3 Edge Removal

As mentioned in Section 2.2, the constructed graph has numerous branches and cycles. Consequently, it is important to select the connections in the path from which a contig is constructed. **Figure 4** shows examples of branches. A vertex with multiple edges connecting to other vertices is illustrated in Fig. 4 (a). A vertex with multiple edges connecting from other vertices is shown in Fig. 4 (b). Although the outdegree of the vertices with branches is two in Fig. 4, the maximum number of the out degree is four. Figure 4 (c) shows an example of a cycle. Actually, these branches and cycles are intricately intertwined. A path from the directed graph in which branches and cycles are included is needed to generate a contig. It is necessary to determine a unique simple path based on some criteria. The edge removal process we have used is as follows.

(1) A start vertex (*k*-mer) that has the largest number of occurrences is selected.
(2) The start vertex is set to the current vertex.
(3) Check for vertices that are connected to the current vertex.
  (a) If one connected vertex is found, the vertex is set to the current vertex. Go to 3.
  (b) If multiple connected vertices are found, one of them is set to the current vertex. (The details for this selection are described in later in this section.) Go to 3.
  (c) If the connected vertex is not found, the current vertex is regarded as the end vertex. Go to 4.
(4) Check for additional vertices that have not been selected yet.
  (a) If there are additional vertices that have not been selected, a new start vertex with the largest number of occurrences is selected from the vertices that have not been checked yet. Go to 2.
  (b) If all vertices have been checked, the process is finished.

In this process, the vertices that are put together in a path are assigned the same label. A path from the start vertex to the end vertex represents a subgraph. Multiple subgraphs are created in this process.

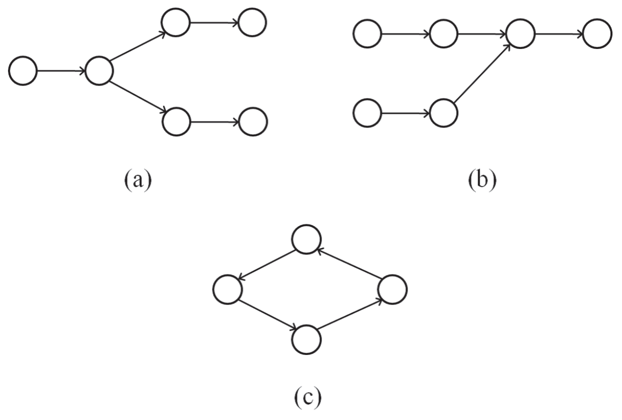


Fig. 4 Examples of branches and cycle.

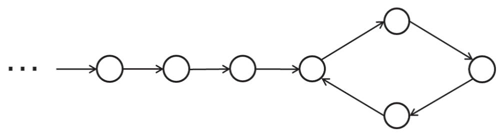


Fig. 5 Example of a cycle which is a part of a path.

The inclusion of branches and cycles in vertex selection is as follow: when there are multiple out going edges from the current vertex as shown Fig. 4 (a), the edge connected to the vertex in which the number of occurrences is the largest is selected, and the other outgoing edges are removed. When there are multiple incoming edges as shown in Fig. 4 (b), the edge with the largest number of occurrences is kept and other incoming edges are removed. In this edge selection process, when the difference in the number of *k*-mers between the selected vertex and other unselected vertices is less than a given threshold, the current vertex is regarded as an end vertex and a new start vertex is selected again. The current vertex is also regarded as the end vertex when the label of the selected vertex is the same as that of the current vertex as shown in Fig. 4 (c) (in the case of a cycle). In many cases, a cycle is a part of a path as shown in **Fig. 5**, and the vertices, which are in a cycle, are assigned the new path label. Thus, a cycle is identified separately from the current scanning path.

There is also the case in which the selected vertex has been assigned to other label as shown in **Fig. 6**. In this case, all vertices in the path that has the selected vertex are reassigned to the label of the current vertex.

2.4 Subgraph Connection and Contig Construction

To construct a longer path, subgraphs obtained by the process described in previous section are connected. The outline of the subgraph connection process is as follows. First, a subgraph with simple path is selected. The subgraph with the longest path is selected from those subgraphs that have not previously been selected. However, a subgraph with a simple cycle can be selected more than once. This subgraph is set as the start subgraph. Next, the subgraphs in which the start vertex or the end vertex are connected to the start or end vertex of the selected subgraph are searched. The details of selecting connections for the subgraphs with a simple cycle are described later in this section. The process of checking the *k*-mer integers from the subgraphs is the same as described in the previous section. If a connecting subgraph is

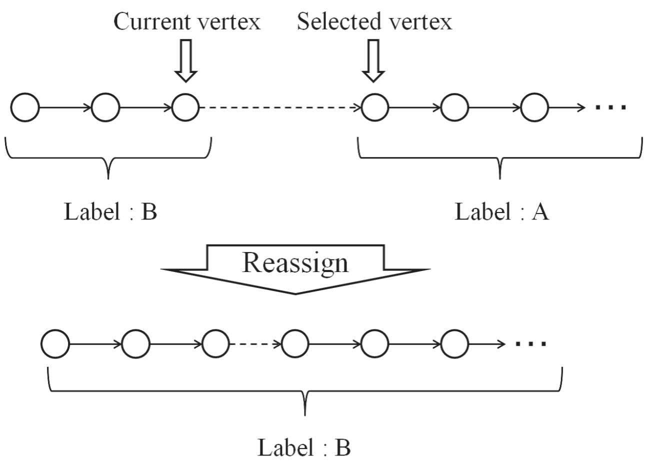


Fig. 6 Example of the label reassignment.

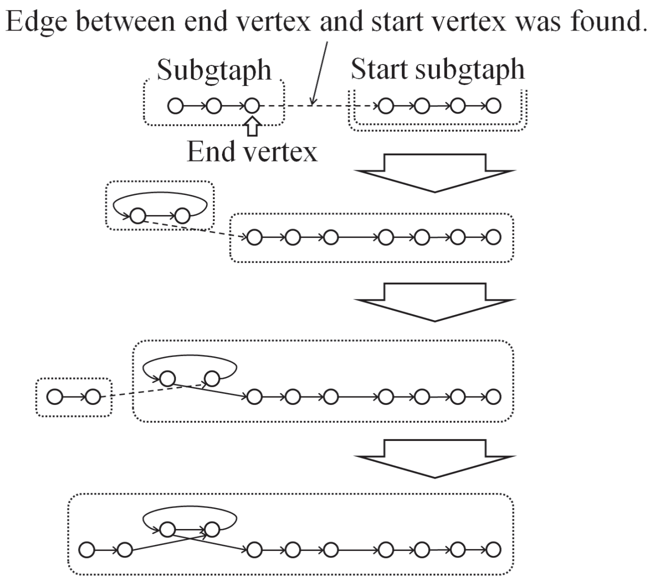


Fig. 7 Example of subgraphs connection.

found, the start (end) vertex is connected to the end (start) vertex, and the two subgraphs are merged into a single subgraph. This graph expanding process is repeated until no more merges can be made. If there are multiple subgraphs that can be connected, the subgraph with longer simple path is selected. **Figure 7** shows an example of connecting subgraphs. The connection in this example is on the left side. The same process is also performed on the right side.

In addition, when connecting to the subgraphs with a simple cycle as shown in **Fig. 8**, the vertex v_a is the start vertex of subgraph G_{path} with a simple path. The vertex v_b is contained in subgraph G_{cycle} with a simple cycle. If the *k*-mers of the vertex v_a and v_b overlap each other, the vertex v_c that connected to v_b is checked to see if v_c is also contained G_{cycle}. The vertices v_b and v_c are regarded as the start (or end) vertex. Then, the G_{path} and G_{cycle} are merged. If the vertex v_c is connected to another vertex that is included in other subgraph, these subgraphs are merged again as shown in Fig. 8.

After the subgraphs are connected, a list of the vertices is obtained by tracing all the paths that are included in the subgraphs. A contig is generated by merging the various *k*-mers that are ref-

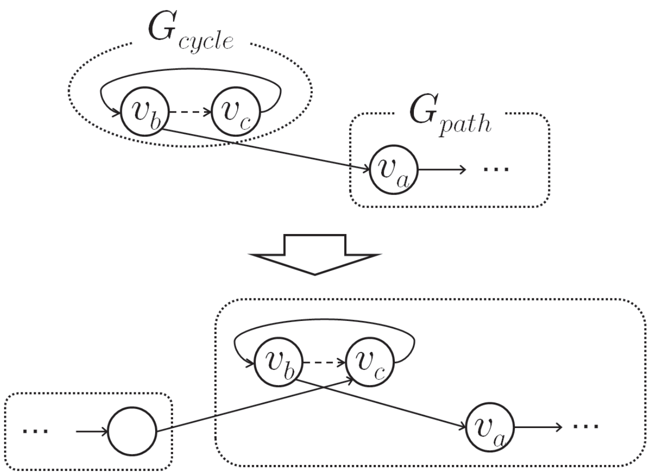


Fig. 8 Example of the connection of subgraph with a simple cycle.

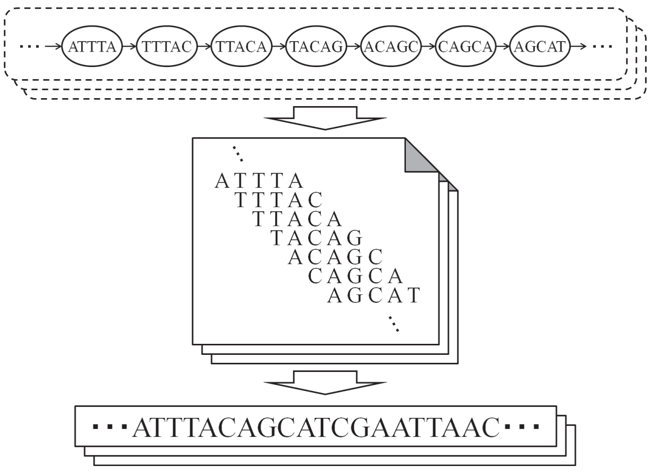


Fig. 9 Generation of contigs.

erenced from the vertices to eliminate the overlapping bases as shown in **Fig. 9**. The final contigs are obtained by repeating this process for all subgraphs. Any contigs that are longer than a given threshold are output. The threshold was set to length of the reads.

3. Experiments and Results

To evaluate the performance of the proposed method, we compare the performance of our method with that of Velvet (ver. 1.2.08) and SOAPdenovo2 (ver. 2.04). Velvet is one of the most popular *de novo* assembly algorithms based on the de Bruijn graph. In many papers on *de novo* genome assembly, Velvet is used as a comparison to assess the assembly performance. SOAPdenovo is also a popular *de novo* assembler based on the de Bruijn graph, which is designed to assemble large genomes. SOAPdenovo has been successfully used to assemble many published genomes. SOAPdenovo2 is the successor of SOAPdenovo. In SOAPdenovo2, assembly performance in memory consumption, accuracy, and coverage is improved. We assessed the maximum memory consumption, the running time, the contig length, and the accuracy of contigs from these programs in comparison to ours. The experimental assemblies using these three programs were all carried out on the same machine. (CPU: Intel Xeon E5-2660 2.2 GHz 8-core, Memory: 189 GByte)

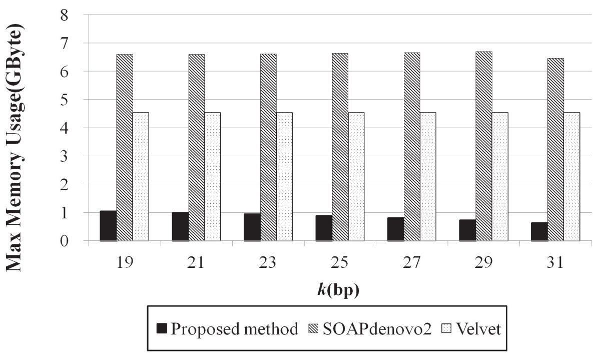


Fig. 10 Comparison of maximum memory consumption (*E.coli*).

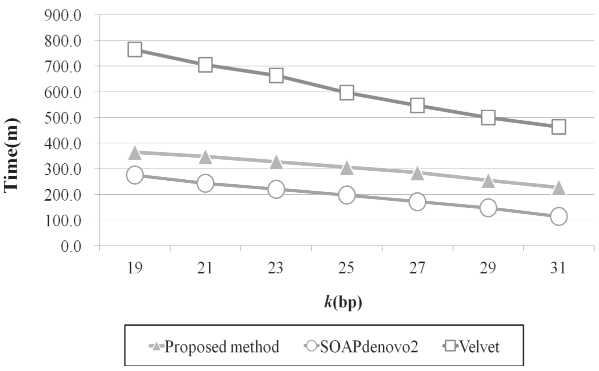


Fig. 11 Comparison of running time (*E.coli*).

3.1 E.coli *De Novo* Assembly

In the first experiment, *E.coli* K-12 strain MG 1655, for which the complete DNA sequence is known, was used. The sequence length is approximately 4.6Mbp. The sequence data does not include gaps ("N"). The experimental assembly was conducted using reads (35 bp) that were generated from the genome by NGS. The assemblers were run with the *k*-mer sizes of 19, 21, 23, 25, 27, 29, and 31. We used the input reads as single end and data in FASTA format.

Figures 10 and **11** show the maximum memory usage and the running time of each assembly algorithm for each tested *k*-mer in *E.coli*. The average maximum memory consumption of the proposed method was approximately 13% of SOAPdenovo2, and approximately 19% of Velvet. Therefore, we met our goal of reducing memory usage. As shown in Fig. 11, the average running time of the proposed method was slightly slower than SOAPdenovo2, and Velvet was faster than both of the other methods were. In the proposed method, the connection between vertices is checked each time a path is traced. Thus, the running time is increased because of additional processing time when searching for the connections between vertices. However, since the path tracing algorithms for resolving branches and cycles are very simple, there were not large differences in the total running time of each method. The running time for all programs was shortened as *k*-mer size increased in all assemblers. In our proposed method, the amount of memory was reduced with increasing size of the *k*-mer. A relationship between k-mer size and maximum memory usage was not seen in Velvet and SOAPdenovo2.

Table 2 shows the results of the assemblies for *E.coli*. The N50 length is defined as the length of the shortest contig where the sum of contigs of an equal length or longer is at least 50% of

Table 2 Comparison of assemblies (*E.coli*).

Assembler	Best *k*-mer size (bp)	N50 (kbp)	# of contigs	Total (kbp)	Genome covered (%)	Error rate (%)
Proposed method	25	16.4	602	4,514	96.76863	0.00272
SOAPdenovo2	29	19.0	2008	4,542	98.33118	0.06988
Velvet	29	22.9	754	4,544	98.02691	0.00000

Table 3 Comparison of assemblies (human chromosome 14).

Assembler	Best *k*-mer size (bp)	N50 (bp)	# of contigs	Total (kbp)	Genome covered (%)	Genome covered without gaps (%)	Error rate (%)
Proposed method	51	1,119	217,033	101,255	79.39556	96.53553	0.71973
SOAPdenovo2	63	3,682	186,558	91,358	81.15031	98.66909	0.03682
Velvet	63	5,108	73,054	83,706	78.97097	96.01927	0.00026

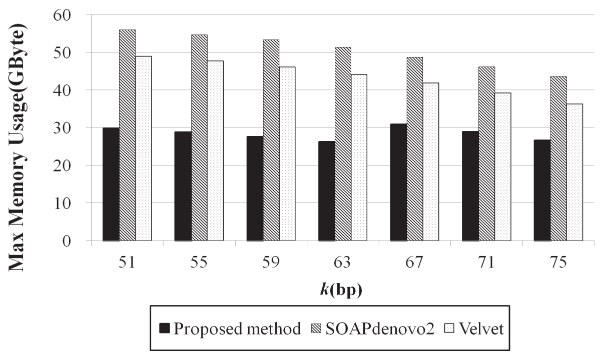


Fig. 12 Comparison of maximum memory consumption (Human Chr14).

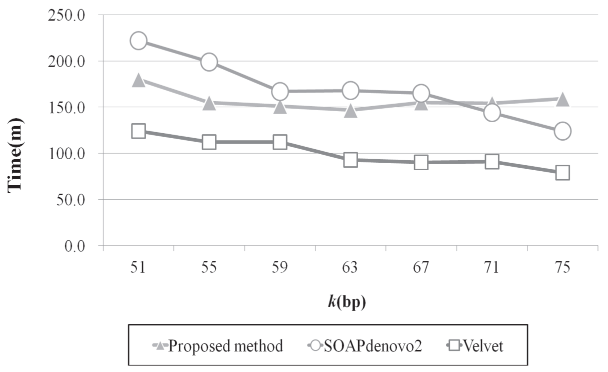


Fig. 13 Comparison of running time (Human Chr14).

the total length of all contigs. The best *k*-mer size was the size providing the largest N50. The results for the best *k*-mer size are shown in Table 2. The N50 length of the proposed method was slightly shorter than that of the others as shown in Table 2. The main reason for these results is that the path-tracing algorithm for resolving branches and cycles is very simple in the proposed method. On the other hand, there were not large differences in the genome coverage and the error rate. It is difficult to say which of the methods is better.

3.2 Human Chromosome 14 *De Novo* Assembly

In the second experiment, the complete DNA sequence for human chromosome 14 was used. The sequence length is approximately 107 Mbp, the ungapped sequence is approximately 88 Mbp. Assemblies were performed using the reads (101 bp) form GAGE [14] datasets. The GAGE (Genome Assembly Gold-standard Evaluations) is one of the performance comparison datasets used for *de novo* assembly algorithms. GAGE has focused on the quality of the assembly, but not on memory requirements. We used the dataset as a single end and FASTA format, converted from the reads in GAGE datasets, which are paired end and FASTQ format. The assemblers were run with *k*-mer sizes of 51, 55, 59, 63, 67, 71, and 75.

Figures 12 and **13** show the maximum memory usage and the running time of assembly of each *k*-mer for human chromosome 14. The average maximum memory consumption of the proposed method for human chromosome 14 was approximately 54% of SOAPdenovo2, and that was approximately 63% of Velvet. The amount of memory used by the proposed method was reduced for both experimental datasets, and the purpose of this method was achieved. However, as shown in Fig. 13, the average running time of the proposed method was slower than that of Velvet and slightly faster than that of SOAPdenovo2. The amount of memory usage was reduced for increased *k*-mer size in both SOAPdenovo2 and Velvet. In the proposed method, a relationship between *k*-mer size and maximum memory usage was not consistent for the human assembly.

Table 3 shows the results of the assemblies for human chromosome 14. The N50 length of the proposed method was shorter than that of the others as shown in Table 3. This is likely due to the simplicity of the path-tracing algorithm as mentioned in Section 3.1. Thus, the maximum length and N50 length could be improved with a more complicated path-tracing algorithm. On the other hand, there were not large differences in the genome coverage and the error rate.

4. Conclusion

In this paper, we propose an algorithm for large-scale *de novo* assembly with low memory usage. In our experiments using the *E.coli* K-12 strain MG 1655, the average amount of memory used in the proposed method was approximately 13–19% of SOAPdenovo2 and Velvet. Moreover, in the experiments using human chromosome 14, the average amount of memory used by the proposed method was approximately 54–63% of the memory used by the other assemblers. These results showed that the proposed method outperformed SOAPdenovo2 and Velvet for memory consumption. On the other hand, the N50 of contigs obtained by the proposed method was worse than that of the other assemblers. Further investigation is needed to improve the N50 of contigs in our method by modifying the path-tracing algorithm.

Acknowledgments This work was supported by a grant from the Japan Society for the Promotion of Science (24240062).

References

[1] Hernandez, D., Francois, P., Farinelli, L., Osteras, M. and Schrenzel, J.: De novo bacterial genome sequencing: Millions of very short reads assembled on a desktop computer, *Genome Res.*, Vol.18, No.5, pp.802–809 (2008).

[2] Chevreux, B., Pfisterer, T., Drescher, B., Driesel, A.J., Muller, W.E., Wetter, T. and Suhai, S.: Using the miraEST assembler for reliable and automated mRNA transcript assembly and SNP detection in sequenced ESTs, *Genome Res.*, Vol.14, No.6, pp.1147–1159 (2004).

[3] Miller, J.R., Delcher, A.L., Koren, S., Venter, E., Walenz, B.P., Brownley, A., Johnson, J., Li, K., Mobarry, C. and Sutton, G.: Aggressive assembly of pyrosequencing reads with mates, *Bioinformatics*, Vol.24, No.24, pp.2818–2824 (2008).

[4] Warren, R.L., Sutton, G.G., Jones, S.J. and Holt, R.A.: Assembling millions of short DNA sequences using SSAKE, *Bioinformatics*, Vol.23, No.4, pp.500–501 (2007).

[5] Jeck, W.R., Reinhardt, J.A., Baltrus, D.A., Hickenbotham, M.T., Magrini, V., Mardis, E.R., Dangl, J.L. and Jones, C.D.: Extending assembly of short DNA sequences to handle error, *Bioinformatics*, Vol.23, No.21, pp.2942–2944 (2007).

[6] Zerbino, D.R. and Birney, E.: Velvet: algorithms for de novo short read assembly using de Bruijn graphs, *Genome Res.*, Vol.18, No.5, pp.821–829 (2008).

[7] Simpson, J.T., Wong, K., Jackman, S.D., Schein, J.E., Jones, S.J. and Birol, I.: ABySS: A parallel assembler for short read sequence data, *Genome Res.*, Vol.19, No.6, pp.1117–1123 (2009).

[8] Butler, J., MacCallum, I., Kleber, M., Shlyakhter, I.A., Belmonte, M.K., Lander, E.S., Nusbaum, C. and Jaffe, D.B.: ALLPATHS: De novo assembly of whole-genome shotgun microreads, *Genome Res.*, Vol.18, No.5, pp.810–820 (2008).

[9] Li, R., Zhu, H., Ruan, J., Qian, W., Fang, X., Shi, Z., Li, Y., Li, S., Shan, G., Kristiansen, K., Li, S., Yang, H., Wang, J. and Wang, J.: De novo assembly of human genomes with massively parallel short read sequencing, *Genome Res.*, Vol.20, No.2, pp.265–272 (2010).

[10] Conway, T.C. and Bromage, A.J.: Succinct data structures for assembling large genomes, *Bioinformatics*, Vol.27, No.4, pp.479–486 (2011).

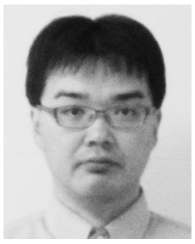
[11] Bowe, A., Onodera, T., Sadakane, K. and Shibuya, T.: Succinct de Bruijn graphs, *WABI*, Lecture Notes in Computer Science, Vol.7534, pp.225–235, Springer (2012).

[12] Chikhi, R. and Rizk, G.: Space-efficient and exact de Bruijn graph representation based on a Bloom filter, *WABI*, Lecture Notes in Computer Science, Vol.7534, pp.236–248, Springer (2012).

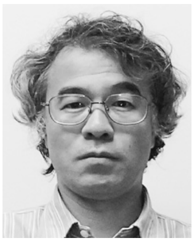
[13] Chikhi, R., Limasset, A., Jackman, S., Simpson, J. and Medvedev, P.: On the Representation of de Bruijn Graphs, *RECOMB*, Lecture Notes in Computer Science, Vol.8394, pp.35–55, Springer (2014).

[14] Salzberg, S.L., Phillippy, A.M., Zimin, A., Puiu, D., Magoc, T., Koren, S., Treangen, T.J., Schatz, M.C., Delcher, A.L., Roberts, M. et al.: GAGE: A critical evaluation of genome assemblies and assembly algorithms, *Genome Res.*, Vol.22, No.3, pp.557–567 (2012).

Yuki Endo was born in 1987. He received his B.S. and M.S. degrees from Utsunomiya University in 2010 and 2012, respectively. He is now a doctoral course student of Utsunomiya University. His research interest is bioinformatics.



Fubito Toyama is an associate professor at the Graduate School of Engineering, Utsunomiya University. He received his Ph.D. degrees in engineering from Utsunomiya University in 2000. His current research interests are evolutionary computation, combinatorial optimization and bioinformatics. He is a member of IPSJ, IEICE and ITE.



Chikafumi Chiba was born in 1965. He received his M.Ed. from Nara University of Education in 1992 and Ph.D. from University of Tsukuba in 1995. He became an associate professor at University of Tsukuba in 2006. His current research interest is the mechanism of body-parts regeneration in the adult newt and humans. He is a representative of the Japan Newt Research Community and a member of the Japan Society of Regenerative Medicine.

Hiroshi Mori was born in 1979. He received his Ph.D. from University of Tsukuba in 2007. He became a researcher at University of Tsukuba in 2007 and an assistant professor at Utsunomiya University in 2011. His current research interest is computer graphics. He is a member of the ACM.

Kenji Shoji received the M.S. and Ph.D. degrees from Keio University in 1978 and 1981, respectively. He has been a professor at the Graduate School of Engineering, Utsunomiya University, since 2009. His research interests include image processing and computer vision. He is a member of IEEE, IPSJ, IEICE, IEEJ, and ITE.

(Communicated by *Tetsuo Shibuya*)