An Algorithm for Classifying DNA Reads

Mohammed Sahli\textsuperscript{1}, Tetsuo Shibuya\textsuperscript{2}

\textsuperscript{1}Department of Computer Science, Graduate School of Information Science and Technology, University of Tokyo - 7-3-1 Hongo, Bunkyo-ku, Tokyo 113-0033, Japan
\textsuperscript{2}Human Genome Center, Institute of Medical Science, University of Tokyo - 4-6-1 Shirokanedai, Minato-ku, Tokyo 108-8639, Japan

Abstract. A DNA read comes from either strand of the genome, and it has been believed in the impossibility of determining which one is the actual strand for the last three decades. Therefore, for the first time, we developed an algorithm for classifying DNA reads according to which strand they were generated from. Consequently, misassemblies can be reduced by integrating this algorithm as part of the general genome assembling pipeline. Moreover, the genome assembly problem can be further simplified by building an assembler which this algorithm is involved in. We validated this algorithm by comparing some existing assemblers with an assembler that uses our algorithm The DNAClassifier tool, that uses only this algorithm, was developed so that it can be pipelined with other tools in the genome assembly process. All versions of our assembler as well as the DNAClassifier tool are both available for free to the public and can be downloaded from http://sourceforge.net/projects/dnascissor/files/.

Keywords: genome assembler, reads classification, short reads, shotgun sequences.

1. Introduction

The genome assembly problem is a challenging problem in Bioinformatics. It consists in bringing millions of genomic reads all together in order to reconstruct the original sequence from which they were randomly originated from. Sequencing errors, repeats and reads orientation are the main constraints of this problem. Due to the difficulty of these constraints, the current assemblers are not able to reconstruct the genome but a set of contiguous sequences (i.e. contigs or scaffolds). Two approaches have been studied extensively in the literature: the Overlap-Layout-Consensus approach and the de Bruijn graph-based approach. The first approach consists in computing the overlaps between all genomic reads and constructing the so-called “the overlap graph”. Some assemblers that use this approach include: PCAP\textsuperscript{[7]}, TIGR\textsuperscript{[10]}, CAP3\textsuperscript{[6]}, the string graph of Myers\textsuperscript{[11]} and MIRA\textsuperscript{[12]}. However, computing the overlaps is the most time-consuming task and the main drawback of this approach. In addition, dealing with repetitive reads is quite difficult in the overlap graph. Consequently, in order to overcome these weaknesses, a fundamentally different approach was adopted by Pevzner et al.\textsuperscript{[2]}. In this approach, sequences are cut into smaller reads of the same length \(k\) (generally called \(k\)-mers) and mapped as paths through the graph (i.e. de Bruijn graph). Some of the assemblers that use this approach are: Euler assembler\textsuperscript{[2]}, SSAKE\textsuperscript{[13]}, EULER-SR\textsuperscript{[3]}, Velvet\textsuperscript{[8]}, ALLPATHS\textsuperscript{[14]}, ABYSS\textsuperscript{[15]}, and SOAPdenovo\textsuperscript{[9]}. It has been shown that the de Bruijn graph-based assemblers are more effective in practice for short read assemblies. However, despite the advantages of the de Bruijn graph data structure, it has not been analyzed deeply in current assembly methods. We developed an algorithm that traverses this graph in order to classify DNA reads depending on where they came from. We give the details of this algorithm so that it can be used by researchers for future assemblers. We developed a de novo assembler (Arapan-M) that integrates this algorithm. The DNAClassifier is another tool that we developed and it uses only the classification algorithm so that it can be easily pipelined in the assembly process of other assemblers.
2. Methods

2.1. De Bruijn sequences

Let $S$ be a string of length $n$, the $k$-mer composition or spectrum of $S$ is the multiset of $n-k+1$ $k$-mers in $S$ and is written $\text{Spectrum}(S,k)$ [1]. Let $k=4$ and $S=\text{ACGTTGTG}$; then $\text{Spectrum}(S,4)=\{\text{ACGT, CGTT, GTTG, TTGT, TGTG}\}$. This idea is used in the de novo genome assemblers and it was originally introduced in [2]. We consider a set of DNA reads which were generated from the same genome. In order to reconstruct the original sequence, we cut the reads into even smaller $k$-mers of the same length $k$. We say two $k$-mers $p$ and $q$ overlap if $\text{overlap}(p,q)=k-1$, that is, the suffix of length $k-1$ of $p$ coincides with the prefix of length $k-1$ of $q$ [1]. Concatenating all the $k$-mers leads to a graph structure called the de Bruijn graph (Figure 1). This graph, which became the pillar of the de novo genome assemblers, can be used to solve the problem of repeats more effectively than the overlap graph.

2.2. Basic idea for classifying reads

This paper suggests an approach to the DNA reads classification problem based on the notion of the de Bruijn graph. In this graph, each node $N$ represents a $k$-mer and has an opposite node $\bar{N}$ in the graph. The node $\bar{N}$ is characterized by the reverse compliment $k$-mer of that of the node $N$. The constructed graph may have $p$ connected components. The main idea underlying this algorithm is to create two sets of nodes for each connected component. Each set will represent the $k$-mers which were generated from the same strand. This task is done by traversing the corresponding connected component starting at an arbitrary node to create the first set (i.e. the first strand) whereas the second set (i.e. the reverse strand) is being simultaneously created starting at the opposite node $\bar{N}$. The details of the algorithm are given below.

2.3. The algorithm

Once the graph is created, we identify its connected components by running the depth-first search algorithm. If at least one node and its opposite node exist in the same connected component, it will be declared as a false component. For each false connected component, we do the following operations:
A) **Light isolation:** An edge is identified as a false connection if it goes out of a node \( N \) to its opposite node \( \bar{N} \). This happens frequently in the case of an even length palindrome as illustrated in Figure 2. All false edges are eliminated in this stage. The pseudo-code of this procedure is given below.

**Algorithm:** light_Isolation(Graph G(V,E), Set M)

**Input:**
- \( V \): set of nodes;
- \( E \): set of edges;
- \( M \): set of the couple "node and its opposite"

Begin

for \( x=(a,b) \in M \) do

if \( (a,b) \in E \) then

\( E := E \setminus \{(a,b)\} \)

End.

B) **Hard isolation:** Let \( S \) and \( R \) be two sets. We say there is a *direct connection* between \( S \) and \( R \) if and only if there is an edge that goes out of a node in one set and points to another node in the other set. Initially, only a single start node \( N \) and its opposite node \( \bar{N} \) are marked "discovered". \( S \) and \( R \) are initialized by \( N \) and \( \bar{N} \) respectively (Figure 3). To completely explore a node, we must evaluate edges going out of it and edges that point to it. If an edge goes to undiscovered node, we mark it and its opposite "discovered" and add them to \( S \) and \( R \) respectively. If an edge goes to a *discovered* vertex, it will be ignored because further contemplation will tell us nothing new about the component. The process stops exploring from a discovered node only if it has a *direct connection* to the other set (Figure 4). When there is no further exploration, we will only get a set of edges that go out of a set and point to another. These edges are simply removed, and as a result the nodes will not share the same connected component that contains their opposite nodes. Consequently, each false connected component will be divided into two subcomponents. The pseudo-code of the algorithm is given below.

**Algorithm:** hard_Isolation(Component C(V,E) , Set M)

**Input:**
- \( V \): set of nodes;
- \( E \): set of edges;
- \( M \): set of the couple "node and its opposite"

**Output:** Component C(V,E);

Set \( T, R, Q, S \);

Integers \( i, a, b \);

1. Begin
2. //Step 1: Initialization
3. Let \( a \) and \( b \) be two nodes such that \( a,b \in V \) and \( (a,b) \in M \);
4. \( V:=V \setminus \{a,b\} \);
5. \( Q:=Q \cap \{x / (a,x) \in E\} \);
6. \( Q:=Q \cap \{z / (z,a) \in E\} \);
7. \( S:=S \cap \{z / (y,z) \in E\} \);
8. \( S:=S \cap \{z / (z,y) \in E\} \);
9. \( R:=R \cap \{x / (x,y) \in E\} \);
10. \( T:=T \cap \{x / (y,x) \in E\} \);
11. \( S:=\emptyset \);
12. \( T:=\emptyset \);
13. \( R:=\emptyset \);
14. \( Q:=\emptyset \);
15. \( Q:=\emptyset \);
16. \( Q:=\emptyset \);
17. \( Q:=\emptyset \);
18. \( Q:=\emptyset \);
19. \( Q:=\emptyset \);
20. end for.
21. end for.
22. end while.
23. end for.
24. Q:=S;
25. S:=\emptyset;
26. end while.
27. end for.
28. //Step 3: remove all edges between T and R
29. for all \( x \in T \) do
30. for all \( y \in R \) do
31. if \( (x,y) \in E \) then
32. \( E:=E \setminus \{(x,y)\} \);
33. end for.
34. end for.
35. end for.

<table>
<thead>
<tr>
<th>Reference genome (ID)</th>
<th>Accession no.</th>
<th>Size (bp)</th>
<th>Read len. (bp)</th>
<th>Coverage</th>
<th>Error rate (%)</th>
<th>Type of dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beta vulgaris genomic clone ZR-47B15 (D1)</td>
<td>ZR-47B15</td>
<td>117150</td>
<td>27</td>
<td>616</td>
<td>2.10</td>
<td>Real</td>
</tr>
<tr>
<td>E. coli str.K-12 substr. MG1655 (D2)</td>
<td>NC_000913</td>
<td>4639675</td>
<td>100</td>
<td>35</td>
<td>1.0</td>
<td>Simulated</td>
</tr>
<tr>
<td>Saccharomyces cerevisiae, Chr. 5 (D3)</td>
<td>NC_001137</td>
<td>576874</td>
<td>100</td>
<td>35</td>
<td>1.0</td>
<td>Simulated</td>
</tr>
<tr>
<td>Saccharomyces cerevisiae, Chr. 7 (D4)</td>
<td>NC_001139</td>
<td>1090946</td>
<td>70</td>
<td>35</td>
<td>3.0</td>
<td>Simulated</td>
</tr>
<tr>
<td>Haemophilus influenzae (D5)</td>
<td>NC_007146</td>
<td>1914490</td>
<td>100</td>
<td>35</td>
<td>1.0</td>
<td>Simulated</td>
</tr>
</tbody>
</table>
Table 2. The comparison

<table>
<thead>
<tr>
<th>Genome</th>
<th>Assembler</th>
<th>No. of contigs</th>
<th>N50</th>
<th>Average length</th>
<th>Max length</th>
<th>Total length</th>
<th>Genome coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>D1</td>
<td>Arapan-M</td>
<td>69</td>
<td>2147</td>
<td>1581</td>
<td>8449</td>
<td>109124</td>
<td>93.15%</td>
</tr>
<tr>
<td></td>
<td>Velvet</td>
<td>75</td>
<td>1912</td>
<td>1446</td>
<td>9108</td>
<td>108495</td>
<td>92.61%</td>
</tr>
<tr>
<td></td>
<td>ABYSS</td>
<td>68</td>
<td>1900</td>
<td>1576</td>
<td>7088</td>
<td>107211</td>
<td>91.52%</td>
</tr>
<tr>
<td>D2</td>
<td>Arapan-M</td>
<td>898</td>
<td>8249</td>
<td>4969</td>
<td>35440</td>
<td>4462975</td>
<td>96.19%</td>
</tr>
<tr>
<td></td>
<td>Velvet</td>
<td>1083</td>
<td>6491</td>
<td>4139</td>
<td>41201</td>
<td>4483587</td>
<td>96.64%</td>
</tr>
<tr>
<td></td>
<td>ABYSS</td>
<td>931</td>
<td>8113</td>
<td>4795</td>
<td>35440</td>
<td>4464731</td>
<td>96.23%</td>
</tr>
<tr>
<td>D3</td>
<td>Arapan-M</td>
<td>80</td>
<td>15293</td>
<td>6794</td>
<td>41953</td>
<td>543540</td>
<td>94.22%</td>
</tr>
<tr>
<td></td>
<td>Velvet</td>
<td>109</td>
<td>8777</td>
<td>5060</td>
<td>25051</td>
<td>551618</td>
<td>95.62%</td>
</tr>
<tr>
<td></td>
<td>ABYSS</td>
<td>92</td>
<td>15293</td>
<td>5991</td>
<td>41953</td>
<td>551233</td>
<td>95.56%</td>
</tr>
<tr>
<td>D4</td>
<td>Arapan-M</td>
<td>168</td>
<td>11129</td>
<td>6184</td>
<td>32711</td>
<td>1039034</td>
<td>95.24%</td>
</tr>
<tr>
<td></td>
<td>Velvet</td>
<td>235</td>
<td>8210</td>
<td>4463</td>
<td>36131</td>
<td>1048944</td>
<td>96.15%</td>
</tr>
<tr>
<td></td>
<td>ABYSS</td>
<td>186</td>
<td>10201</td>
<td>5657</td>
<td>32711</td>
<td>1052327</td>
<td>96.46%</td>
</tr>
<tr>
<td>D5</td>
<td>Arapan-M</td>
<td>661</td>
<td>3787</td>
<td>2777</td>
<td>20265</td>
<td>1835788</td>
<td>95.89%</td>
</tr>
<tr>
<td></td>
<td>Velvet</td>
<td>634</td>
<td>3768</td>
<td>2900</td>
<td>21502</td>
<td>1838987</td>
<td>96.06%</td>
</tr>
<tr>
<td></td>
<td>ABYSS</td>
<td>688</td>
<td>3564</td>
<td>2664</td>
<td>20265</td>
<td>1833402</td>
<td>95.76%</td>
</tr>
</tbody>
</table>

3. Result

We developed the DNAClassifier tool to classify DNA reads depending on their strands and outputs a set of k-mers (that came from the same strand) in FASTA format. It was developed so that it can be pipelined in the assembly process. Moreover, we also integrated the DNA reads classification algorithm as a main phase in our de novo assembler (Arapan-M). All datasets which were used for making the comparison are shown in Table 1. D1 was downloaded from sharcs.molgen.mpg.de/download.shtml and it was initially used in [4][5]. All the experiments were run on an Intel(R) Core(TM) i7 CPU computer with eight cores operating at 2.93 GHz, 11.8 GB of memory, and Kernel Linux 2.6.32-64-generic operating system (Ubuntu). DNAClassifier and Arapan-M were implemented in C/C++ and compiled and run with the aid of a cross-platform application framework called Qt version 4.6.2 (64 bit). We compared the result of our assembler with ABySS and Velvet 1.1.3. The results are shown in Table 2. Arapan-M always achieved fewer contigs and higher N50 length compared to other assemblers. Although Velvet achieved the best maximum length of contigs, it produces great number of contigs compared to other assemblers. ABySS was the most competitive to Arapan-M in most cases.

4. Discussion

It was believed in the impossibility of solving the DNA reads classification problem. However, for the first time, we introduced an algorithm for tackling this problem. We also developed two tools that used this algorithm to illustrate its utilization and compared the results with other de novo assemblers. Our assembler can output the de Bruijn graph in different formats. Figure A and Figure B (see the appendix) shows the graph diagram of the white tailed deer coronavirus genome. These graphs were generated by another version of our assembler (Arapan-S). The two strands of the genome can be seen clearly in the graphs. However, this is only the case of a tiny genome, but for medium and big size genomes, the graphs will consist of a set of connected components. Each of these has its opposite in the graph. This is because of sequencing errors that limit the robustness of the DNA reads classification problem.

5. Acknowledgements

This work was partially supported by the Grant-in-Aid from the Ministry of Education, Culture, Sports, Science and Technology of Japan. We are grateful to Pr. Satoru Miyano (the head of Laboratory of DNA
Sequence Analysis and Laboratory of Sequence Analysis, Human Genome Center, University of Tokyo) for his additional support in publishing this work. We thank Mr. Yassine Bouhmadi for his correction.

6. References


Appendix

Figure A: The resulted graph of white tailed deer coronavirus genome by GraphViz visualization tool (sfdp layout)

Figure B: The resulted graph of white tailed deer coronavirus genome by Tulip visualization tool (Force Direct layout).