Assembling DNA genome from de novo transcriptome

Santiago Passos Patiño
Universidad Eafit
Colombia
spassos@eafit.edu.co

Juan David Arcila Moreno
Universidad Eafit
Colombia
jarcil13@eafit.edu.co

Mauricio Toro
Universidad Eafit
Colombia
mtorobe@eafit.edu.co

PROBLEM

We are facing a problem about DNA assembling, with some DNA segments the genome all the sequence represent must be reconstructed to be related with a specie. It must be developed an on-line algorithm that will constantly reading DNA sequences and it should determine if it belong or no to a bacterium, this process must be done efficiently in terms of memory and time with the less possible mistakes generated in the final complete sequence.

1.RELATED WORK

A de novo Genome Assembler based on MapReduce and Bi-directed de Bruijn Graph [1]

This paper talk about the development of an algorithm that uses MapReduce and Bi-directed Bruijn Graph for efficient DNA assembly, MapReduce is used in the process to parallelize and optimize the construction, compaction and cleaning of the graph.

First the algorithm read the sequences from the de novo, and while reading, it creates an undirected de Bruijn Graph, this kind of graph in its classical way is a directed graph, because of this, creating a Bi-directed form require some specials changes during the construction, after that it must be cleaned, because during the reading it may happen some errors, so the algorithm eliminated duplicated, and unnecessary nodes, and repeated or useless paths between the nodes. The pros of this implementation are that the traversal of the graph once it's already cleaned (fixed all the bubbles and gaps) it's very easy to reconstruct the genome, giving more consistent results with less errors.

HipMer: An Extreme-Scale De Novo Genome Assembler [2]

In this article, the authors explore efficient ways of: generating the k-mers of the gnome, traverse the de Bruijn graph and solve the errors occurred during the DNA reading.

For the first step while reading the sequence they categorize the k-mers according of its occurrence in the sequence, that way the ones with less recurrence can be determined as an error, as this algorithm is applied the memory used is reduced by 85%. Another improvement is in the traversal of the de Bruijn graph where the algorithm divide the graph

into different contigs, so each of the contigs can be traversed by a different processor dividing the amount of time required for reconstructing the genome. The last step of the algorithm is to identify and solve the problems caused by the bad reading, things like overlapped contigs, gaps between contigs, and ambiguous contigs of the graph called bubbles, this is solved by parallelizing the process and generating different states of the genomes with different possibilities of reconstructing it.

Using Matching DNA sequences Algorithms: Aho-Corasick (AC) and Boyer Moore (BM) [3]

Is usual to see these two algorithms to make high-efficiency comparison between two or more completes DNA. In our actual application, we search two equal n-sequence (Prefix and Suffix) present in two segments of DNA, because those algorithms were made for exact string matching and multi pattern finding, and we are looking for a not exactly length string, however, it can be very useful to search the longest n-substring common in all sequences, something that we need to construct K-Overlap graph or Bruijn Graph. For this reason, is important to consider in our solutions this kind of Algorithms.

The Aho-Corasick algorithm use two main stages: A Finite state machine construction stage and a matching stage. The finite state machine construct a suffix-tree, and the matching stage makes find out the pattern set within the given string. This make a "graph" that can contain cycles, and make possible find all the paths given the suffix of the sequences and patterns.

The Boyer Moore algorithm is an efficient string searching algorithm that is consider
the most efficient string-matching algorithm in usual applications, for example, in text editors and commands substitutions. The reason is that it works the fastest when the alphabet is moderately sized (in this case 4) and the pattern is relatively long.

K-Overlap Graph with TSP [4]

A K-overlap graph is a di-graph in which each string in a collection is represented by a node and node *s* is connected

to *t* with a directed edge if and only if some suffix of *s* equals a prefix of *t.* We say that *k* is the length of the suffix and prefix present in both strings, the weight of every edge is the k for those nodes connected by this edge. For our application, we assume that the length of the string is more than *k* and less than 100, also the nodes are the n-segments of DNA. To see the final and complete DNA sequence we find the shortest path which visits every node exactly once, this is the TSP (Traveling Salesman Problem).

The problem with this solution is that we are using NP-complete problems to find the complete DNA sequence, more exactly the TSP and get the common k-substrings between all the n-segments, so its complexity is not really good, but always give a correct solution. Another important characteristic of this solution is that it can't consider the possible ambiguity between the overlap of two or more segments, in that case we need to make a check to verify if every three nucleobases of the ARNm exist, in that case, the DNA sequence is correct.

2. Algorithm Selection

The algorithm that was chosen to solve the problem was the **bruijn graph** with this algorithm the program can reconstruct with the read the whole DNA sequence, using K-mers for construct every node in the graph. Every time that a string is read the program divide the sequence into K-mers to construct the nodes and the edges for the graph, once the graph is completely constructed the graph is traversed in order, from the first node (the one is not receiving an edge) to its adjacent node, this process continue until the last node is reached, then the original DNA sequence is finally finished.

3. Complexity

The complexity of the algorithm is divided in two parts, the first one is the construction of the graph, in this case the algorithm read **N** strings to reconstruct the dna sequence, each of this string are divided into k-mers. This require **n** the length of the read, then encode each K-mer in a hash table, that action is executed in O(1) where **m** is the size of the map, the total complexity is O(Nn) , **m** could be defined as the maximum number of k-mer , then m = S - k (k is the number chosen for the k-mer and S is the final sequence's size) that give us O(Nn).

The second part of the complexity is the reconstruction of the sequence, that is given by the traverse of the graph, in this case, if one pick enough large K-mer the graph is like a straight line, then the time to go through all the graph is |E| the numbers of edges, this is the same as S-1, so it's a linear time O(S).

The execution time was calculated by taking 5 different DNA sequences, making 3 random generations named G1, G2 and G3, to each DNA sequence. Those generations have a size between 300 and 3000 characters, and 100.000 sub-strings of length.

DNA sequence		***Time (S)***		
DNA	*Size(n)*	*G1*	*G2*	*G3*
AY325307	15600	14.482	13.8715	14.3172
AB042837	16692	15.3816	15.0784	14.849
AP002943	16508	15.0477	14.7665	14.7802
X54252	13794	13.4441	13.9547	13.6415
Y11832	17085	14.7137	14.9623	14.8736

Table 1: Comparision in the time used

The complexity in memory in the worst case is just the necessary to keep the map with the k-mers and the final DNA sequence, that give us O(S), the next table shows the memory required in different cases.

DNA Sequence	Size	Memory(MB)
AY325307	15600	5.1
AB042837	16692	5.3
AP002943	16508	5.5
X54252	13794	4.6
Y11832	17085	5.6

Table 2: Comparision in the memory used

REFERENCES

[1] Yuehua Zhang, 2016. A de novo genome assembler based on MapReduce and bi-directed de Bruijn graph. *2016 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*. (2016).

[2] Georganas, E. 2015. HipMer. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on - SC '15*. (2015).

[3]Pandiselvam.P. 2012. *A comparative study on string matching algorithms of biological sequences.* (2012)

[4] Jones, N. and Pevzner, P. 2004. *An introduction to bioinformatics algorithms*. MIT Press