Vol. 00 no. 00 2014
Pages 1–6

Optimal Selection of Enzyme Triad for Index Alignment of Contigs to Optical Maps Using MapReduce

Darshan Washimkar [1*]
[1]Department of Computer Science, Colorado State University, Fort Collins, CO, USA.

ABSTRACT
Motivation: Optical mapping creates a restriction enzyme map that has been used for scaffolding contigs and assembly validation for large genomes. There is a lot of research going on to increase the efficiency of contig alignment to the optical map. One of the problems encountered during the process of contig alignment to an optical map is a missing or wrong restriction sites. Frequently it happens, that a specific combination of restriction enzymes could not digest a recognition sequence, and hence the process of contig alignment has to deal with a missing or wrong restriction sites. There is a subtle balance between a good digestion process and the enzymes used for digestion. Different combinations of restriction enzymes can be tested in the laboratory to get best digestion, but that is very expensive and laborious.
Project Goal: The goal of this project is to bypass the expensive process of laboratory testing to figure out the best restriction enzyme triad for alignment of contigs to optical map. Another important part of project is to write a script that will find all combinations of enzyme triads. Enzymes selected to create the combinations must have cutting frequency specified by the user. Our software take a list of the enzymes combinations and contigs as the input parameters and output the best restriction enzyme triad for particular genome.

1 INTRODUCTION

Field of biophysics has seen many advances from the time when Frederick Sanger determined the sequence of insulin. One of the major milestone in the field of biophysics was a use of computer to study and compare multiple sequences. In 1970, Paulien Hogeweg and Ben Hesper coined the term 'Bioinformatics' to refer to the study of information processes in biotic systems. In 45 years from the advent of Bioinformatics field, we have seen numerous techniques that can assemble genome very efficiently in linear time. Even though next generation assembly techniques are really efficient in time complexity, there is always a trade off between running time and quality of the assembly for genome assembler techniques. If genome contains many repetitive regions, then most of the assembly techniques are unable to assemble good quality genome.

One of the reasons for erroneous assembly is a use of only short read information for assembling a genome. In case of the genome with repetitive region, it becomes very hard to find unique position for a sequence of nucleotides. One of the ways to overcome the misassembly problem in a genome with repetitive regions is a use of longer read information such as optical maps.

*Correspondence: darshanw@cs.colostate.edu

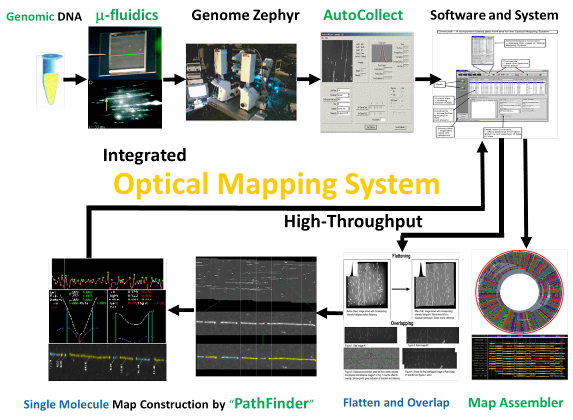


Fig. 1. Optical Mapping System [8]

1.1 Restriction enzyme

A restriction enzyme is an enzyme that cuts DNA at or near specific recognition nucleotide sequences known as restriction sites. There are over 417 restriction enzymes that have been studied in detail. In most optical mapping experiments, a combination of three restriction enzymes out of applicable 417 enzymes is used to digest the DNA molecule.

1.2 Optical Mapping

Process of optical mapping gives genome-wide, high resolution and ordered restriction map for single DNA molecule. The restriction map obtained from optical mapping is called optical map. Optical maps have a locations of restriction sites along the DNA. Information about the restriction sites collectively gives unique fingerprint for that particular sequence. A new technology used in creating an optical map can detect approximately as long as 30kb or as small as 800bp of fragment sizes. This parameter is very important to select value of bucket size from our algorithm. Optical maps can be use to scaffold and validate genomes.

Figure 1 shows a process of creating an optical map. In the first step of creating an optical map DNA is extracted from a cell and placed onto a special surface which contain microfluidic channels. This surface is called as optical mapping surface. In the next step, restriction enzymes are added onto optical mapping surface

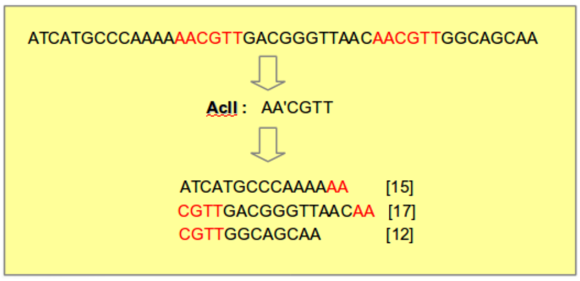


Fig. 2. *In silico* digestion

which create number of fragment of digested DNA molecule. Each fragment size is recorded by taking high resolution images and applying some machine learning algorithms on that images. With the help of map assembler, fragment sizes are used to generate genome-wide optical map.

1.3 *In silico digestion*

In silico digestion is a computational process to convert to convert contigs from sequence domain to the optical mapping domain. In *in silico* digestion process each restriction enzyme would cleave the short segment of DNA defined by the contig. Thus, *in silico* digested contigs are miniature optical maps.

Figure 2 shows the process of *in silico* digestion. For an enzyme AcII, which has a restriction sequence as AA'CGTT, linear search is carried out on contig sequence to find locations of restriction site. Restriction site locations are useful in constructing a fragment sizes. Figure shows that, after digesting a contig with AcII restriction enzyme, we get three fragments, each of size 15, 17 an 12. Information of fragment size is important for the method used this project.

1.4 Suffix Tree

Given a string S of length m, a suffix tree T encodes all suffixes of S, i.e. $S[i, m]$ for $1 \leq i \leq m$ The construction of such a tree for a string S takes time and space linear in the length of S. [16] Once constructed, several operations can be performed quickly, for instance, locating a substring in S, locating a substring if a certain number of mistakes are allowed etc.

For this project, we have used generalize suffix tree. Generalize suffix tree is also a suffix tree, but created with a set of strings. If the set of strings is represented by $T = S_1, S_2, S_3, ..., S_n$ having total length L then generalize suffix tree is a Patricia tree containing all L suffixes of the strings [13]. Generalize suffix tree can be constructed using Ukkonen's algorithm with $O(n)$ time complexity [14]. Generalize suffix tree help in find all z occurrences of a string P of length m in time complexity of $O(m + z)$ [13].

Figure 3 shows the generalize suffix tree built using two strings S_1= abab and S_2= aab. Suffixes that belong to S_1 are represented by '\$' symbol while suffixes of S_2 are represented by a special character '#'. If you have n different strings, then you will need n different special characters to represent suffixes of each string distinctly. Special characters should not be a part any of the strings represented in the suffix tree.

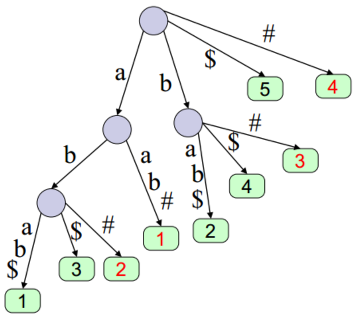


Fig. 3. Generalize suffix tree example with S_1= abab and S_2= aab [15]

2 RELATED WORK

De novo assembly can produce large contigs from a short sequence reads, but this process still produces a substantial number of errors [1, 2]. There are methods that use optical mapping information to improve the quality of assembly like SOMA [4], AGORA [3], and TWIN [5]. SOMA is an abbreviation for Scaffolding using Optical Map Alignment. SOMA is a scaffolding technique which employ a dynamic programming approach with time complexity of $O(n^2m^2)$. SOMA align *in silico* digested contigs to optical maps. On the similar line, TWIN aligns *in silico* digested contigs to optical map using FM-index and suffix array. TWIN is an index-based method that can align even larger genomes within seasonable running time. On the other hand, AGORA uses optical mapping information to build de Bruijn graph. This graph then used to assemble the genome.

There are some other methods like Gentig9, Valouev et al. 10 and BACop 11 that uses optical mapping information. Hence it becomes very important to get optical maps that can align uniquely to contig sequence. The optical mapping process heavily relies on a good digestion of DNA molecule by restriction enzymes. Since a large number of restriction enzymes are available [6], the choice of the most effective enzyme to digest a DNA molecule become difficult. Mathematical models like the one incorporated in REDI [7], can help to predict the chances of a particular enzyme to be useful for given DNA sequence. Experiments show that the use of multiple restriction enzymes give more accurate optical map, but the programs like REDI, which are based on individual enzyme probability could not help in such scenario.

3 METHOD

There are two main challenges in finding the best triad combination of enzyme that will generate a unique fingerprint for each contigs. The first challenge is to find all common fragment lengths from the fragments sizes obtained after *in silico* digestion in linear time. We used generalized suffix tree to address this problem. Second hurdle is an overall time taken to find common fragment lengths for all possible combinations of enzyme triad. We tried to solve this problem by implementing our algorithm in MapReduce framework and distribute tasks over multiple CPUs.

Our method of finding common fragment lengths need two input parameters. First is a name of three enzymes (triad) for which we

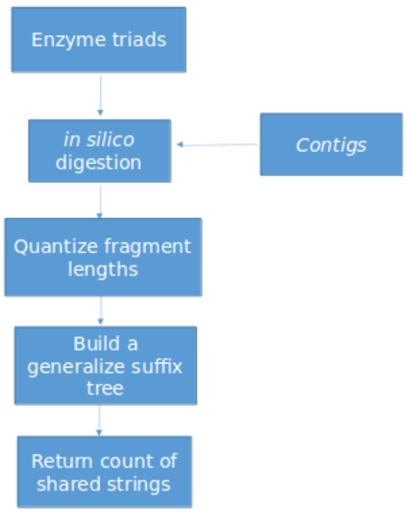

Enzyme triads
in silico digestion
Contigs
Quantize fragment lengths
Build a generalize suffix tree
Return count of shared strings

Fig. 4. Flow diagram of an algorithm to find number of common fragment lengths for given contigs and restriction enzyme triad

need to find a count of common fragment lengths. The second is the assembled contigs. We divided our method into three steps as follows. Figure 4 shows a flow diagram of an algorithm that counts numbers of common fragment lengths.

3.1 Converting Sequence Domain Information into Optical Domain Information

Contigs obtained from the assembler are the strings of nucleotides. Size of contigs may vary depending on the assembler use, size and quality reads. The contig information is in sequence domain format. To convert it to optical domain, we can use *in silico* digestion process explained in the above section. Let's consider, we have enzyme triad as E_1, E_2, E_3 and contigs sequence as C_1, C_2,C_3,...,C_n. *In silico* digestion process takes linear time to search restriction sites for given enzymes E_1, E_2, E_3 and on each contig C_1, C_2,C_3,...,C_n. Restriction sites can give us a size of each fragment created due to restriction sites. We don't consider first and last fragments for further calculation because they are most probably the artifacts of assembly process. Also, on the similar line if some contig produce fragments less than or equal to two then we don't consider those fragments.

3.2 Quantization Of Fragment Lengths

Optical maps are not precise enough. They can have an error margin of 100 to 200 base pairs. So to accommodate this error margin, we need to quantize the values of fragment sizes. In this project, we are using simple quantization technique with bucket size k. Value of k can vary depending on the expected error in optical mapping experiment. However, we are using a bucket size of 300 that means we have an error window of 150 base pairs. Each quantize value of fragment size is then converted to a character representing that bucket. All the quantized fragment sizes of contig C_n gives one string S_1. For each contig present in the input file, string of quantized fragment sizes is created. Let's call this strings as S_1, S_2, S_3,...,S_n.

3.3 Build And Traverse Generalize Suffix Tree

To find all common sub-strings of strings S_2, S_3,...,S_n in linear time, we use generalize suffix tree. Ukkonen's algorithm proposed in 1995, can generate generalize suffix tree in $O(n)$ time. Once the generalize suffix tree is built, it is a trivial task to find common substrings of S_1, S_2, S_3,...,S_n. Consider if, S_i = abmnob and S_j = abcde. We can say that, 'ab' is a common sub-string and character 'b' is repeated thrice. Hence we will return number of shared sub-string value as '5'.

Once the sub-string values of all possible triad combinations is calculated, the combination that gives the lowest value of sub-string can be called as the best enzyme combination that will produce a unique fingerprint for given contigs sequence. Thereby gives best digestion and good optical maps which can be uniquely aligned to contigs using TWIN or SOMA to eliminate assembly errors. Below is a pseudo code for the algorithm described in this section.

```
For each $\binom{n}{3}$ combinations of enzymes
{
  For each contig $C$ present in contig file
      (.fasta)
  {
    *in silico digestion*
    Find fragment lengths
    Quantize fragment lengths
    Convert quantized lengths to
        corresponding characters and create
        string $S_i$
  }
  Build suffix tree($S_1$, $S_2$, $S_3$,...,$S_n$)
  Count shared sub-string in suffix tree
  E = save(Enzyme combination, Count)
}

min_count = Find minimum count in E.
return(enzyme combination corresponding to
    min_count)
```

4 IMPLEMENTATION

The algorithm described in the previous section is implemented in MapReduce framework. We used Hadoop implementation of MapReduce. Map reduce helps to process the quantity of data in distributed environment. In Hadoop, instead of data being transferred to the computations, computations are transferred to the data. This approach is followed because bringing computations close to data is much easier on the network than transferring data over the network. In MapReduce, there are mappers, combiner and reducers which are managed by the master node.

Master Node: Master node manages the metadata of the cluster. Master node keeps track of the distribution and replications of the data in HDFS. All workers nodes send their status periodically to the master via heartbeats. These heartbeats contain information such as health of the chunk stored at that node, any discrepancies present

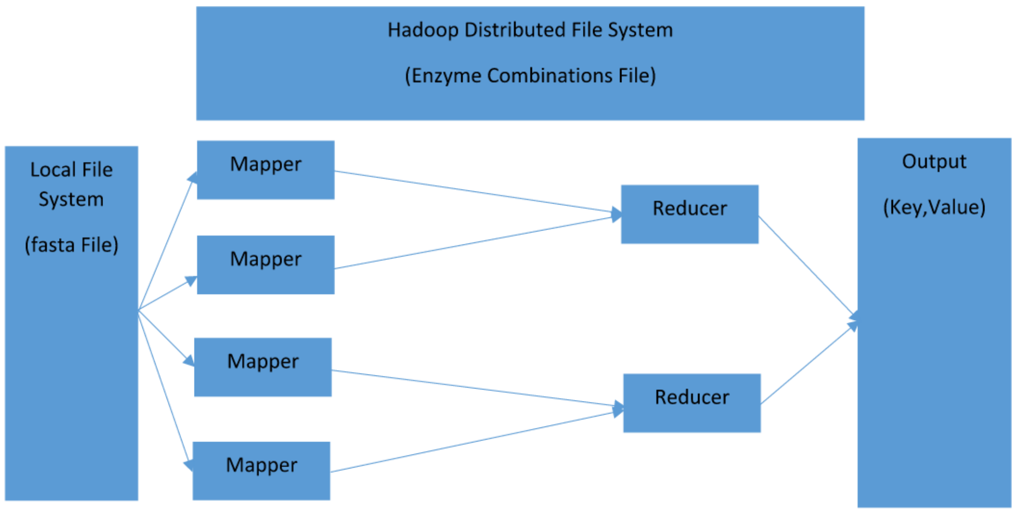


Fig. 5. System architecture - An implements of algorithm given in section 3 using Hadoop

in the data, stale data chunks and their information. If master nodes find out about the failure of any of the nodes in the cluster, the data stored at the failed node is replicated at another randomly chosen node. Once the failed node comes up, the temporary node which contains the data from the failed node is notified to delete the data.

Mapper: Mapper is the primary task executors or workers to whom, data is directly fed from the DFS. Mappers process the chunk of data assigned to them and produce intermediate key-value pairs. These intermediate key-value pairs are then stored in the main memory of the mapper until all data which has been assigned to mapper is processed completely.

Reducer: The Reducer is the second component of the MapReduce process. Reducer receives the intermediate key-value pairs from the mapper. Final processing is done on intermediate key-value pairs and the output is produced. Each reducer is associated with multiple mappers. The output of the reducer is stored in HDFS.

Combiner: At large scale, reducers are associated with large number of mappers. Due to this association, network traffic increases significantly. Chances of occurrences of a bottleneck at the reducer are very high due to such scalability issues. To solve these problems, combiners were introduced in MapReduce. Combiner process the intermediate key-value pairs received from mappers. Combiners are run on each mapper node. This intermediate component helps to reduce the amount of data that will be transferred over the network to the reducers.

Most of the Bioinformatics libraries are written in python. To take an advantage of these libraries, we have designed MapReduce implementation in python. Hadoop setup provides a Hadoop-streaming-2.3.0.jar which is used as an interface for mappers and reducers written in languages other than Java. In python, coding for the mapper and reducer relaxes the developer from worries such as types of the input key-values and output key-values. Also, standard input and standard output objects assist in accessing the HDFS and make changes to HDFS. We have used BioPython libraries to create the enzyme objects which allow us to use Bio.Python.Restriction class. Bio.Python.Restriction class contain many methods and variables that return properties of the restriction enzyme. Details of our MapReduce implementation are as given below:

4.1 Creating All Possible Combinations Of Enzyme Triads

We found out that there are 417 enzymes present in Biopython package having cutting frequency greater than or equal to 4096. To perform *in silico* digestion, we are using enzymes in the combination of three. To create distinct combinations of such triads, we wrote a python script which reads the enzymes file and gives a simple text file which contains all possible distinct combinations of triads of enzymes. We received around 12 million combinations of triads of enzymes, which were without any repetitions. This file is then distributed in the MapReduce cluster. When this file is distributed in HDFS, to assure the availability of the data, data is replicated three times. Also, it should be noted that, each node in the cluster is responsible for a certain amount of the total data distributed in HDFS.

We required to provide contigs file to our program program locally. Major reason for providing the contigs file through local system is the uniformity in processing data in HDFS. If both of these files were to be stored in the HDFS, it would have been a tedious task to identify the type of data that has been read into the mapper and multiple map reduces would have to be employed in order to get the streamline results. To avoid such cumbersome tasks, we have provided conigs file through the local file system. This approach will also help in testing all Contigs for a small sub set of enzyme triads that have been assigned to the mapper.

4.2 Mapper Implementation

The input to the mapper is received from the HDFS through the standard input object. Enzyme triads are received as input to the mapper. Each line read from the HDFS represents the triads of enzymes. We split them in three parts to get the three enzymes' strings. We have used the Bio.Restriction API to initialize the enzyme object. We passed the string which contains the name of the enzyme along with the Bio.Restriction API to the getattr method. This method returns the object of the enzyme. These objects will be used in performing *in silico* digestion on the contigs read from the fasta file. Once all enzyme are initialized, fasta file is open for reading. Each line in the fasta file represents a contig. To get the contig object from the line read from the fasta file, we have used Bio.SeqIO.parse() API. This API returns the a contig object. *In silico* digestion works at the time complexity of O(n), since each enzyme works on catalysis process on contig sequence. The output which is received from the catalysis of the contig sequence with first enzyme is then passed as an input to the catalysis process performed by the second enzyme. Same goes for the third enzyme. Output received from the catalysis performed by the third enzyme is then passed to the quantize method. Quantize method returns the quantized value. All these quantized values are stored in the fragment size array. This fragment size array is converted to unicode strings variable. (UTF-8 encoded) This variable is later used to construct the generalized suffix tree. On this generalized suffix tree, we performed a postorder traversal to find out the total number of shared suffixes. The total number of shared suffixes is then passed as a value along with the triad of enzyme that was read from the HDFS as the key to reducer.

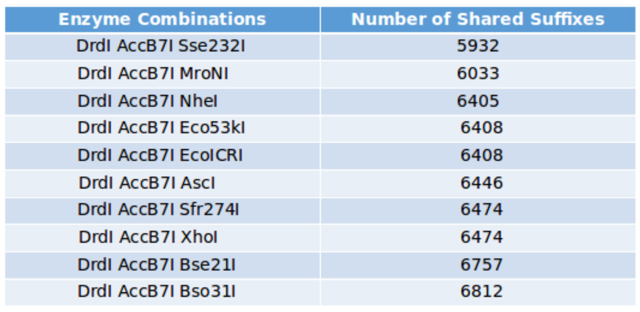
4.3 Reducer Implementation

Reducer reads the intermediate key-value pairs of the data produced by the mapper. Initially, the intermediate data is stored in the main memory of the node where mapper was being executed. Later, these key-value pairs are stored in the HDFS. We created an object list, which will store these key-value pairs. Once we have all pairs, which contain a combination of the triads of enzymes and the number of shared suffixes for the respective triad of enzyme; we have sorted them in ascending order. As per the requirement, we output top n enzyme triads and their number of shared suffixes.

5 EXPERIMENT AND RESULTS

We chose an Escherichia coli genome to test our software. We got E.Coli genome assembled using velvet assembler. It had 180 contigs with varying length from 200-50,000bp. We calculated all possible combinations for 417 enzymes, which comes up to be 11998480 combinations. But soon we realize that biopython library doesn't have classes for all 417 enzymes. So we sorted the enzymes for which classes don't exist in biopython library. We have used Dell Zhang's generalize suffix tree implementation that has a support for unicode (UTF-8 encoded). As this code support addition of only 36 strings to the generalize suffix tree, we had to change this implementation to support n addition number of strings to generalize suffix tree.

Figure 6 gives the results of our experiment. We calculated first 10 combination of enzyme triads. First column containing enzymes DrdI, AccB7I, Sse232I gives 5932 sub-strings which is a minimum value and hence combination of DrdI, AccB7I, Sse232I will produce an optical map that can be uniquely align to contigs.



Enzyme Combinations	Number of Shared Suffixes
DrdI AccB7I Sse232I	5932
DrdI AccB7I MroNI	6033
DrdI AccB7I NheI	6405
DrdI AccB7I Eco53kI	6408
DrdI AccB7I EcoICRI	6408
DrdI AccB7I AscI	6446
DrdI AccB7I Sfr274I	6474
DrdI AccB7I XhoI	6474
DrdI AccB7I Bse21I	6757
DrdI AccB7I Bso31I	6812

Fig. 6. Top 10 results

6 CONCLUSIONS

We have successfully implemented a tool to find the best possible combination of restriction enzyme to get a good optical map using MapReduce framework. Our software has been tested against the real data sets such as ecoli data set. We have shown that, it is possible to find a combination of enzymes that will output best optical map. As a part of this project, we distributed complex Bioinformatics computations over distributed environment. We also successfully adopted existing python library to build a suffix tree as per our need. It is an achievement because it was very complex. As a part of this project we wrote python scripts for mapper and reducers, which was a learning experience for us. We found the coding for mappers and reducers in python, is way easier than it is in Java.

There is a vast horizon for improvement and scope of this project. As future scope for testing the scalability of our software, we can use even larger and complex dataset like eukaryote genomes, Budgerigar genome, and human genome. while experimenting with different combination of enzymes triads, we found out that suffix tree implementation is slightly buggy. As a future scope, better implementation of suffix tree will help improving scalability and performance of existing project.

REFERENCES

[1]Ronen, R., Boucher, C., Chitsaz, H., Pevzner, P.: SEQuel: Improving the Accuracy of Genome Assemblies. *Bioinformatics* 28(12), 188?196(2012)

[2]Alkan, C., Sajjadian, S., Eichler, E.E.: Limitations of next-generation genome sequence assembly. Nat. Methods 8(1), 61?65 (2010)

[3]Lin, H.C., et al.: AGORA: Assembly guided by optical restriction alignment. BMC Bioinformatics 12, 189 (2012)

[4]Nagarajan, N., Read, T.D., Pop, M.: Scaffolding and validation of bacterial genome assemblies using optical restriction maps. Bioinformatics 24(10), 1229?1235 (2008)

[5]Martin D. Muggli, Simon J. Puglisi, Christina Boucher: Efficient Indexed Alignment of Contigs to Optical Maps, Algorithms in Bioinformatics, WABI (2014)

[6]Roberts RJ: Restriction enzymes and their isoschizomers, NuclAcidr Res, 17 (1989) r347-r415.

[7]Todd w. Sands, Michael l. Petras and Janny Van Wijngaarden, A computer program to assist in the choice of restriction endonucleases for use in DNA analyses, Int J Biomedcomput, 25 (1990) 39-52

[8]Medicago truncatula re-assembly using Optical Map, `http://jcvi.org/medicago/archives\_assembly.php`, 17-12-2014.

[9]Anantharaman, T., Mishra, B.: A probabilistic analysis of false positives in optical map alignment and validation. In: Proc. of WABI, pp. 2740 (2001)

[10]Valouev, A., et al.: Alignment of optical maps. J. Comp. Biol. 13(2), 442462 (2006)

[11]Zhou, S., et al.: A single molecule scaffold for the maize genome. PLoS Genet. 5(11), 1000711 (2009). doi:10.1371/journal.pgen.1000711

[16]Suffix tree, Wikipedia, `http://en.wikipedia.org/wiki/Suffix\_tree`, 17-12-2014.

[13]Generalized suffix tree, Wikipedia, `http://en.wikipedia.org/wiki/http://en.wikipedia.org/wiki/Generalized\_suffix\_tree, 17-12-2014.`

[14]Ukkonen, E. (1995). ”On-line construction of suffix trees”. Algorithmica 14 (3): 249260. doi:10.1007/BF01206331

[15]Ron Shamir and Roded Sharan, Suffix Trees, `http://www.cs.tau.ac.il/~rshamir/algmb/presentations/Suffix-Trees.pdf`, 17-12-2014.

[16]Dell Zhang, A suffix tree implementation with Unicode support `http://researchonsearch.blogspot.com/2010/05/suffix-tree-implementation-with-unicode.html`, 17-12-2014.