Compressing the Human Genome against a reference

Pragya Pande
prpande@cs.stonybrook.edu
Stony Brook University

Dhruv Matani
dmatani@cs.stonybrook.edu
Stony Brook University

December 8, 2011

Abstract

We present a way to compress the complete human genome against a reference genome to achieve *139x - 895x* compression. Our approach can potentially compress *BUILD 36.3* of the human reference genome against (about *2.6GiB*) *BUILD 36.1* (also known as *hg18*) to achieve a compression ratio of *139x* (about *19.2MiB*). The Korean genome *KOREF_20090224* (about *2.9GiB*) can be compressed against *KOREF_20090131* to achieve a compression ratio of *895x*, resulting in a file of size about *3.3MiB*[1].

The compressed differences are stored such that decompression from random offsets is fast. Furthermore, our *range format* for storing differences allows us to *efficiently* compute *transitive differences* between the target genome and different reference genomes. Our method does not rely on the target genome being almost of the same size as

[1] These results are extrapolated from tests we ran on a subset of chromosomes for each genome pair

the reference genome, or the existence of any alignment program available to us. We shall also prove that for the representation used, the technique we propose achieves optimal compression.

We shall first motivate the problem itself, show existing research that has been done in this direction, present our approach, and compare it with prior work.

Contents

1 Motivation

The Complete Human Genome was sequenced in 2003. Since then a lot of research is being done in genomics and computational biology. The major input for most of the computation is the 2.9 billion base pairs [2][3] of the human genome which correspond to a maximum of about 725 megabytes of human genome data.[4]

Furthermore, reduction in the cost of sequencing (via the "next-gen" sequencing platforms) has given birth to the 1000 genomes project [2] which aims to sequence the genomes of a large number of people. Just like the other human genome reference projects, this data (estimated 8.2 billion bases per day) would be made available via public databases for the scientific community.[5]

As we can now see, we are dealing with megabytes and megabytes of data when we work with genomes! This gives rise to challenging problems with respect to storage, distribution (downloading, copying), and sharing of this genomic data. Hence we need to consider better compression techniques for the genomic data. That apart, when working with genomic data, we want to be able to optimize decompression so that working with these compressed genomes is no harder than working with the uncompressed genome.

The goal of the 1000 genomes project is to find most genetic variants that have frequencies of at least 1% in the populations studied. Similarly, once the $1000 genome project[10] becomes successful, storage costs for all the sequenced genomes will need to be kept under control. To be able to do this, we need an space (and time) efficient way of compressing the sequenced genomes so that the DNA of more people can be sequenced for a reasonable price.

Multiple laboratories might use different reference genomes to compress the genomes they sequence. For example, a Korean lab might want to use a Korean genome as reference since it will be closer to the genome of other

[2] `http://www.1000genomes.org/`

Koreans. Other laboratories might have their own reference genomes. However, if labs want to share sequenced genomes, it becomes hard to share the difference compressed genome if the differences have been computed with respect to different reference genomes. One option is to decompress the sequenced genome and re-compress it using the target reference as the reference genome. However, since the compression is a time consuming process, we would like to be able to quickly re-encode the genome with reference to a different reference genome.

We explore the problem of genome compression and see if we can:

- Better align 2 human genomes so as to facilitate better compression on them
- Better compress the human genome to a smaller on-disk representation
- Enable faster decompression
- Enable space-efficient decompression
- Enable I/O efficient decompression so that working with the compressed genome is the norm rather than the exception
- Support *efficient transitive compression* of a genome with respect to multiple reference genomes

2 Existing Research

2.1 gzip & bzip2

These 2 applications are general loss-less compression routines that use run-length encoding and block sort compression respectively. They can compress the human genome (about 3GiB) to about 830MiB, which is a compression ratio of *3.67*.

2.2 GenCompress, BioCompress & Cfact

GenCompress[7], BioCompress[8] & Cfact[9] are tools that mostly rely on either Huffman Coding, Ziv-Lempel[6], or Arithmetic Coding to compress the human genome. They are able achieve better compression when compared with *gzip & bzip2*, but not as much as some of the difference encoding methods mentioned below. These methods achieve compression ratios of anywhere from *4.82 – 7.00*.

2.3 Difference Encoding Techniques

Difference encoding schemes are getting very popular since they can achieve very high compression ratios of greater than *100* since *99.9%* of the genomes of 2 humans are similar to each other. There are very few variations between the genomes of 2 individuals.

DNAzip[1][16] was the first algorithm to compress the target genome by storing differences between the target and reference genome. However, DNAzip does not solve the problem of aligning 2 genomes with each other, but instead assumes the existence of an *SNP – Single Nucleotide Polymorphism* file, which it takes in as input. In our experience, aligning 2 genomes is a very time consuming process and is orders of magnitude slower than performing the actual compression itself.

Wang & Zhang[11] have come up with a difference compression technique that is able to compress *2986MiB* of a Korean genome to *18.8MiB*. This was done by difference compressing the target genome against another Korean genome which was sequenced using similar methods and had the about the same size. their technique considers blocks of bases of size 50, 20 or 10 million, and groups each chromosome from both the target and the reference in these blocks to compute the difference between them. This may not always work with genomes that have sequenced and assembled using different techniques, or if you are using the human genome as a reference to compress the mouse genome for example.

Researchers at Virginia Tech, Blacksburg[12] have also come up with a difference compression algorithm that focuses on decompression speed rather than raw compression ratio. We are also motivated by some of the same factors that have motivated their research. The technique presented achieves decompression times of $O(\log n)$ per random offset query[3]. We also target a similar complexity for random offset queries. They haven't publish exact numbers on compressing the human genome, but claim that they achieve *98.8%* compression when compressing mitochondrial sequences, which roughly compresses the human genome of *3GiB* to *36MiB*. Their technique also relies on align sequences of chromosomes in the target with those in the reference genome. They use the multiple sequence alignment application MUSCLE[13] to align the sequences with the reference genome.

Researchers at Cairo University, Egypt[14] use existing local-sequence-alignment programs to align the target and reference sequence with each other. They then record insertions, deletions and modifications with respect to the reference sequence and encode these differences in a compact fashion. They have achieved *99.4%* compression on the human genome, which means that they can compress *3GiB* of genomic sequence data to about *18.5MiB*. However, they don't talk much about decompression, and it seems as if it is not entirely easy to decompress to random offsets in the target genome. i.e. Ask for a sequence of length l at offset k from the uncompressed genome.

3 Our Solution

3.1 Outline of the Approach

We compress the genome a chromosome at a time. Ref_i is our reference chromosome and $Victim_i$ is the chromosome to be compressed.

We shall find the optimal (least) number of ranges from Ref_i that can com-

[3] We haven't yet done a High Probability analysis on the number of disk blocks scanned to resolve a range of a certain length k

pletely cover $Victim_i$. To do this, we shall construct a Suffix Array on Ref_i & $Victim_i$ and then find the least range covering. We shall prove that for an encoding that only encodes *(offset, length)* pairs from the reference, this representation is optimal in terms of the number of ranges needed to completely cover the entire target chromosome.

Subsequently, the plain-text file containing these optimal number of ranges is encoded in a binary format and the resulting binary file is further compressed using *gzip* to produce the final compressed difference file.

Hence, it is easy to see that our approach will provide good compression as long as there is a sufficient amount of overlap between the target and reference genomes even if:

- The 2 genomes have different lengths.
- The genomes have been sequenced and assembled using different apparatus and methods.
- You try to compress the mouse genome with a human genome as a reference.

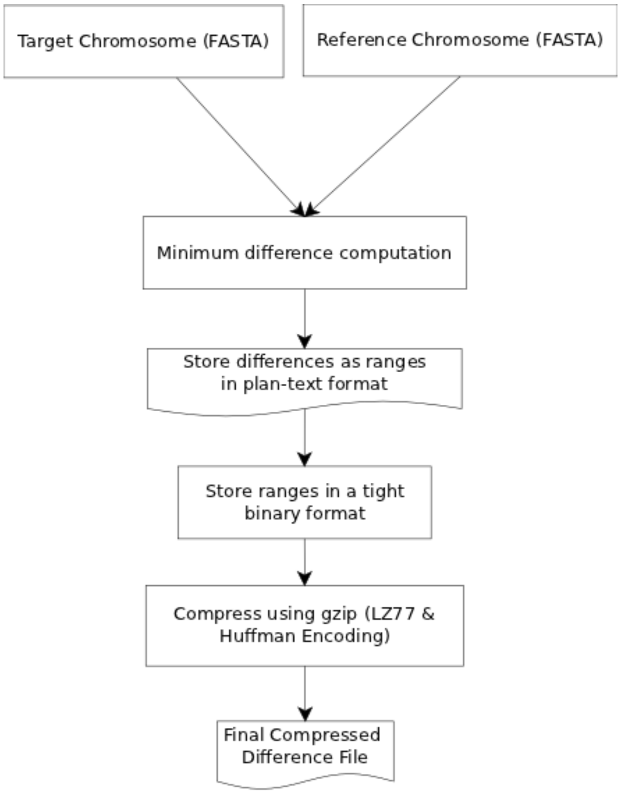
Target Chromosome (FASTA)
Reference Chromosome (FASTA)
Minimum difference computation
Store differences as ranges in plan-text format
Store ranges in a tight binary format
Compress using gzip (LZ77 & Huffman Encoding)
Final Compressed Difference File

Figure 1: The flow of Genome Compression

3.2 The Algorithm

1. For each corresponding chromosome pair Ref_i & $Victim_i i \in [1 \ldots 23]$, build a *suffix array*[4] from the string $Ref_i \# Victim_i \$$.

2. For every suffix in $Victim_i$, find the length of the longest prefix that matches with some string at index j in Ref_i.

3. Now that we have all the ranges in Ref_i that $Victim_i$ can be covered by, we find the least number of ranges that can completely cover $Victim_i$. This can be done in $O(n \log n)$[5] time using *Dynamic Programming* and *Segment Trees* as an online *Range Minimum Query* data structure. The problem of finding the minimal number of completely covering sub-ranges exhibits optimal substructure. i.e. A solution for the range $[i \ldots n]$ can be constructed using the solution to the ranges $[i+1 \ldots n], [i+2 \ldots n], [i+3 \ldots n], \ldots, [i+k-1 \ldots n]$, where k is the length of the range starting at index i.

3.3 Transitive Difference Computation

Suppose V is the victim genome that is already compressed with reference to the reference genome R_1, and we want to compress it with respect to reference genome R_2, then we need to compute how genome R_1 compress with respect to genome R_2. Since we expect to compress many victims with respect to a fixed number of variable references, we can ignore the cost of computing this difference. Besides, this is just a one time process and once the delta $R_1 - R_2$ has been computed, we can reuse it indefinitely.

To be able to compute transitive differences, we need to slightly modify the output of the compression routine. Instead of storing just the tuple

[4] This is done using Manber & Myers $O(n \log n)$ suffix array construction algorithm[15], which allows us to compute the *Longest common Prefix* of 2 adjacent suffixes in $O(\log n)$ time

[5] There is also a (single-pass) $O(n)$ time algorithm to do the same, but we don't talk about it here

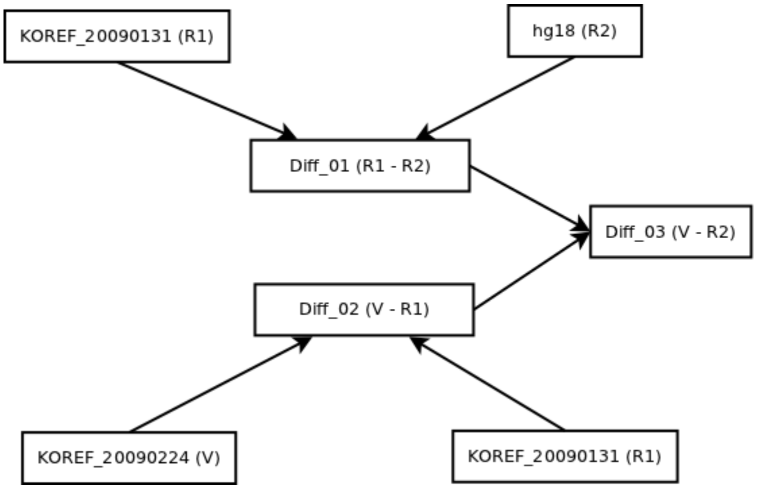

Figure 2: Transitive Difference Computation

(offset, length), we store the tuple *(offset, length, span)*, where *span* is the actual span of the range (length of maximum match) between the victim and reference. This span is the value computed using the suffix array and longest common prefix algorithm.

Now, given the files $V - R_1$ and $R_1 - R_2$, we need to find, for every offset i in V, in index j in R_1 where it can be found. Once we know the value of j, we lookup $R_1 - R_2$ to locate the index k in R_2 of the index j in R_1. This lookup need not be done on a per index basis, but can instead be done on a range basis. This is an optimization that reduces the running time of this algorithm.

Each lookup into the difference file $R_1 - R_2$ costs $O(\log n)$, (where n is the number of lines in $R_1 - R_2$) and we need to process $O(m)$ such ranges (since ranges may get split during this operation). Here, m is the total number of lines in the file $V - R_1$. Hence, the total running time of the transitive difference computation routine is $O(m \log n)$.

We must mention however that the differences computed using the transitive difference routine are *not* optimal.

3.4 Proof of Optimality

We shall prove by contradiction that for the range based encoding scheme we have used, the algorithm presented above achieves optimal compression in terms of the number of ranges used to represent the completely covered *victim* chromosome.

Given the following range lengths starting at the indexes at which they occur, what is the least number of ranges that completely cover the whole range?

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Range Length	3	2	4	1	1	1	3	6	8	2	1	2	1	1

Figure-3: Indexes and lengths of ranges at those indexes

The best solution extends the smallest range that it can cover to its right.

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Range Length	3	2	4	1	1	1	3	6	8	2	1	2	1	1
Min n(Ranges)								2	**1**	3	3	2	2	1

Figure-4: If we have the solution till index 7, and we want to extend it to compute the solution till index 6, then we scan up to 3 (the length of the longest range at index 6) places to the right of index 6 (shown in red), locate the minimum value in that range (shown in bold face), and add 1 to it.

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Range Length	3	2	4	1	1	1	3	6	8	2	1	2	1	1
Min n(Ranges)	4	4	3	5	4	3	2	2	1	3	3	2	2	1
Next Index	3	2	6	4	5	6	8	8	14	10	11	13	13	14

Figure-5: The numbers in red indicate the range end-points that make up the ranges that cover the entire sequence. If n numbers make up the range, then there exist *(n-1)* ranges. Range extension stops when we have gone past the last index in the sequence (14 in this example).

If there is a better solution at any stage, then it would have been the one chosen by our algorithm for extending a range from a given index.

3.5 Binary Encoding

We use a tight binary format to represent the *(offset, length)* pairs into the reference that are needed to represent the target genome.

Each line in the intermediate plain-text range file holds an *(offset, length)* pair that needs to be encoded. Since offsets can be quite large, we always encode them as fixed 32-bit integers. The length values however have a different distribution. For a chromosome we compressed, here is the distribution of length values.

Range of Values	[1…127]	[128…16383]	[16384…]
Counts	56620502	60686338	2602
Percentages	48.27%	51.73%	0.0022%

Figure-6: Distribution of length values in the compressed ranges file.

All values in the range [1…127] are encoded as a single byte integer with the MSB set to `1`. Values in the range [128…16383] are encoded as 2-byte integers, with the first 2 most significant bits set to `01`. Values greater than 16383 are encoded as 4-byte integers with the first 2 most significant bits set to `00`. This allows us to uniquely identify the size of the value based just on the value of the 2 most significant bits of the integer.

3.6 Decompression

The difference encoded file looks like this:

```
61771 51
7829  89
```

```
17288 12
7288  6523
```

These are ranges with lengths 51, 89, 12, & 6523. We construct an intermediate file containing the cumulative lengths to allow quick random retrieval. The pre-processed data looks like this:

```
61771 0
7829  51
17288 140
7288  152
--    6675
```

Now, to get the decompressed chromosome data of length 100 at offset 130, we need to just do a *Binary Search* into the cumulative length array and find the location of offset 130. This happens to be at offset 7829+79 in the reference file. The range of length 200 happens to span 3 ranges, one is the previous range, and the other ones are at offset 17288 with length 12 and offset 7288 with length 109. Since all the blocks in the reference come from the same chromosome, we can prevent disk seeks by loading the whole chromosome into memory before starting decompression (or alternatively incur a cost of a few disk seeks per retrieval). If we consider the case where we have many genomes compressed with respect to a fixed reference genome, then it is sufficient to just keep that genome, and all the difference files in memory to ensure fast random & sequential decompression.

3.7 Practical Considerations

Every chromosome of the target genome was stripped off any *N* characters (unknown base-pair) before compressing it against the reference genome's corresponding chromosome. These unknown base-pairs mostly occur at the beginning and end of each chromosome.

We ran the experiments on a 32-bit VMWare Player Virtual Machine running Ubuntu 11.04, with Windows 7 as the Host Operating System. Since the guest is a 32-bit system, the process address space is limited to 4GiB, out of which about 1.5GiB of the higher address space is used up by the kernel and dynamically loaded libraries such as *glibc*, etc.... This leaves us with 2.5GiB to use. When constructing the Suffix Array, we have observed that there are rarely sub-strings greater than *65,536* in length that overlap in 2 genomes. This means that we can cap the extra space required by the suffix array construction algorithm to $16n \times sizeof(int)$. An additional $3n$ space overhead is incurred to store the input string and a $5n \times sizeof(int)$ overhead is incurred while computing the optimal overlap of ranges.

All this adds up to (in bytes):
$16n \times sizeof(int) + 3n + 5n \times sizeof(int)$
$= 16n \times 4 + 3n + 5n \times 4$
$= 64n + 3n + 20n$
$= 87n$

Hence, the space requirement for our algorithm is $O(n)$, with a fairly high constant overhead.

Equating the 2 sides, we get:
$2.5 \times 1024^3 = 87n$
Solving for n, we get:
$n = 30854650$
which translates to about *30 million*.

Hence, we can process only strings of length *30 million* at a time with the process address space that we have.

Experimentally, we determined that we got the best results if we chose *16MiB* of data from the reference chromosome & *6MiB* of data from the target chromosome (totaling *22MiB* in all). We had to rune some more experiments to find out the best way to align these blocks so that maximum compression for the target could be achieved. We found that we occasionally need to align each block of *6MiB* with potentially *2–3* blocks of size *16MiB* from the reference to get the best compression. Generally *uncompressed plain-text* difference files for *6MiB* from the target range anywhere from *90KiB to 300KiB*. If we see results in this range, we don't try other ranges.

This *block based* processing of the chromosomes results in sub-optimal compression, but:

- Allows us to work around process address space limits even though the machine might have enough physical memory.
- Results in faster decompression since the hard disk needle need not seek too far (the 16MiB block from which the differences are generated is all that is needed to decompress a 6MiB block). This makes decompression a very *I/O-efficient* process.

Compressing a *full* target chromosome against another *full* reference chromosome will result in better compression, but will require more memory during decompression (or will result in more disk seeks if available memory during decompression is bounded).

You can see that by varying these block sizes, one can achieve a balance between compression ratio and decompression speed/memory required to compress & decompress a chromosome.

4 Approaches Compared

4.1 DNAzip

The DNAzip group relies on an existing *SNP* file being present, and hence they don't do sequence alignment before compressing the difference file.

It isn't possible to trivially get the decompressed genome data at a certain *(offset, length)* pair without decompressing the complete genome.

4.2 Wang & Zhang's method

Wang & Zhang's method is called *The GRS tool* and it works in blocks of size 50, 25, or 10 million. It relies on being able to compute the longest common sequence between 2 blocks. It isn't immediately apparent how easy it would be to support fast & efficient random offset querying on the compressed data generated by *The GRS tool*.

4.3 Virginia Tech Research

The approach that the Researchers at Virginia Tech, Blacksburg[12] use is one of using existing sequence alignment programs such as MUSCLE[13] to do the heavy lifting of aligning sequences with each other and then encode the differences using one of the following 5 operators:

1. insertion
2. deletion
3. replacement
4. insertion after replacement, and
5. deletion after replacement

These difference operations are then tightly compressed using *Huffman Compression* to produce the final output.

The highlight of their approach is that they support very fast random offset decompression.

4.4 Tabulated Comparison

Method	Does Sequence Alignment	Works on FASTA	Human Genome Compressed to	Comp. Ratio	Fast Decompression	Random Offset Querying
DNAzip	No	No	4.1MiB	724	Yes	No
GRS	Yes	Yes	18.8MiB	159	Yes	Not clear
VTech	Yes	Yes	36MiB	85	Yes	Yes $O(\log n)$
Cairo	Yes	Yes	18.5MiB	166	Yes	No
Proposed	Yes	Yes	22-3.3MiB	139-895	Yes	Yes $O(\log n)$

Figure-7: A comparison of various approaches

5 Experimental Results

When run on *hg18* as the reference genome and *BUILD 36.3* of the human genome reference as the target genome, here are the results we got for some chromosomes.

Chromosome	Original (MiB)	Compressed (KiB)	Compression Ratio
14	83	554	153.4
15	73	562	133.0
20	57	387	150.8
21	32	219	149.6
22	32	243	134.8

Figure-8: Compression of certain human chromosomes.

When run on *KOREF_20090131* as the reference genome and *KOREF_20090224* as the target genome, here are the results we got for some chromosomes.

Chromosome	Compressed (KiB)	Compression Ratio	GRS Compressed (KiB)	GRS Compression Ratio
16	97	918.43	554.7	158.9
17	86	893.02	494.1	158.3
18	73	1009.97	399.0	189.4
22	48	682.67	256.3	192.6

Figure-9: Comparison of compression achieved by our method v/s that achieved by the *GRS Tool* on certain chromosomes sequenced from Korean individuals. Chromosome 16 was NOT stripped off the *N* characters to show that removing these characters doesn't impact the compression ratio.

6 Future Work

Compressing the genome *6MiB* at a time against *16MiB* of data from the reference takes about *15.5 minutes* on a single core (CPU).

This computation can be parallelized across multiple cores. If the compression is being done a chromosome-at-a-time, then each chromosome can be processed on a separate CPU so that the total time required is the *maximum* of the time required to compress each chromosome rather than the *sum* of their times.

7 Acknowledgments

We would like to thank Dr. Skiena for giving us the opportunity to work on this project and rightly penalizing us for a less-than-optimal project proposal & progress report. This made us work harder on the project. Though, we would like to believe that the real reason for our earlier lethargy was the phenomenon of *Temporal Discounting*[6]

[6] `http://en.wikipedia.org/wiki/Temporal_discounting`: Temporal discounting refers to the tendency of people to discount rewards as they approach a temporal horizon

We would also like to thank Aniruddha Laud, Gaurav Menghani, & Madhava K R, who patiently explained the working of DNAzip[1] to us.

in the future or the past (i.e., become so distant in time that they cease to be valuable or to have additive effects).

References

[1] DNAzip: DNA sequence compression using a reference genome http://www.ics.uci.edu/~dnazip/

[2] How much of the human genome has been sequenced? http://www.strategicgenomics.com/Genome/index.htm

[3] Finishing the euchromatic sequence of the human genome http://www.nature.com/nature/journal/v431/n7011/abs/nature03001.html

[4] Wikipedia on The Human Genome http://en.wikipedia.org/wiki/Human_genome

[5] The 100 genomes project aims to provide a comprehensive resource on human genetic variation. http://www.1000genomes.org/about

[6] A Universal Algorithm for Sequential Data Compression, IEEE Trans. Information Theory, vol. 23, pp. 337-343, 1977.

[7] DNABIT Compress – Genome compression algorithm. Pothuraju Rajarajeswari and Allam Apparao

[8] A New Challenge for Compression Algorithms: Genetic Sequences, S. Grumbach and F. Tahi, Information Processing Management, vol. 30, no. 6, pp. 875-886, 1994.

[9] A Guarantee DNA Sequences for Repetitive DNA Sequences, E. Rivals, J.P.Delahaye, M.Dauchet, and O.Delgrange, LIFL Lille I University, Technical Report IT-285, 1995.

[10] Anticipating the 1,000 dollar genome, Mardis ER

[11] A novel compression tool for efficient storage of genome resequencing data. Congmao Wang and Dabing Zhang

[12] A Genome Compression Algorithm Supporting Manipulation. Lenwood S. Heath, Ao-ping Hou, Huadong Xia, and Liqing Zhang

[13] MUSCLE: Multiple sequence alignment with high accuracy and high throughput, R. C. Edgar, Nucleic Acids Research, 32 (2004), pp. 17921797.

[14] DNA Lossless Difference Compression Algorithm Based On Similarity Of Genomic Sequence Database, Heba Afify, Muhammad Islam, and Manal Abdel Wahed

[15] Suffix arrays: A new method for on-line string searches, Udi Manber and Gene Myers

[16] Human genomes as email attachments. Scott Christley, Yiming Lu, Chen Li, Xiaohui Xie