ALGORITHMS AND DATA STRUCTURES TO DETECT ONCOVIRUSES IN HUMAN CANCER USING NEXT GENERATION SEQUENCING DATA

A Thesis Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Rui Zhu

December 2012

ALGORITHMS AND DATA STRUCTURES TO DETECT ONCOVIRUSES IN HUMAN CANCER USING NEXT GENERATION SEQUENCING DATA

_____

Rui Zhu

APPROVED:

_____

Dr. Yuriy Fofanov, Chairman

_____

Dr. William Widger

_____

Dr. Nikolaos Tsekos

_____

Dr. Mark A. Smith, Dean

College of Natural Sciences and Mathematics

ACKNOWLEDGEMENTS

My sincerest appreciation goes to Dr. Yuriy Fofanov, Ph.D., for his guidance and support throughout this endeavor. I would also like to thank Georgiy Golovko, Mark Rojas, Otto Dobretsberger, Meenakshi Sharma, and Efren Ballesteros for their technical efforts and suggestions, without which this project would never have reached fruition. Last, but certainly not least; my deepest gratitude goes to my parents and my husband for their constant encouragement and total support in my attainment of this goal.

ALGORITHMS AND DATA STRUCTURES TO DETECT ONCOVIRUSES IN HUMAN CANCER USING NEXT GENERATION SEQUENCING DATA

A Thesis Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Rui Zhu

December 2012

ABSTRACT

Evidence suggests human cancer can be induced by viruses. One way to test this hypothesis is to look for viral sequences in the human cancer genome. Next Generation Sequencing (NGS) technology sequences the whole human genome in a short period of time. This opens a door for a systematic analysis of the human genome and a thorough search for oncogenic viral sequences in cancer. However, a huge amount of sequencing reads generated by NGS poses a great challenge on the computational part of data analysis in terms of computing speed and memory usage. Data structures such as hash and tree are widely implemented to improve the performance of computing algorithms. Here, I described both data structures that have been developed in our center and compared their performance. Hash out performed tree when mapping the reads to a small reference sequence database. Subsequently, real human cancer data were analyzed by using the hash-based mapper and different oncoviral sequences were found in different cancers.

CONTENTS

1 Introduction

1.1 Next Generation Sequencing (NGS)

Next Generation Sequencing (NGS) is a high through-put sequencing technology, in which genomic material (i.e. DeoxyriboNucleic Acid or DNA) of organisms can be sequenced in a massive parallel way. NGS produces tens of billions of short sequence reads in a few days, which is considered lightening speed when compared to the Sanger Sequencing. The Sanger sequencing is also called the first generation sequencing, which was used by the Human Genome Project to sequence the whole human genome for the first time. It took 20 institutes all over the world 13 years to complete a draft of the whole human genome, whereas NGS takes only a few days to a week to sequence a whole human genome.

1.2 The search of infectious agents in cancer benefits from NGS technology

Studies [1] suggest that infectious agents, such as viruses, bacteria, parasites, etc. are causative to human cancer. However, due to the long latency to malignancy and technical difficulties in whole genome profiling using first generation sequencing methods (automated Sanger sequencing) [2-4], the direct evidence was hard to find. The Next Generation Sequencing (NGS) technique enables the profiling of the whole human genome and makes a systematic search of non-human genetic material feasible. With NGS data available in the lab, I focused on assessing the role of viruses in human cancer.

Study shows there are tens of known virus species or oncoviruses that can cause cancer. These oncoviruses are identified using several techniques. For example, Merkel cell polyomavirus was recently identified by digital transcriptome subtraction [5], in which the human sequences were subtracted from the tumor transcriptome computationally. The analysis suggests that oncoviruses are transcribed in the cancer tissue and thus their genomic sequences are present there. Therefore, I hypothesized that the profiles of identified oncovirus species in the genomic DNA samples from cancer inflected tissue differ from normal tissue. To test this hypothesis, I constructed an oncovirus reference sequence database and mapped NGS reads from samples of brain, colon, and cervical cancer to this database. Then I compared the mapping result between sequences generated from cancer samples and their normal counterparts.

1.3 Computational challenges in NGS data analysis

A few decades ago, biologists were sipping the information from cups of data produced in traditional biological experiments and as a consequence there was no need for computer scientists to help with data processes in a large scale. With the development of sequencing technology, the amount of the data produced in modern biological sequencing experiments has increased dramatically. Biologists are now flooded by the overwhelming amount of data. This has posed challenges on many aspects of computer science, such as data storage, data transfer, as well as data analysis and interpretation.

My project was to search for evidence of oncoviruses in the cancer genome compared to the normal human genome. A critical step of data analysis was to map NGS reads to the oncoviral database and identify any viral sequences from the pool of mapped reads.

Therefore, algorithms and data structures for the mapping (alignment) of short reads are important for my project.

Many mapping algorithms and data structures have been developed. In summary, there are two major categories:

(1) Hash-based algorithms

Some of these work by hashing the read sequences and scan through the reference sequence, such as human genome sequence or oncoviral database in my case. Some examples are RMAP [6], MAQ [7], ZOOM [8], SeqMap [9], and CloudBurst [10]. Others programs work by hashing the reference sequences. For instance, SOAPv1 [11], PASS [12], MOM [13], and ProbeMatch [14].

(2) Tree-based algorithms

This type of algorithm makes use of the theory on string matching using Burrows-Wheeler Transform (BWT) [15] and put the reference genome in a prefix tree. Examples are SOAPv2 [16], Bowtie [17], and BWA [18].

In order to assess the efficiency of different mapping algorithms, I compared the performances of exact matching using the hash-based algorithm and the tree-based algorithm that were developed in our lab at Center for BioMedical and Environmental Genomics.

2 Algorithms and data structures

2.1 Construction of oncoviral database

First, I performed internet and literature search for all known oncovirus species. This search produced 339 virus names including many different subtypes. In order to obtain the genomic sequences from all virual strains of all species, I eliminated the restriction of subtypes and condensed the list to 34 search keys (Table1). Then I implemented a Perl script to extract genomic sequences that match the search keys from an existing comprehensive C_Sequence-formated virus database in the lab. The Perl script algorithm constructed a regular expression of search keys. The script writes the first line, the following information and the sequence to a new text file when there is a match to the regular expression . This search generated 6574 complete oncovirus genomes. The large numbers was due to different substrains of same viruses.

Table1. Search Keys for construction of oncoviral database.

Avian erythroblastosis virus	immunodeficiency virus	myeloproliferative leukemia virus
B10 xenotropic virus 1	Human Mammary Tumor Virus	NZB virus
C3H/Fg virus 2	Human papillomavirus	Pogo virus
C58 endogenous ecotropic virus	Human polyoma viruses	Rous sarcoma virus
Duplan murine retrovirus	Human T-cell lymphotropic virus	salmon swimbladder sarcoma
Epstein-Barr virus	Kirsten murine sarcoma virus	virus
Finkel-Biskis-Reilly murine	mammary tumor virus	Simian sarcoma virus
sarcoma virus	MC29 avian myelocytoma virus	Simian virus 40
Harvey rat sarcoma virus	mink cell focus-forming	T region endogenous virus
Hepatitis B virus	endogenous virus	v-crk sarcoma virus
Hepatitis C virus	Moloney murine sarcoma virus	xenotropic mink cell focus-
Hepatitis D virus	murine leukemia virus	forming leukemia virus
Human herpesvirus	Murine Polyoma virus	

2.2 Mapping NGS reads to oncoviral database

2.2.1 Preprocessing NGS reads

NGS reads produced by the sequencing machine are initially in FASTQ format. The reads used in this study are from the Illumina GAIIx sequencer, which produces FASTQ reads of the same length. Subsequently, FASTQ read files are passed through quality analysis using the programs, Pipeline for Illumina Quality Assessment (PIQA) or Technology Independent Quality Assessment (TIQA). Filtering decisions were made based on the quality assessment report. Reads with bad quality scores, or reads with unknown characters were removed. Sometimes, reads with biased prefix were trimmed to remove adapter sequences while retaining the informative part of the sequences.

After filtration, an output text file containing every unique read sequence and corresponding copy number in the pool of reads was produced. This output file is in an Array_Subsequence or AS format. The AS file is the input file for reads in both the hash-based and the tree-based mapping programs presented below.

2.2.2 Hash-based mapping algorithm

SLIM is the software package that has been developed in our lab using a hash-based mapping algorithm. The basic idea is to hash the read sequences into a hash table and scan through the reference sequence with the same sequence lengths as the original data to find the exact matches in the hash table of reads. There are two major parts of this algorithm. The first part is the conversion of each unique read sequence into its numeric

signature. The second part is to use a hash function to put the numeric signature of each read sequence into a hash table as a key and the copy number of each read sequence as a value.

2.2.2.1 Signature conversion of read sequences

The idea of transforming read sequences to numeric signature came when implementing a hash function. The first question is how to hash a string of characters in sequence reads with a 4-alphabet set ('A', 'T', 'C', 'G') to a hash table. There are many ways of doing this. For example, one can use the ASCII value of each of the four letters and add them up to get a number. However, one of the potential problems would be the loss of uniqueness of the sum of the ASCII value. To produce a unique set of signatures for each sequence in the sample, a schema of the signature conversion using bit operations has been developed in the lab (Figure 1).

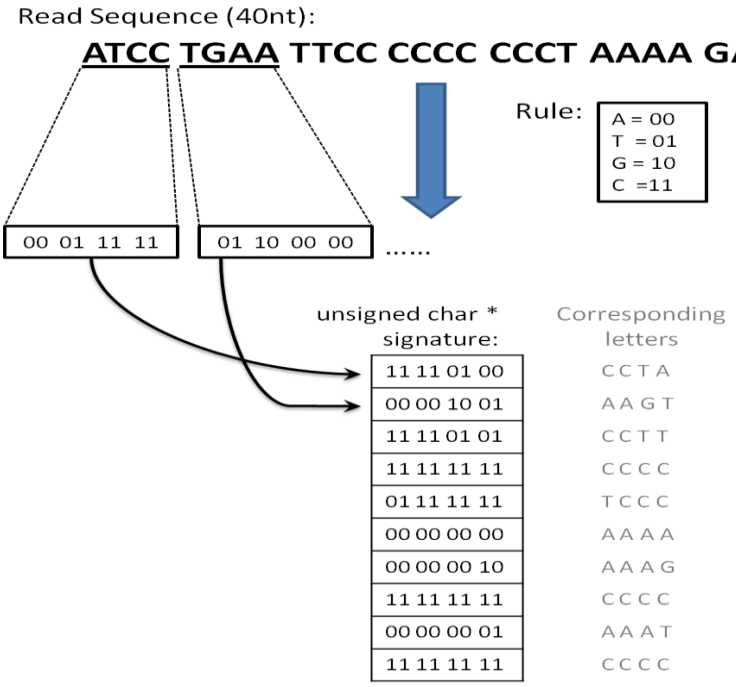


Figure 1. Schematic representation of conversion from a DNA read sequence to a numeric signature.

For this conversion, a 40-nucleotide (nt) long read sequence is converted based on the character-to-binary rule as 'A'→ 00, 'T'→ 01, 'G'→ 10, 'C'→ 11, Figure 1. In this way, it is sufficient to represent each letter in two bits and where a one byte character space (8 bits) is able to store the information of four nucleotides (e.g. "ATCC" can be stored in one byte as 00011111). In order to store the signature of an entire read sequence, an array of characters is created with a flexible size according to the read length (i.e. 40/4 =10 in this case) and each byte storing a 4-nucleotide signature.

In our current code, the 4-nucleotide signature conversion generates a reversed signature. For example, "ATCC" will be converted to 11110100 instead of 00011111. This is due

to the current conversion algorithm (Pseudocode 1). However, it does not affect the accuracy and performance of the conversion.

Pseudocode 1. Signature conversion for hash-based mapper.

```
#signature conversion

Signature_Size =  Read_Length / 4
IF Read_Length % 4 != 0
        Signature_Size++
ENDIF
FOR i in 1 to Signature_Size
        Signature[i] = 0
ENDFOR
FOR i in 1 to Read_Length
        SWITCH Read[i]
                CASE 'A' or 'a': x=0
                CASE 'T' or 't': x=1
                CASE 'G' or 'g': x=2
                CASE 'C' or 'c': x=3
                DEFAULT: RETURN
        ENDSWITCH
        x = x << (i%4)*2
        Signature[i/4] += x
ENDFOR
```

One of the advantages of developing a unique signature for each non-redundant read sequence is to minimize memory usage. Once the reads have been converted, there is no need to store read sequences. Signatures can be easily converted back to read sequences

by following specific rules. Also, the storage of numeric signatures is much more compact than storing character strings. For example, to store a 40-nucleotide (nt) long read sequence as a character array, one would require 40 bytes for each character plus one byte for "\0" at the end. When converted to a numeric signature, one byte can store the signature of four characters and only 10 bytes are needed. Numeric conversion produces a 4-fold reduction in memory usage.

Another advantage of using a numeric signature is that computation speed increases dramatically. The conversion itself does not impose much overhead due to the efficiency of bit operations. However, a big portion of time will be saved by reducing the comparison between two m-long character arrays ($O(m)$, m is the length of one read sequence) to the comparison between two integers ($O(1)$) during the mapping process.

2.2.2.2 The hash function and collision resolution

2.2.2.2.1 Determine the size of the hash table

The size of the hash table is determined by the number of reads and the population rate of the table. The number of reads to be hashed is provided by the input read file. The population rate is 0.25 by default and can be changed when needed. Once these two parameters are known, the size of the hash table is determined by using the nearest prime number of the anticipated maximum capability of the hash.

$$anticipated \max capability\ of\ the\ hash = \frac{(the\ number\ of\ reads)}{(the\ population\ rate\ )}$$

The reason to choose a prime number as the final size of hash table is to enable linear probing for collision resolution.

2.2.2.2.2 The data structure of the hash table

Once the size of the table is determined, two parallel arrays can be allocated: one is the array of keys and the other is the array of values. The array of keys in our application contains the numeric signatures from all converted read sequences. As explained in the Figure 1, each signature is an array of characters based on the length of reads. Therefore, the array of keys is a two dimensional array. The array of values stores the copy number of each read and is of same size and index as the array of keys.

2.2.2.2.3 The hash function

Upon successful signature conversion from the reads, the job of the hash function is to ensure all the numeric signatures of the reads can be hashed into the hash table. The modulus operation is sufficient for the first index in the hash table:

$$h(Signature) = Signature \ \% \ (tableSize)$$

Where $h()$ is a hash function. When Signature is converted from up to 32nt-long reads, it means that the size of the character array of Signature is 8 bytes or less. In this case, Signature can be easily converted to a 64-bit long unsigned integer number to be used to compute the index using the above function. However, when Signature is converted from reads that are more than 32nt-long, only the first 8 bytes of the Signature can be used in the above function. Then the remaining bytes have to be compared when collision happens.

2.2.2.2.4 Collision counter and collision resolution

Collision is monitored by using the variable, Collision_Counter. The Collision_Counter keeps track of the number of collisions and monitors the performance of the hash. One way to resolve the collision is to use linear probing:

$$h_i(Signature) = [h(Signature) + c(i)]\% n$$

$$for\ i = 0, 1, \quad \ldots, n - 1$$

Where $h()$ is a hash function, n is the size of hash table, and i is the number of collisions.

$c(i)$ is a linear function in i of the form: $c(i) = a * i$, which is has the following two properties:

Property 1: $c(0) = 0$

Property 2: The set of values $\{c(0)\ \%\ n, c(1)\ \%\ n, c(2)\ \%\ n, \ldots, c(n-1)\ \%\ n\}$ must be a permutation of $\{0, 1, 2, \ldots, n - 1\}$, that is, it must contain every integer between 0 and $n - 1$ inclusive.

In order to satisfy Property 2, a and n need to be relatively prime. Since n is chosen as a prime, therefore, we choose to have $a = 2^4 = 16$ here (Pseudocode 2).

Pseudocode 2. Collision resolution for hash-based mapper.

```
#collision resolution

'''Index here is the first index created by the hash function in above section'''
Collision_Number = 0
FOR Collision_Number in 0 to MAX_COLLISIONS_NUMBER
        IF Keys[Index] is empty
                Keys[Index] set to current Signature
                Values[Index] set to current CopyNumber
                increment NumberOfElementsInHash
                RETURN
        ENDIF
        ELSE
                compare the Signatures byte by byte
                IF equal
                        add the current CopyNumber to Values[Index]
                        RETURN
                ENDIF
                Index = (Index + ((Collisions_Number+1)<<4)) % SizeOfHashTable
                Increment Collision_Counter
        ENDELSE
ENDFOR
```

2.2.2.3 Operations

The purpose of creating the hash table is to enable quick searching of subsequences in reference genomes and retrieving of the copy numbers corresponding to each subsequence. Therefore, once the hash table is created, no deletion operations will be performed. The insertion and lookup operations are sufficient to serve our purpose here.

To insert a particular read sequence r, one would examine array location $h_0(r) = h(r)$. If there is a collision, array locations $h_1(r), h_2(r), \ldots, h_{n-1}(r)$ are examined until an empty slot is found. Similarly, to lookup a particular subsequence in the reference genome, one would examine the same sequence of locations in the same order until a matched subsequence is found. The time complexity of linear probing is dependent on the maximum number of collisions. In the worst case, all n sequences are hashed to one location, which makes the insertion and lookup $O(n)$ operations. The space complexity of m-nt long reads with a population rate p ($0 < p \leq 1$) is $O(\frac{mn}{4p})$ due to actual storage of Signature of reads is $\frac{m}{4}$ bytes.

2.2.3 Tree-based mapping algorithm

The SCRUFFY software package developed in the lab implements a Tree-based mapping algorithm. The idea is to first sort all the read sequences alphabetically and put them into a radix tree, to scan through the reference sequences to find exact matches and retrieve copy numbers for each mapped read from the tree.

2.2.3.1 Sorting the reads alphabetically by LSD radix sort

One feature of our mapping is that the read sequences in one mapping task are of same length. This allows the use of a least significant digit (LSD) radix sort on each letter of the reads.

The basic sorting operation of radix sort is a counting sort, which is not a comparison sort. The $\Omega(n \log n)$ lower bound for comparison sorting is not applied [19]. A counting sort operates by counting the occurrence of each to-be-sorted object and uses

arithmetic on those counts to determine the positions of each object in the output. The time and space complexities are both linear in the number of objects, $O(n + k)$ for the Pseudocode 3 [20].

Pseudocode 3. Counting sort used by radix sort.

```
#counting sort

''' allocate an array Count[0..k] ; initialize each array cell to zero ; THEN '''
FOR each input item x:
        Count[key(x)] = Count[key(x)] + 1
ENDFOR
total = 0
FOR i = 0, 1, ... k:
        c = Count[i]
        Count[i] = total
        total = total + c
ENDFOR

''' allocate an output array Output[0..n-1] ; THEN '''
FOR each input item x:
        store x in Output[Count[key(x)]]
         Count[key(x)] = Count[key(x)] + 1
ENDFOR
RETURN Output
```

The LSD radix sorting algorithm is easy to understand. Suppose we have an input of a set of n reads $R = \{S_1, S_2, S_3, \dots, S_n\}$ of strings of length k over the alphabet {A,T,G,C},

and outputs R in increasing alphabetical order. The algorithm will look at the last position $(k-1)$ in all the reads (the least significant digit or LSD) and counting sort all the reads based on letters at this position. Once sorted, it continues to count sort reads based on positions $(k-2), (k-3), \ldots, 0$. Upon completion of the iteration to position 0, the set of reads R is sorted alphabetically. (Figure 2) The time and space complexities are both $O(nk)$.

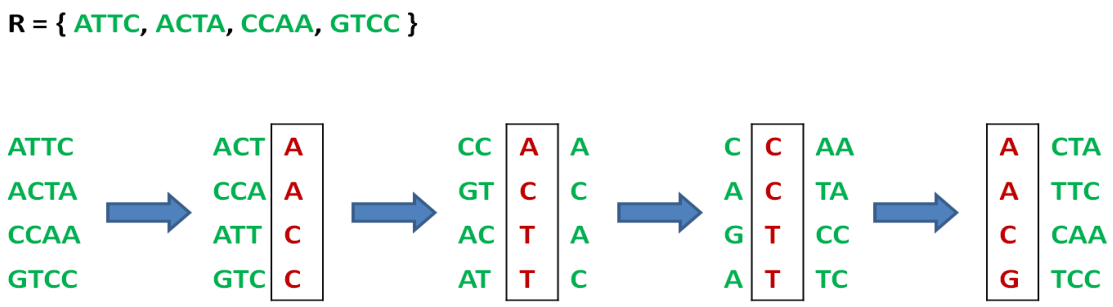


Figure 2. Example of LSD radix sort on 4nt-long sequences.

2.2.3.2 Radix Tree

Once the read sequences are sorted alphabetically, a radix tree can be built for rapid searching and copy number retrieval used in our mapping protocols.

2.2.3.2.1 The idea of radix tree

Radix tree is an optimized compact tree to store strings with a high occurrence of prefixes. Its edges are common prefixes from the parent node to a child node. For

instance “ACTA” and “ACTC” have same prefix “ACT”, which is the edge from root to one child node, seen in Figure 3,. The last letters “A” and “C” are different and become edges from that child node to two different grandchild nodes. The copy number will be stored in the node to easily retrieve the corresponding copy numbers of each path.

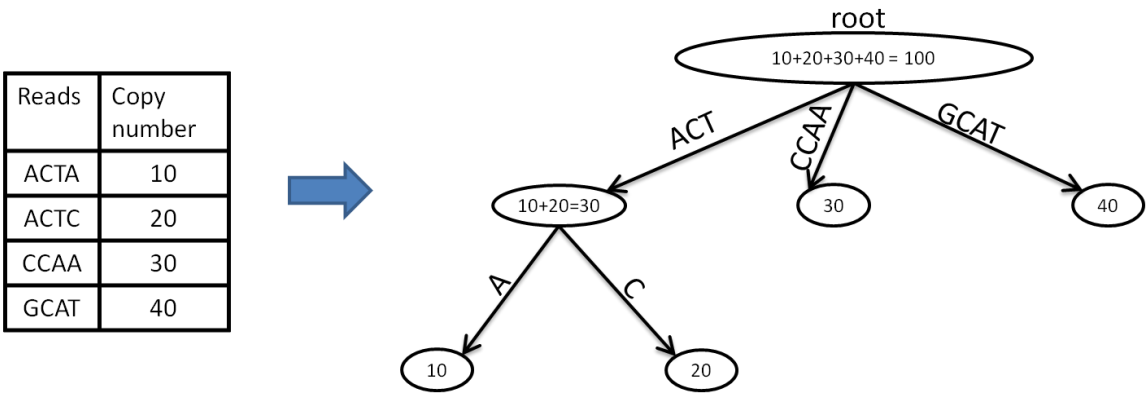


Figure 3. Schematic representation of a radix tree developed from four 4 nt-long sequences.

A radix tree is a data structure selected for quick searching in SCRUFFY because of the characteristics of the DNA sequences. There are only four letters in the sequences {A,T,C,G}, but the number of read sequences can be in millions. Therefore, there will be a very high occurrence of prefixes in the pool of reads. The compact feature of radix

tree is most applicable here to minimize the space complexity. Also, since the height of the tree is not larger than the length of the read sequences, the constant m, the worst traversal time (m), is also a constant for a lookup.

2.2.3.2.2 The node of radix tree

To understand the tree structure, each node is designed to have the following member variables.

- *Copy*: the sum of the copy numbers of all subsequences (or prefixes) till the node
- *Depth*: the position of the first letter in the edge in the context of read sequence
- *Motif*: a character array storing the prefix from the root to the node
- *First_Location*: the first occurrence of one particular prefix in the sorted array of read sequences
- *Last_Location*: one position after the last occurrence of the particular prefix referred to by the *First_Location*
- *Node_A*: pointer to a child node if the first letter of the edge is 'A'
- *Node_T*: pointer to a child node if the first letter of the edge is 'T'
- *Node_G*: pointer to a child node if the first letter of the edge is 'G'
- *Node_C*: pointer to a child node if the first letter of the edge is 'C'

2.2.3.2.3 Creation of radix tree from sorted reads

The tree is created by recursively calling the constructor of the node, *Radix_Tree_Node* (Pseudocode 4 and Figure 4). The initial call will pass the object of *_Sorted_Reads*, *_First_Location* = 0, *_Last_Location* = total number of reads in the object

_Sorted_Reads, and *_ParentNode_Depth* = 0. In this way, the tree will be created from the root to the leaf node. When all the letters from *_First_Location* to *_Last_Location* are the same letter, the code will continue to check if the letters in the next position are the same. When different letters are found, new child nodes are created corresponding to each letter by recursive calls.

Pseudocode 4. Implementation of Radix_Tree_Node constructor, showing the creation of the whole tree by recursively calling the constructor.

```
# constructor of the Node
'''member variables are indicated in the previous section'''
'''assume object _Sorted_Reads has the following functions:
        get_read_length()
        get_copy_number(index)
        get_read_sequence(index)
'''
create FUNCTION Radix_Tree_Node (_Sorted_Reads, _First_Location, _Last_Location,
                                 _ParentNode_Depth)
    First_Location = _First_Location
    Last_Location = _Last_Location
    initialize all other member variables to 0 or NULL
    Motif_Length = _Sorted_Reads->get_read_length()
    allocate Motif based on the Motif_Length
    //exiting condition: only one sequence left
    IF _Last_Location - _First_Location == 1
        assign whole sequence to Motif
        assign corresponding copy number to Copy
        assign Motif_Length to Depth
        RETURN
    ENDIF
    //not exiting:
    //count the sum of the copy number till the current Node
    FOR i in First_Location to Last_Location
```

```
            Copy += _Sorted_Reads->get_copy_number(i)
        ENDFOR
        //assign characters in the parent Node to Motif of current Node
        FOR i in 1 to _ParentNode_Depth
            Motif[i] = _Sorted_Reads->get_read_sequence(_First_Location)[i]
        ENDFOR
        //count the number of reads having A/T/G/C at current position
        FOR k in _ParentNode_Depth to Motif_Length
            Copy_A = 0
            Copy_T = 0
            Copy_G = 0
            Copy_C = 0
            FOR j in First_Location to Last_Location
                SWITCH _Sorted_Reads->get_read_sequence(j)[k]
                    CASE 'A' or 'a': Copy_A++
                    CASE 'T' or 't': Copy_T++
                    CASE 'G' or 'g': Copy_G++
                    CASE 'C' or 'c': Copy_C++
                    DEFAULT: RETURN
                ENDSWITCH
            ENDFOR
            IF all reads have same letter (A/T/G/C) at the current position
                assign this letter to Motif[k]
                CONTINUE
            ENDIF
            BREAK
        ENDFOR
        Depth = k;
        IF Depth >= Motif_Length
            RETURN
        ENDIF

        IF the count of any of the letters is greater than 0  (i.e. Copy_A >0 )
            //recursive call to create new Radix_Tree_Node for this letter
        Node_A= new Radix_Tree_Node (_Sorted_Reads, First_Location,
                                        First_Location + Copy_A, Depth+1)
        ENDIF
        RETURN
ENDFUNCTION
```

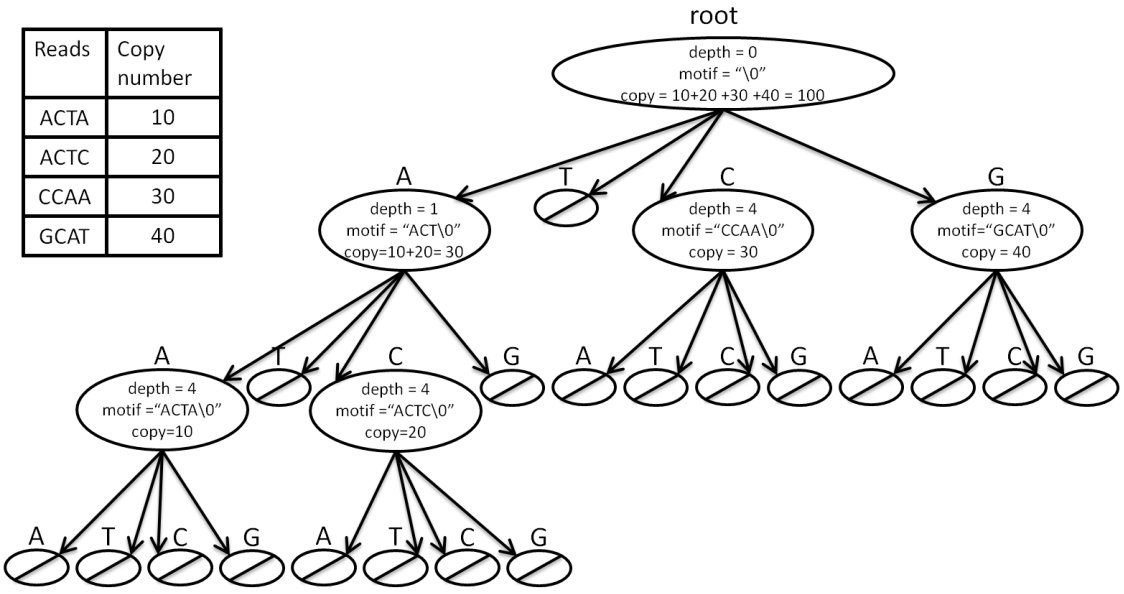


Figure 4. Representation of a radix tree that has been implemented in SCRUFFY. The tree is developed from the same example of four 4nt-long sequences.

2.2.3.3 Operations

Since the radix tree serves as a data structure for string searching, there is no need for insertion and deletion operations once the tree is built. The only operation needed is lookup. To lookup a particular subsequence in the reference genome, one should follow the path defined by the subsequence and also compare each letter to the Motif of the Nodes in the path. The comparison of each letter in a string will add some overhead to the lookup operation.

2.2.4 Performance analysis

Both hash and radix tree data structures enable fast searching for exact matches in mapping. There are two mapping programs developed in our lab. One is Slim, which uses hash for exact mapping. The other is Scruffy, which uses a radix tree for exact mapping. To evaluate their performance, both mapping programs were run with same sets of reads, where each set contained different number of reads ranging from 1,000 (1k) to 100,000,000 (100mil). All data were collected on the same machine using one processor. Each data point was collected three times and the mean was plotted with the standard deviation as shown with the associated error bars.

Memory usage by both programs includes the same inputs (the object of read sequences and the reference database) and the same outputs (coverage, reads used in mapping, and report). The time consumed for I/O and scanning through the reference database are also similar between these two programs. If a difference between performances is seen, it would arise from the different data structure used for the mapping process (Table 2).

Table 2. Factors affecting performances of each data structure. (m is the length of the reads)

Operations	Hash	Radix Tree
Create Data Structure	Signature conversion Collision resolution (linear probing)	Radix Sort of reads Recursive calls
Lookup	$O(\frac{m}{4})$ + collision resolution	$O(m)$

The oncoviral database is the smallest database in the lab, and consists of about seven thousand viral genomic sequences. As seen in the Figure 5, the performances of two programs are similar below 10 million reads. When the number of reads increased to 100 million, Slim out performed Scruffy in both time and space complexities. This is because when the reference database is small, there is less occurrence of collision during the lookup. The time for lookup in hash is closed to $O\left(\frac{m}{4}\right)$ due to the signature conversion. However, the lookup in the radix tree takes a bit more time, $O(m)$, and the recursive call to create the tree also adds to the overhead in both time and memory consumption.

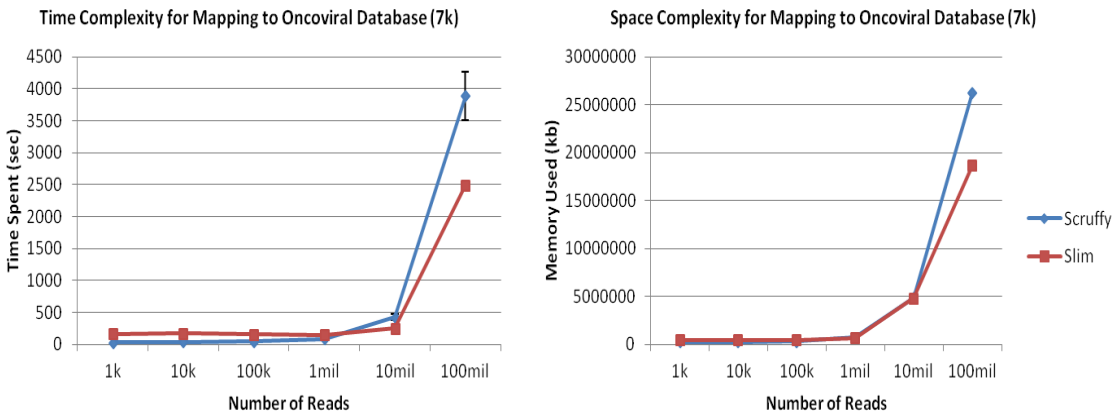


Figure 5. Comparison of time and space complexities between Scruffy (radix tree) and Slim (hash) programs for mapping to Oncoviral Database with exact matches.

3 Analyzing cancer data using Slim mapping program

Based on the better performance of hash, the Slim program was used in the subsequent data analysis. Data from three types of cancer were obtained for analysis. The goal of the analysis was to compare oncovirus genomic sequences present in cancerous tissue to those in normal tissues in several patients. Therefore, the following steps were performed to determine the differences in the oncovirus profile between cancer and normal tissue from the mapping results.

To assess the role of oncovirus in cancer, one can ask questions such as:

1) How different (similar) is the profile of oncovirus in cancerous tissue compared to that in normal tissue?

2) Are there any cancer-specific or normal tissue-specific oncoviruses in each patient?

3) What are the common oncovirus species in both cancer and normal tissue from each patient?

4) Does the copy number of oncovirus change in cancerous tissue compared to that of normal tissue?

The answers to these questions will shed light on the potential role of oncoviruses in cancer development.

3.1 Similarity in oncovirus profile of cancer vs. normal

To reveal the similarity between two sets of oncovirus species, the Jaccard similarity coefficients of each pair of samples (cancer vs. matched normal) were calculated by dividing the size of the intersection to the size of the union of the sample sets:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}.$$

The result is shown in Table 3. As we can see, all the pairs have at least 50% similarity. These data suggest that at least 50% of the oncovirus species are shared between cancer and matched normal tissue. This is not a surprise because each tissues pairs arise from the same patient. Each patient is expected to be susceptible to a subset of oncoviruses. It'll be interesting to see what kinds of oncoviruses are present in all individuals. This point will be discussed in the last section of results.

Secondly, the degree of similarity seen in different types of cancer differs. For the patients with brain tumor, the average similarity is 57%. This is quite different from the average similarity of 78% for patients with colon cancer and 92% for patients with cervical cancer. This difference could be due to the source of genomic DNA. In patients with brain tumor, blood DNA was used as the normal counterparts while in patients with colon and cervical cancer, DNA from normal solid tissues was used instead of blood derived DNA. Comparisons between same solid tissue types are expected to be more similar than comparisons between solid and liquid tissues.

Last, even in the three healthy persons, many oncoviruses are mapped and share more than 50% oncoviruses (10 out of less than 20 oncovirus species in each person are shared

among all three persons). This can be explained by the long latency of oncovirus infection. More information will be needed to assess this observation, such as age, family history for cancer, etc.

Table 3. Number of mapped oncoviruses and similarity between the set of cancer and the set of matched normal samples.

Sample Type	Person ID	Number of Oncoviruses Mapped			Similarity (%)
		Cancer	Normal	Common	
Brain	13	20	13	12	57%
	14	12	14	10	63%
	51	20	10	10	50%
Colon	24466	9	12	9	75%
	26807	15	14	13	81%
Cervical	19210_1	36	36	35	95%
	19210_2	35	35	33	89%
Healthy person	72	N/A	19		
	74	N/A	16	10*	
	75	N/A	15		

* common among the three healthy person.

3.2 Tissue-specific oncovirus species

Next I determined which are the oncovirus species can be found in cancer but not in the matched normal tissue samples and vice versa. If oncoviruses are the sole cause of cancer, more oncoviruses would be expected in cancerous tissue than in normal tissue. However, the result (Table 4) is different from the expectation. The result varied among the different types of cancer. In glioma patients, as expected, more oncovirus species are observed in cancerous tissue than in normal tissue. While in patients with colon or cervical cancer, this is not the case. Some patients showed more, while the majority of them have less viral species in the cancerous tissue than in normal tissue. This can be

due to the different sources of normal tissue, such as blood in the case of glioma or normal solid tissues of colon and cervical sample.

Table 4. Oncovirus species found only in cancer compared to those in the normal counterpart in cancer patients.

Cancer Type	Patient ID	Cancer-only oncoviruses	Normal tissue-only oncoviruses
Glioma	13	Human herpesvirus 3 DNA, Human herpesvirus 3 DNA, Human herpesvirus 3 strain Human herpesvirus 3 strain Human herpesvirus 3 strain 22 Human herpesvirus 3 strain 8 Human papillomavirus type 6 , Recombinant Hepatitis C virus J6 (5'UTR(Cell-U6)-NS2)/JFH1	HIV-1 isolate SF33 from
	14	Human herpesvirus 3 strain Human herpesvirus 3 strain 22	Human papillomavirus type Recombinant Hepatitis C virus J6(5'UTR(Cell-U3)-NS2)/JFH1 Recombinant Hepatitis C virus J6(5'UTR(Cell-U6)-NS2)/JFH1
	51	Human papillomavirus type 71 Hepatitis C virus subtype 1b Hepatitis C virus subtype 1a Human herpesvirus 1 transgenic Human herpesvirus 5 transgenic Recombinant Hepatitis C virus J6(5'UTR(Cell-U6)-NS2)/JFH1	
Colon	24466		Human papillomavirus type Human herpesvirus 3 DNA, Hepatitis C virus subtype 1b from Hubei
	26807	Rous sarcoma virus - Schmidt-Rous sarcoma virus strain Schmidt-Ruppin B	

Cancer Type	Patient ID	Cancer-only oncoviruses	Normal tissue-only oncoviruses
Cervical	19210_1	Hepatitis C virus subtype 1b from Hubei	Human papillomavirus type 20 Recombinant Hepatitis C virus J6(5'UTR(Cell-U3WTS1)-NS2)/JFH1
	19210_2	Human papillomavirus type 20 HIV-1 isolate SF33 from USA	Hepatitis C virus subtype 1b Recombinant Hepatitis C virus J6(5'UTR(Cell-U3)-NS2)/JFH1 Recombinant Hepatitis C virus J6(5'UTR(Cell-U3WTS1)-NS2)/JFH1

In glioma patients, multiple herpesviruses are commonly found in cancer in all three individuals but with different viral types and strains. This data suggests that the herpesvirus can be a candidate cause to glioma. In patients with colon cancer, the Rous Sarcoma virus was found in cancer from one individual but not in the other. This suggests that it may be causative for a subtype of colon cancer or it is merely a secondary infection after cancer is initialized rather than the primary cause to colon cancer. In patients with cervical cancer, the result seems less clear. Viruses that are cancer-specific in one patient are present only in normal tissue in another patient. This may due to contaminated normal samples by cancer cells during biopsy or DNA preparations. More data from different individuals will be needed to clarify this result.

3.3 Oncoviruses that are common in cancer and normal tissue show different levels of coverage

I then took a look at the viruses shared between cancerous and normal tissue of patients. I compared the average coverage of oncovirus in cancer to that in normal tissue. According to the hypothesis, I expected that oncoviruses would have higher coverage in cancer than that in normal. Results show the coverage differences are obvious (Figure 6) but not always high coverage in cancerous tissue. It was very interesting to see that three groups of patients have three different profiles.

In the group of glioma patients (Figure 6, Glioma_13_Common, Glioma_14_Common, and Glioma_51_Common), almost all species of oncoviruses are higher covered in cancer than in normal. This indicates glioma may be associated with a wide spectrum of oncoviruses. In the group of patients with cervical cancer (Figure 6, Cervical_19210_1_Common, and Cervical_19210_2_Common), only papillomavirus consistently showed significantly higher coverage in cancer. It suggests cervical cancer is related to one or more typical oncovirus, such as papillomavirus. In the group of patients with colon cancer (Figure 6, Colon_24466_Common, and Colon_26807_Common), however, almost all the oncoviruses are covered less in cancer than in normal. This was totally unexpected. If there are no errors in the experiments, such as mixing cancer and normal tissues, it implies that colon cancer may not be a virus-causing cancer type.

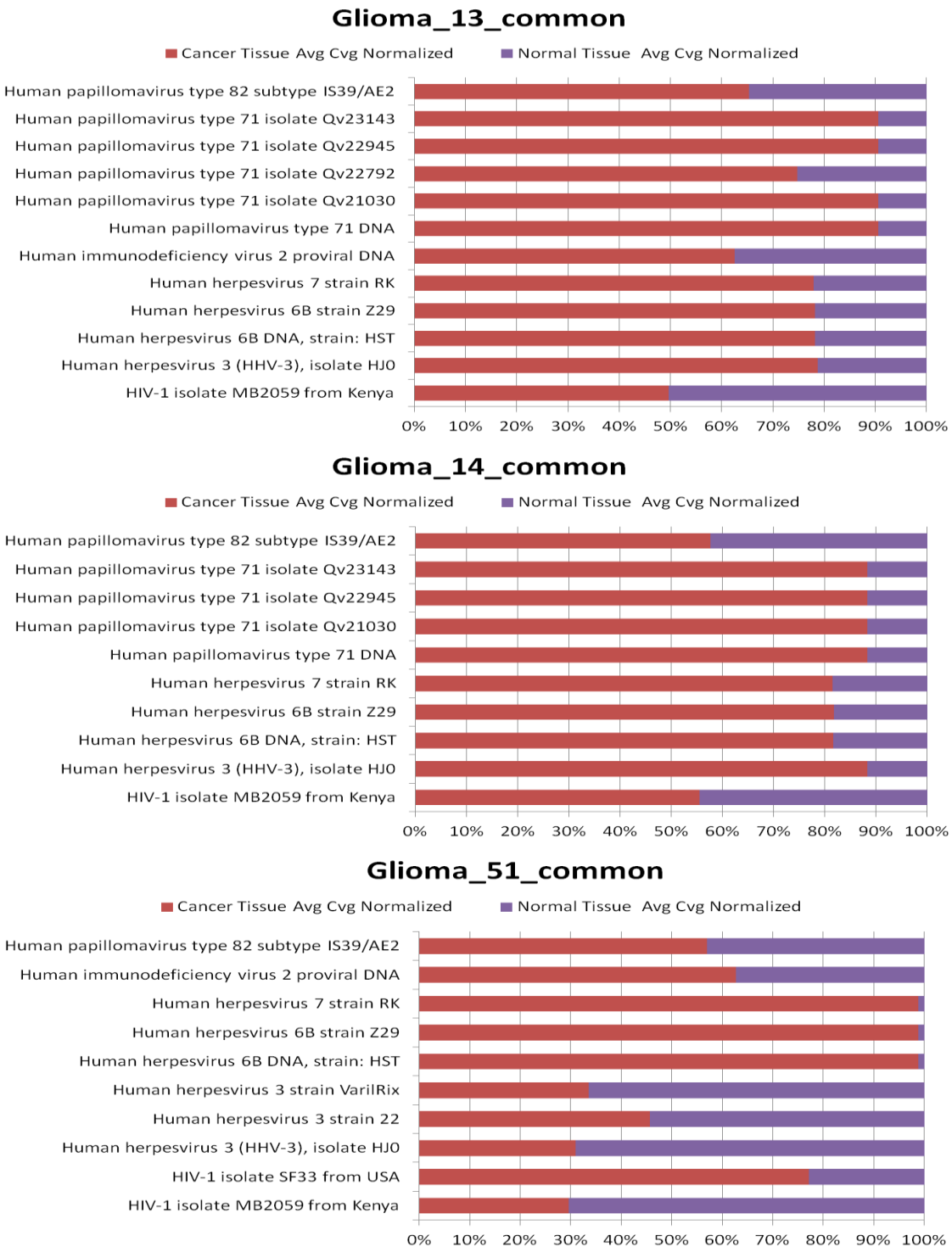
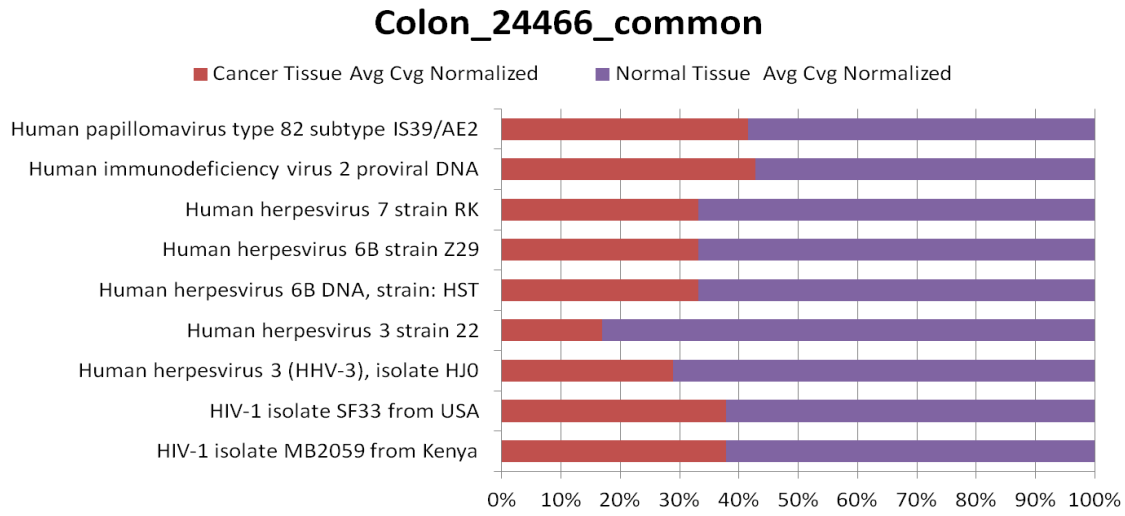


Figure 6. Oncoviruses that are common in cancer and normal tissue show different levels of coverage.



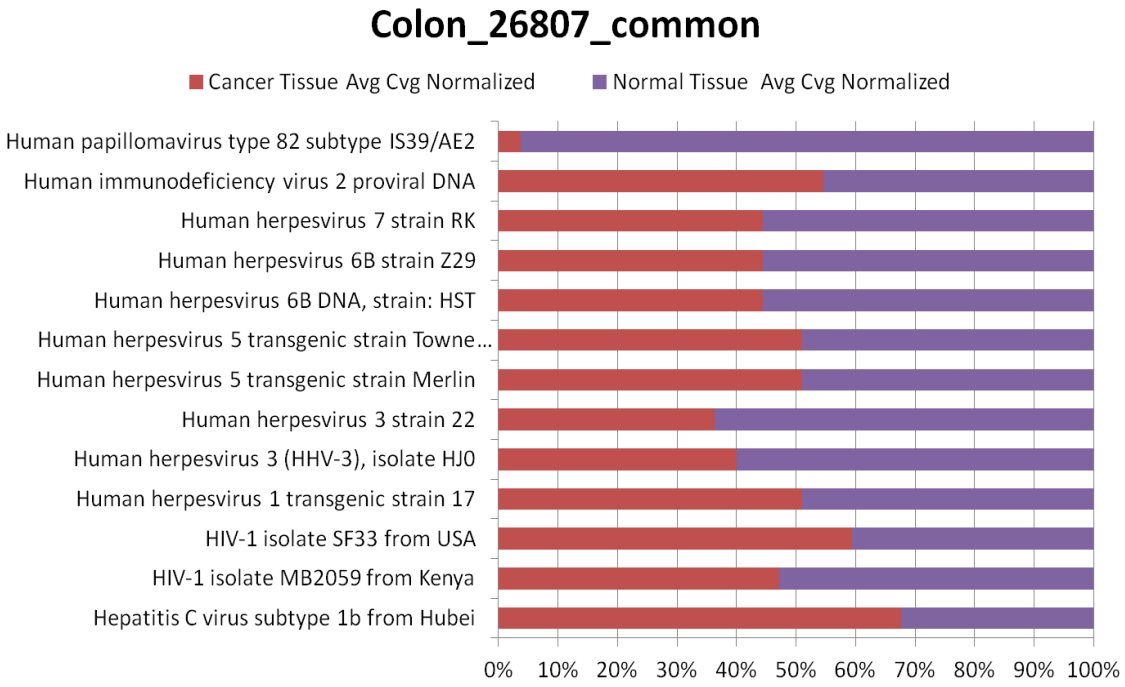


Figure 6. (cont)

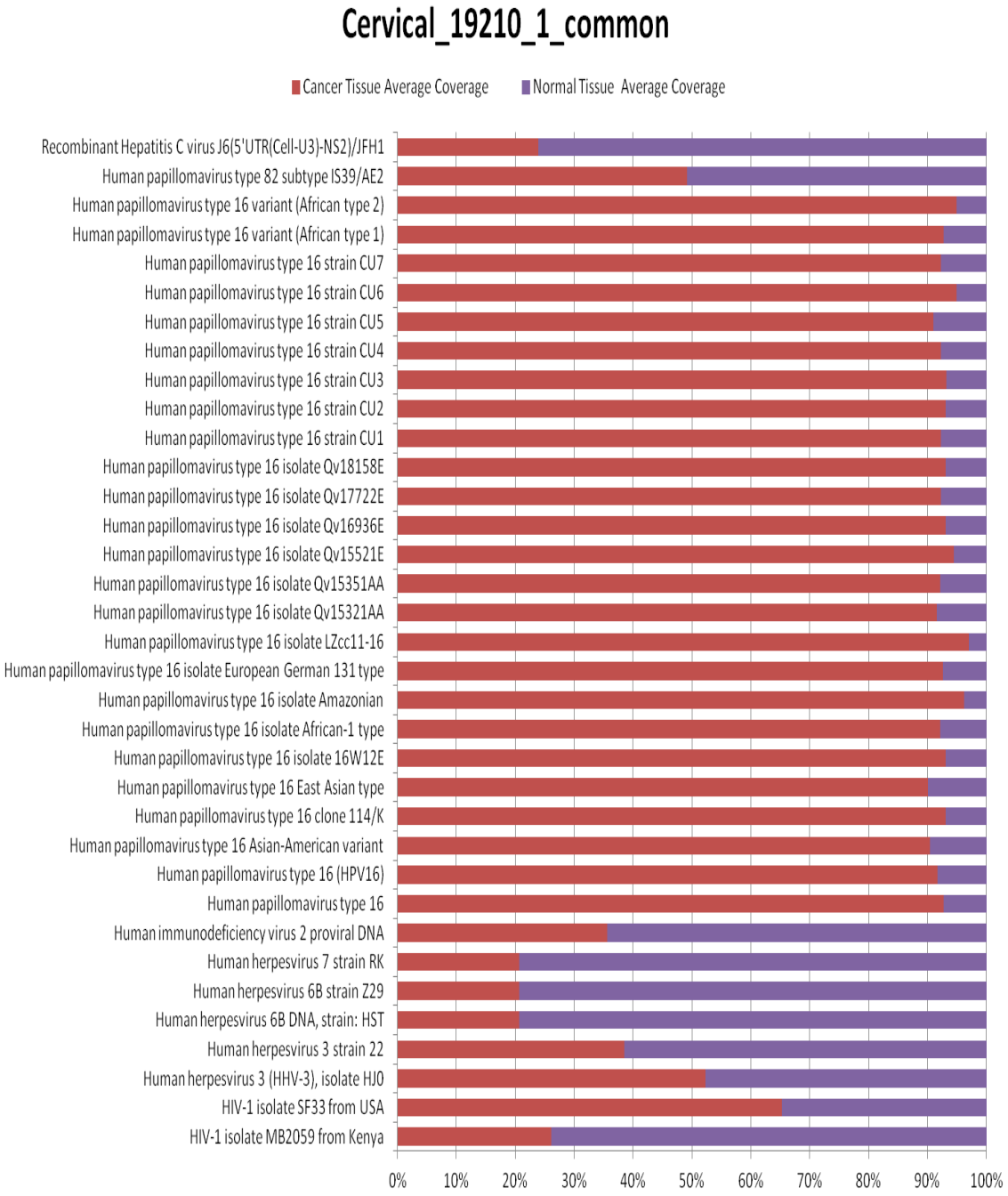


Figure 6. (cont)

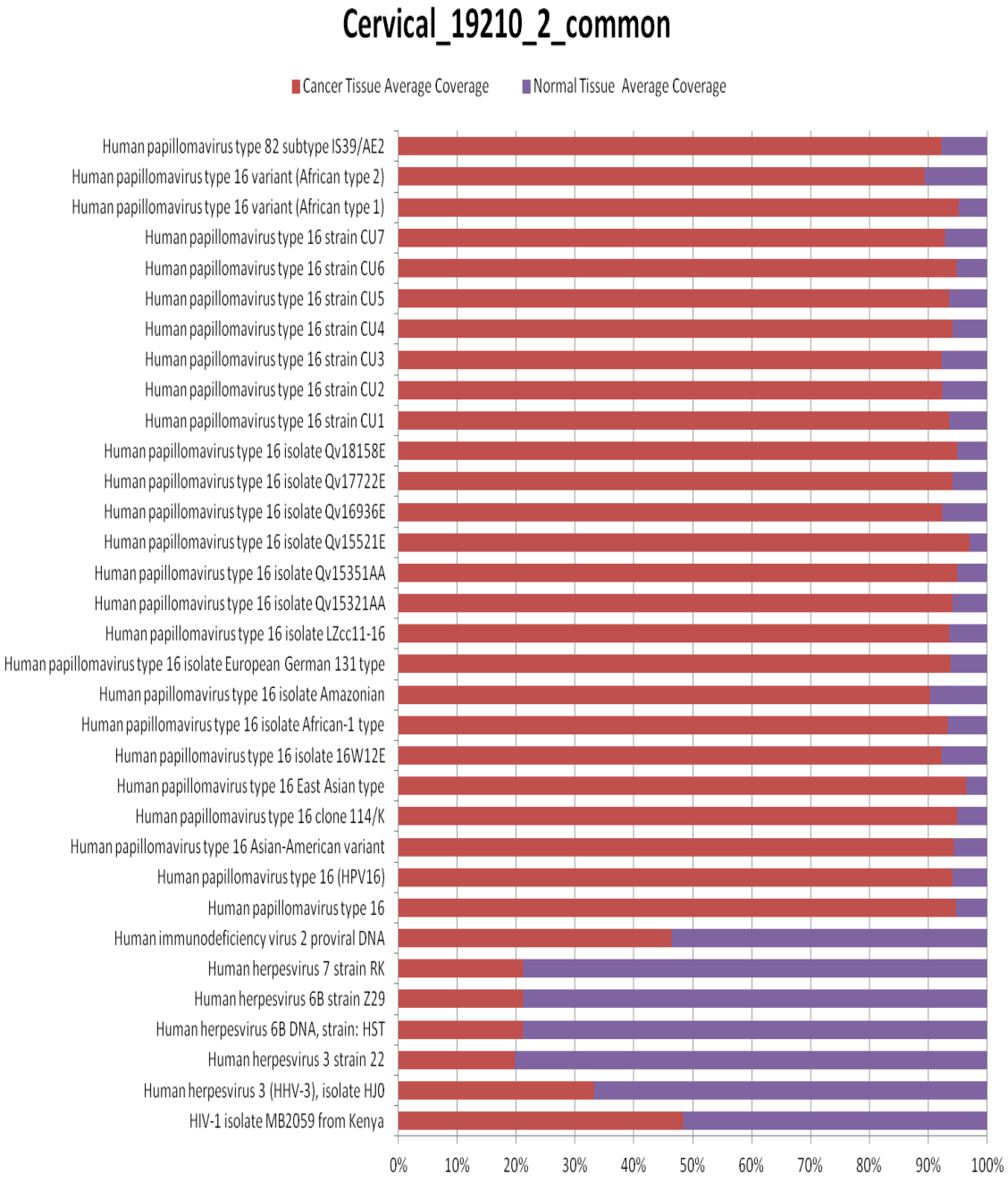


Figure 6. (cont)

3.4 Five common oncoviruses are found in all examined individuals

There are five common species of oncoviruses present in both healthy and cancer-affected individuals. Four of them are different strains or types of herpesvirus and one is papillomavirus type 82. As we can see in Figure 7, the there is low coverage of these oncoviruses in healthy individuals. A slight increase in coverage of all cancer samples is observed. The most striking fact was the dramatically elevated level of coverage in normal tissues from cervical and colon cancer patients.

Another interesting observation was the spectrum of these five oncoviruses are consistent among all samples with herpesvirus 6B Z29 (green) and papillomavirus (purple) the most dominant and HHV-3 the least (light blue).

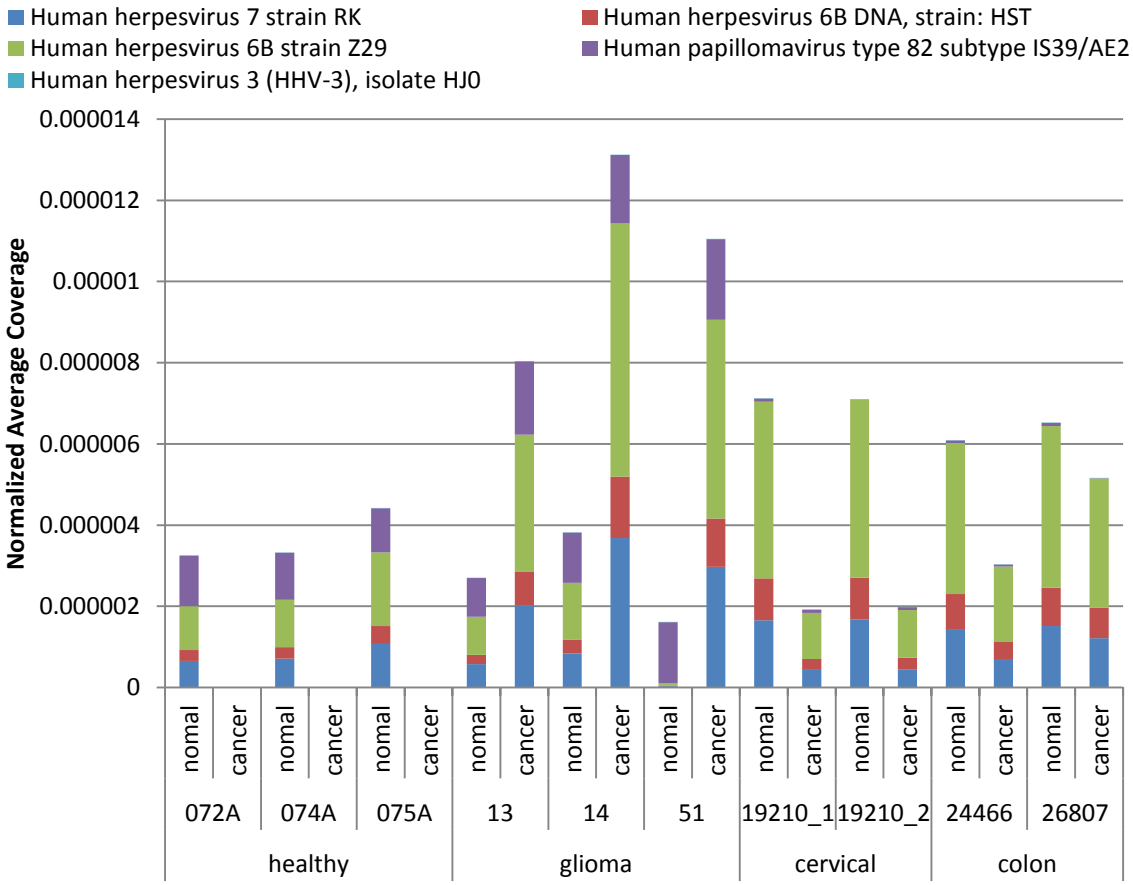


Figure 7. Average coverage of common oncovirus species in all individual examined. Five oncoviruses are shared among all individual and the average coverage of each of them are shown as stack with five different colors in every sample. Color representations can be found on the top of graph. Source data are present in Table8.

3.5 Summary and conclusion

This was a pilot work towards a more comprehensive and sophisticated project with more data and a larger viral database. There are certain limitations to the result of this work. Due to the limited overall nucleotide coverage (less than 1) of the sequencing reads, the regions in the samples that are not sequenced will result in false negative. On the other hand, virus genomes contain sequences that are highly homologous to human genomic sequence, which will lead to false positive. Improvements can be made by increasing the overall coverage of sequencing of each sample, such as deep sequencing, and also by eliminating human sequences from virus genome or examine expression of virus instead of genomic material.

In spite of all the limitations in the data, this work provides a systematic work flow from raw sequencing data to a meaningful biological outcome. Sequencing data were processed by following the routine in the lab. Mapping results were interpreted in four steps regarding finding the difference between the cancerous and normal samples. First, I took a look at the overall similarity of cancer result compared to the normal and found that the oncovirus species were of great similarity between them. Next, I listed all the cancer-only and normal-only virus species and compared among three groups of patients. I then examined the viruses shared by cancer and normal and compared their coverage. Last, I found the five oncoviruses shared by all individuals including the healthy and cancer-affected and compared the coverage of them.

The preliminary result shows that the profiles of oncoviruses are different depended on cancer types. Glioma is more likely to be related to multiple oncoviruses. Cervical

cancer is only associated with a certain type of oncoviruse. Colon cancer may not be solely caused by oncoviruses. More data and further experiments are needed to make a solid conclusion.

4 References

1. zur Hausen, H., *The search for infectious causes of human cancers: where and why (Nobel lecture).* Angew Chem Int Ed Engl, 2009. **48**(32): p. 5798-808.
2. Prober, J.M., et al., *A system for rapid DNA sequencing with fluorescent chain-terminating dideoxynucleotides.* Science, 1987. **238**(4825): p. 336-41.
3. Hutchison, C.A., 3rd, *DNA sequencing: bench to bedside and beyond.* Nucleic Acids Res, 2007. **35**(18): p. 6227-37.
4. Metzker, M.L., *Emerging technologies in DNA sequencing.* Genome Res, 2005. **15**(12): p. 1767-76.
5. Feng, H., et al., *Human transcriptome subtraction by using short sequence tags to search for tumor viruses in conjunctival carcinoma.* J Virol, 2007. **81**(20): p. 11332-40.
6. Smith, A.D., Z. Xuan, and M.Q. Zhang, *Using quality scores and longer reads improves accuracy of Solexa read mapping.* BMC Bioinformatics, 2008. **9**: p. 128.
7. Li, H., J. Ruan, and R. Durbin, *Mapping short DNA sequencing reads and calling variants using mapping quality scores.* Genome Res, 2008. **18**(11): p. 1851-8.

8. Lin, H., et al., *ZOOM! Zillions of oligos mapped.* Bioinformatics, 2008. **24**(21): p. 2431-7.

9. Jiang, H. and W.H. Wong, *SeqMap: mapping massive amount of oligonucleotides to the genome.* Bioinformatics, 2008. **24**(20): p. 2395-6.

10. Schatz, M.C., *CloudBurst: highly sensitive read mapping with MapReduce.* Bioinformatics, 2009. **25**(11): p. 1363-9.

11. Li, R., et al., *SOAP: short oligonucleotide alignment program.* Bioinformatics, 2008. **24**(5): p. 713-4.

12. Campagna, D., et al., *PASS: a program to align short sequences.* Bioinformatics, 2009. **25**(7): p. 967-8.

13. Eaves, H.L. and Y. Gao, *MOM: maximum oligonucleotide mapping.* Bioinformatics, 2009. **25**(7): p. 969-970.

14. Kim, Y.J., et al., *ProbeMatch: rapid alignment of oligonucleotides to genome allowing both gaps and mismatches.* Bioinformatics, 2009. **25**(11): p. 1424-1425.

15. Burrows, M., Wheeler, D.J., *A block-sorting lossless data compression algorithm.* Technical Report, 124 Palo Alto, CA: Digital Equipment Corporation; 1994.

16. *http://soap.genomics.org.cn/*. Accessed on 6/19/2012.

17. Langmead, B., et al., *Ultrafast and memory-efficient alignment of short DNA sequences to the human genome.* Genome Biol, 2009. **10**(3): p. R25.

18. Li, H. and R. Durbin, *Fast and accurate short read alignment with Burrows-Wheeler transform.* Bioinformatics, 2009. **25**(14): p. 1754-60.

19. Cormen, T.H. and Books24x7 Inc., *Introduction to algorithms, second edition.* 2001, MIT Press: Cambridge, Mass.

20. *http://en.wikipedia.org/wiki/Counting_sort#The_algorithm*. Accessed on 6/19/2012.