A Study of Content-Dependent Chunking Algorithms

THESIS

Submitted in Partial Fulfillment of

the Requirements for

the Degree of

MASTER OF SCIENCE (Computer Science)

at the

NEW YORK UNIVERSITY
TANDON SCHOOL OF ENGINEERING

by

Shahzaib Javed

January 2017

A Study of Content-Dependent Chunking Algorithms

THESIS

Submitted in Partial Fulfillment of

the Requirements for

the Degree of

MASTER OF SCIENCE (Computer Science)

at the

NEW YORK UNIVERSITY
TANDON SCHOOL OF ENGINEERING

by

Shahzaib Javed

January 2017

Approved:

Advisor Signature

Date

Department Chair Signature

Date

University ID: **N14482418**
Net ID: **Sj1246**

VITA

Shahzaib Javed was born in Pakistan on September 1 1993. He attended New York University Tandon School of Engineering, where he pursued a dual degree in Electrical Engineering, and Computer Science. While he was a student at NYU, Shahzaib was a part-time teaching assistant for Data Structures and Algorithms. He also participated in summer research with Professor Haldun Hadimioglu, where he programmed FPGA chips using C programming. Shahzaib devoted one year to his thesis, and would like to thank Professor Suel Torsten for allowing him to conduct this research. He would also like to thank the Computer Science and Engineering department, for providing private space to work on his thesis.

ABSTRACT

A Study of Content-Dependent Chunking Algorithms

by

Shahzaib Javed

Advisor: Prof. Suel Torsten, Ph.D.,

Submitted in Partial Fulfillment of the Requirements for

the Degree of Master of Science (Computer Science)

January 2017

There are many cases in which a file is created with only minimal modifications from the previous version. This may occur in versioned document sets such as Wikipedia, where a newer version is created by inserting or deleting a paragraph, fixing spelling issues, or even simply correcting a grammar error. Instead of storing the newer version of the file in the entirety, it would be less expensive to store only the pieces of the file that are missing. In this paper, we examine and explain the current algorithms that are used to detect document similarity between files. The algorithms we examine in this paper are Karp-Rabin, Winnowing, TDDD, and 2Min. We run the algorithms on various datasets, such as the Internet Archive, gcc and emacs source files, and randomly generated files, to determine which algorithm finds the most document similarity. We run timing experiments to determine the speed of each of the algorithms. We also present two new algorithms, by making modifications of the 2Min algorithm, which outperform the original in finding more document similarity.

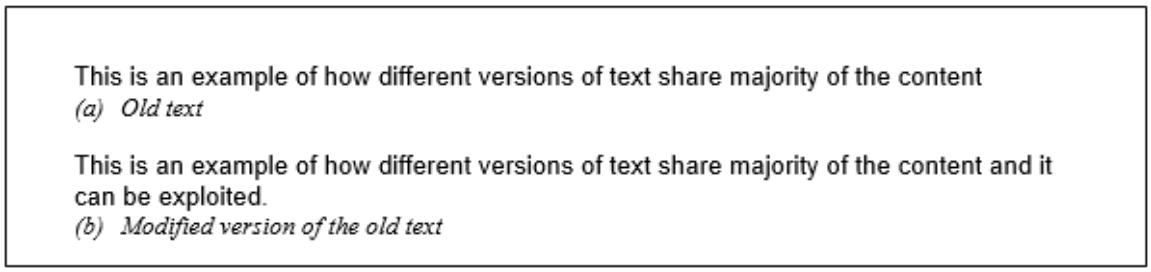
Table of Contents

List of Figures

1. Introduction

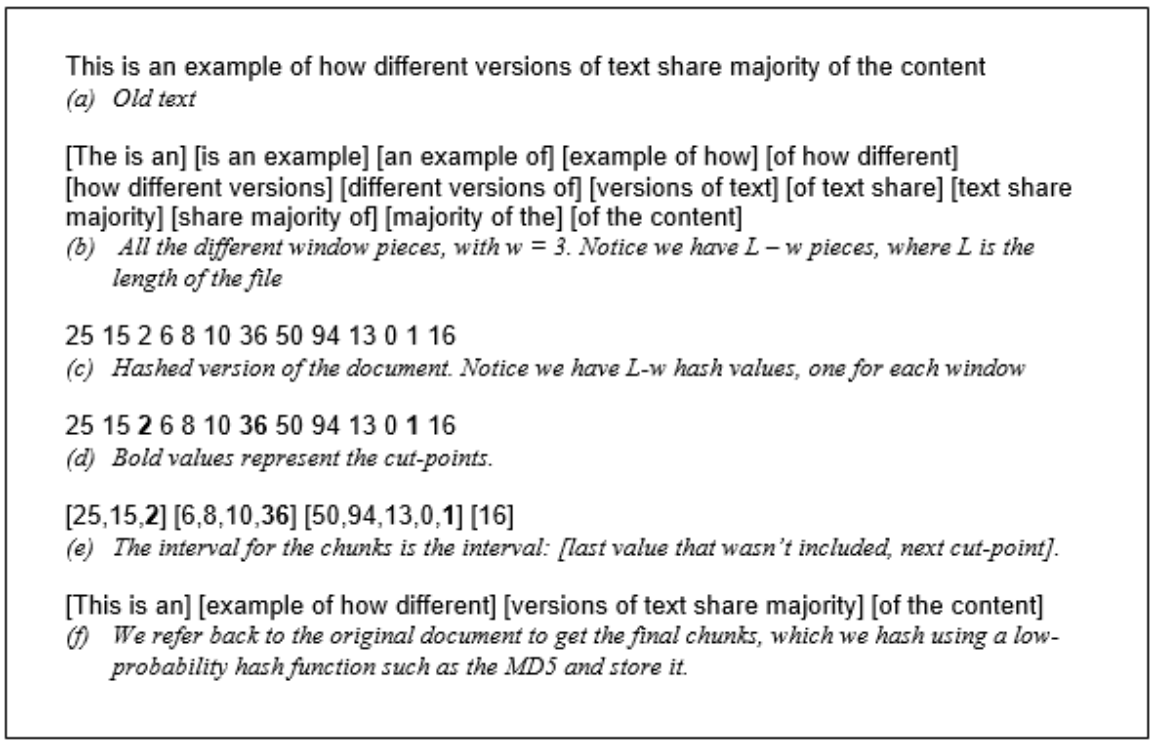
There are many cases in which a file is created with only minimal modifications on the previous version. The majority of the file content, however, remains the same. An example of this are versioned document sets. In versioned documents, such as Wikipedia, a newer version is created by inserting or deleting a paragraph, fixing spelling issues, or even simply correcting a grammar error. There may be a couple of different versions which differ from the previous version only in certain places. It would be expensive to store all these versions in the entirety, especially for large files. Figure 1 (a) and Figure 1 (b) show two versions of a document that has been modified only in certain places, and thus share the majority of the text. Rather than storing both versions individually, we can exploit the redundancy and save on storage space by storing the previous version and only the differences in the newer one.

We can also save on bandwidth costs when sending a file from a server to a client by only sending the differences between the files, provided the client has a similar file. One example is distribution of a software package. Suppose a client has a previous version of the software package and wants to download the newer version. Rather than downloading the entire newer version, it would be faster to download just the pieces of the software that are missing. Thus, the server will only send the pieces of the files that are missing from the clients' version (assuming the server knows which version is held by the client). The client can then reconstruct the newer version from its older version using the missing pieces.


This is an example of how different versions of text share majority of the content
(a) Old text

This is an example of how different versions of text share majority of the content and it can be exploited.
(b) Modified version of the old text

Figure 1: Example of versioned text


This is an example of how different versions of text share majority of the content
(a) Old text

[The is an] [is an example] [an example of] [example of how] [of how different] [how different versions] [different versions of] [versions of text] [of text share] [text share majority] [share majority of] [majority of the] [of the content]
(b) All the different window pieces, with w = 3. Notice we have L – w pieces, where L is the length of the file

25 15 2 6 8 10 36 50 94 13 0 1 16
(c) Hashed version of the document. Notice we have L-w hash values, one for each window

25 15 **2** 6 8 10 **36** 50 94 13 0 **1** 16
(d) Bold values represent the cut-points.

[25,15,**2**] [6,8,10,**36**] [50,94,13,0,**1**] [16]
(e) The interval for the chunks is the interval: [last value that wasn't included, next cut-point].

[This is an] [example of how different] [versions of text share majority] [of the content]
(f) We refer back to the original document to get the final chunks, which we hash using a low-probability hash function such as the MD5 and store it.

Figure 2: Example of how the document is chunked

One application in which Karp-Rabin, a chunking algorithm, is used to save network bandwidth is in the low-bandwidth network file system (LBFS) [11]. LBFS is a file system that utilizes redundancies between different versions of a file when transferring the file between two machines. Rather than sending the entire new file, LBFS only sends the parts that are missing on either the recipients' side. LBFS accomplishes

this by having the sender compute and transmit fingerprints [14] of its version of the document, which the recipient then uses to determine and request the chunks or portions of the file that are missing from its own version. The fingerprints are computed using the Karp-Rabin algorithm [6]. In a nutshell, LBSF is a protocol that uses the similarities between documents to save on network cost.

Now the question arises: how do we best determine which parts of the content is similar? One commonly used approach is to partition the document into chunks using content-dependent chunking algorithms. The files are first hashed, and then these content- dependent chunking algorithms are run on the hashed versions of the document to find cut-points, which are then used to split the document into chunks. Cut-points are hash values that satisfy a predefined condition. It is desirable to output cut-points such that after slight file modifications, the majority of the chunks will still be the same between the different document versions. Figure 2 shows the process of using the cut-points to slice the document into chunks.

Figure 2 (a) - (c) shows the document being hashed. We first define a window size *w*, which we use to slide through the document and hash the content of each window. After the whole document is hashed, we run a content-dependent algorithm to find the cut-points, which is used to determine the final document chunks, as shown in Figure 2 (d) – (e). After we have located the cut-points, we refer back to the original file in order to chop the document into chunks, as shown in Figure 2 (f).

The hashed versions of the chunks, which are hashed using a low-probability collision function - such as the MD5 - are stored and used to determine which of the chunks are missing from a newer version. When a newer version of the document arrives,

we run the same algorithm and check whether the newer chunks exist, by comparing the hashes of the chunks and only sending or storing those that are missing.

These algorithms are content-dependent in the sense that the cut-points are determined based only on the file content around the cut-points. Thus, the algorithm will output the same chunk cut-points in sections where the content has not been modified; this is desirable for reducing file redundancy. There is also an alternative approach in which the document is partitioned into equal pre-defined chunk lengths, known as content-independent or fixed-size chunking. Fixed-sized chunking suffers from the boundary shifting or misalignment problem. Suppose there are two versions of a file, version 1 and version 2. Version 2 is created simply by inserting a single character into the middle of the document. If the file is partitioned using fixed-sized chunks, all the chunks after the inserted byte will be shifted over by one, which we refer to as the boundary shifting problem. Figure 3 portrays an example of the boundary shifting problem for the content-independent approach. In Figure 3 (a), we define a sample text and in Figure 3 (b), we partition the sample text into fixed-size chunks. In Figure 3 (c), the text is modified by inserting one word at the beginning and as a result, the document, shown in Figure 3 (d), is misaligned and divided up into different chunks. As a result, we will not find any redundancy between the versions, despite the majority of the content being repetitive. Therefore, content-dependent algorithms are preferred and will be the primary focus of this thesis. All of the examples were done using strings for clarity purposes. In practice, the file is read as bytes and each individual byte is looked at, as opposed to each string.

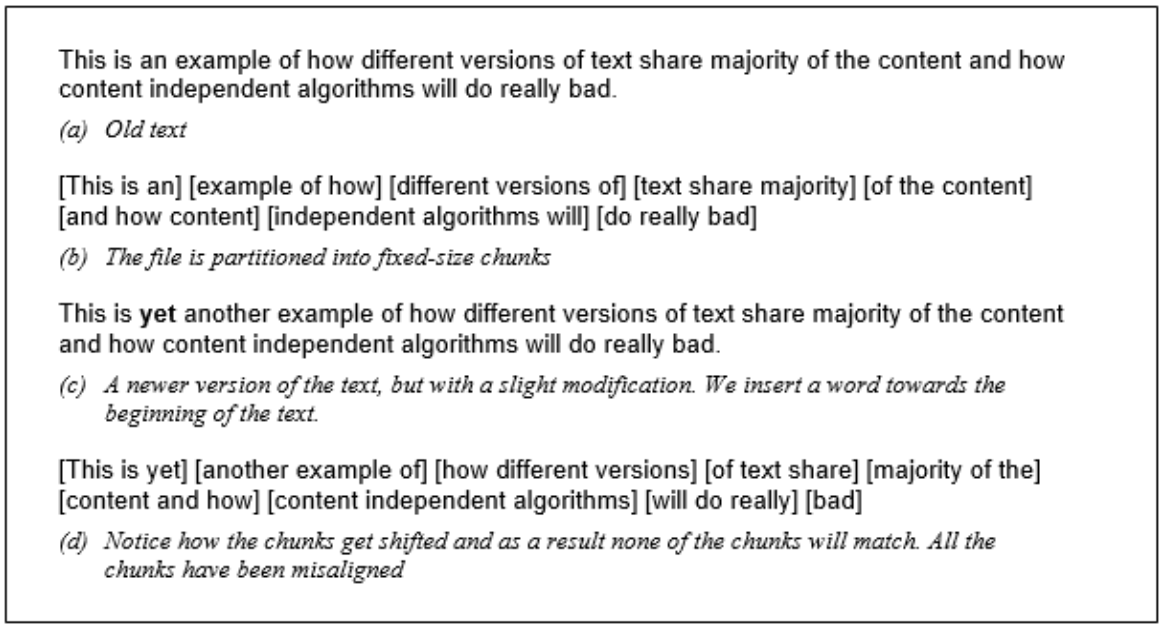


Figure 3: Example of the boundary shifting problem for content-independent chunking

We will analyze four different content-dependent chunking algorithms and assess their performances, such as finding document similarity and speed, in various different datasets. The algorithms we consider are Karp-Rabin [6], Winnowing [14], TDDD [3], and 2Min [1]. We begin the discussion by introducing some background information necessary for understanding how content-dependent algorithms work. In Section 3, we define and discuss the chunking algorithms. In Section 4, we compare these different algorithms by assessing their performances in areas such as speed and ability to find file redundancy. In Section 5, we demonstrate how these algorithms suffer from periodic data. In Section 6, we discuss some applications and in Section 7 we introduce our own modified version of the 2Min algorithm. Finally, in Section 8, we conclude this report.

2. Background

In this section, we provide some background information necessary for understanding how content-dependent algorithms work. First, we show how a file is hashed using the sliding window approach and how chunks are generated based on the cut-points output by the chunking algorithm. Then, we introduce the rolling hash function and how it can be utilized to efficiently hash the whole the document.

2.1 Hashing the File

The first step in running the chunking algorithm is hashing the entire file. The files are hashed using a sliding window approach. We define a window length of size *w* that is used to slide across the document. The content of each window is then hashed and we repeat this procedure until the whole file is hashed.

Figure 4 shows an example of how the sliding window is used to hash the entire file. The chunking algorithms are run on the hashed version of the document and output a list of cut-points. The cut-points are all the hash values that identify the document and are used to slice the document into chunks. A chunk is defined as the range between the byte immediately after the previous cut-point up to and including the next cut-point. Figure 5 shows a conversion from cut-points to chunks. There are two cut-points, 3 and 1, which are denoted by the solid black line.

The chunks are hashed using a low probability collision hashing function such as MD5, which we refer to as a signature. The signatures of the new file are compared against the signatures of the previous file, and only the chunks for which the signatures are missing are stored or sent.

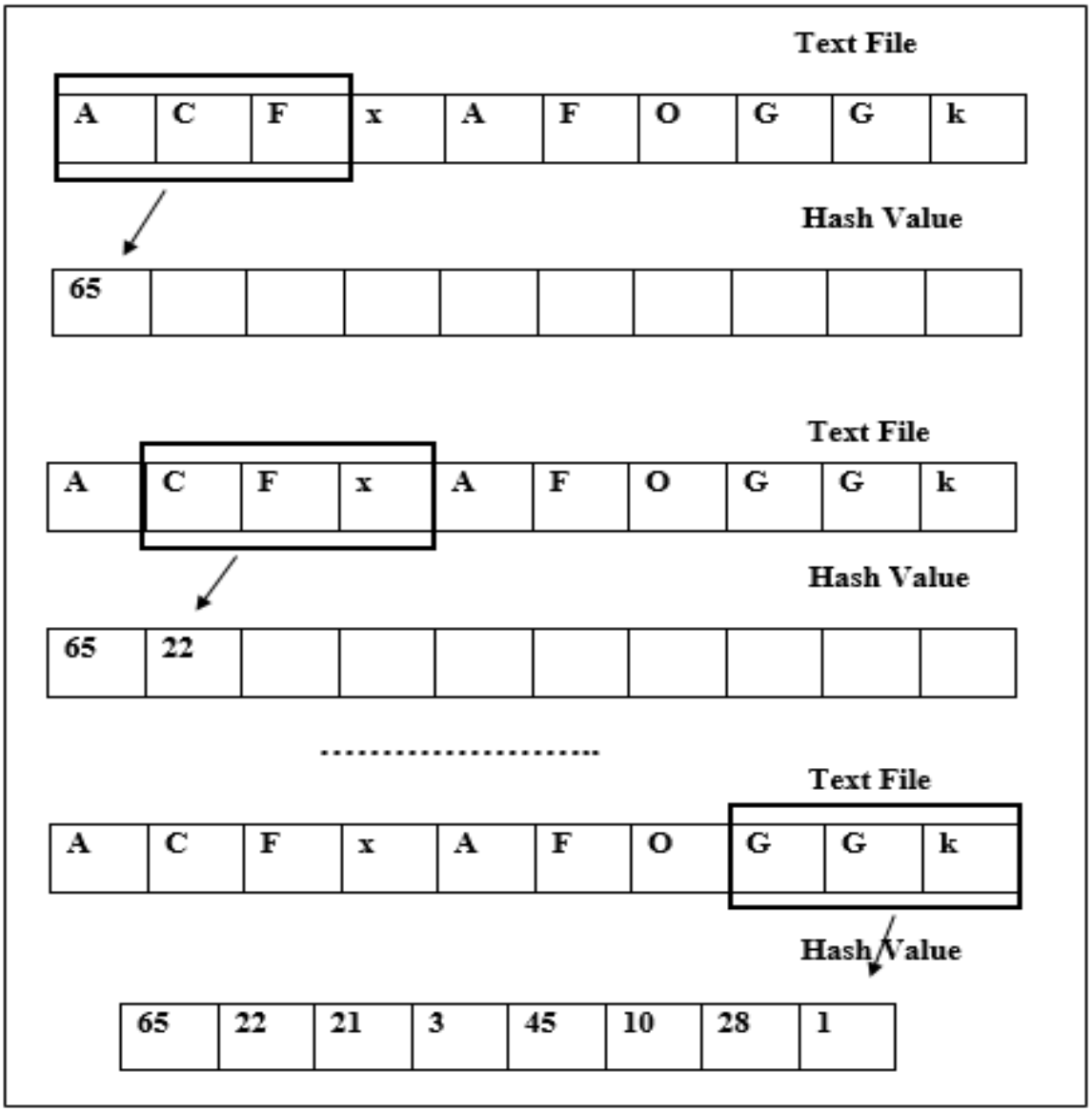


Figure 4: Example of sliding window approach for a file with w = 3

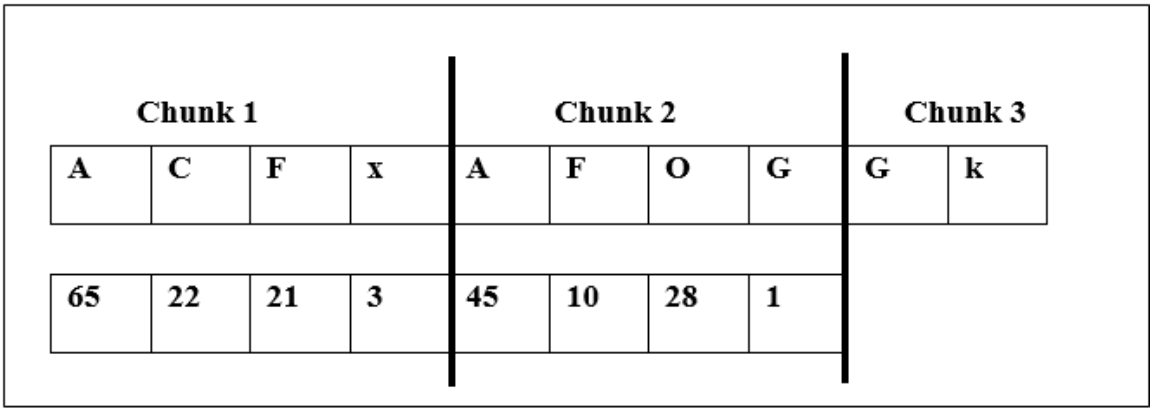


Figure 5: Conversion between cut-points and chunks. 3 and 1 are the cut-points

2.2 Rolling Hash

Hashing the entire document is costly. Another form of a hash function is the rolling hash. Rolling hash functions are fast because they compute the hash of the next window by using the hash of the previous window. The hash value of the new window can be calculated by subtracting off the hash value of the previous byte, which was slid out, and adding in the hash value of the new byte that was slid in. As a result, the hashing step can be computed fairly quickly. Some common rolling hash functions are the Karp-Rabin hash and the Adler hash.

3. Content-Dependent Chunking Algorithms

In this section, we introduce content dependent chunking algorithms. We will discuss the Karp-Rabin, Winnowing, TDDD and 2Min algorithms.

3.1 Karp-Rabin

We start off by introducing the simplest chunking algorithm, Karp-Rabin [6]. We first define the parameters needed for the algorithm. Karp-Rabin determines that a hash value is a cut-point if the hash meets the following condition:

$$H \bmod D = R$$

where *H*, *D*, and *R* are the hash values, divisor, and remainder, respectively. *H* depends on the document content whereas *D* and *R* are selectable. *D* is the divisor and *R* is a predefined constant. *D* determines what the expected block size of the chunks will be [1, 6]. Larger values of *D* result in larger chunks and smaller values of *D* result in smaller chunks. There is a tradeoff between space and coverage. Larger chunks will result in the document being partitioned into fewer chunks; as such less space will be required to store the chunk signatures. However, it is also more difficult to get a matching chunk since it is more probable for a byte to be modified in a larger chunk as opposed to a smaller chunk. Smaller chunks will result in the document being partitioned in more chunks; as such, more space would be required to store the chunks. Since the chunks are smaller in size, there is a higher chance of finding matching chunks between files.

A disadvantage of Karp-Rabin is that there is no minimum or maximum chunk limit, and thus any hash value may be declared a cut-point, which may result in many successive small chunks or a few large chunks, depending on the hash values and the predefined parameters we set. Large chunks are not desirable because they have a lower chance of matching, and small chunks are not desirable because they increase the cost of storing chunk metadata. Figure 6 demonstrates an example of Karp-Rabin's output (the

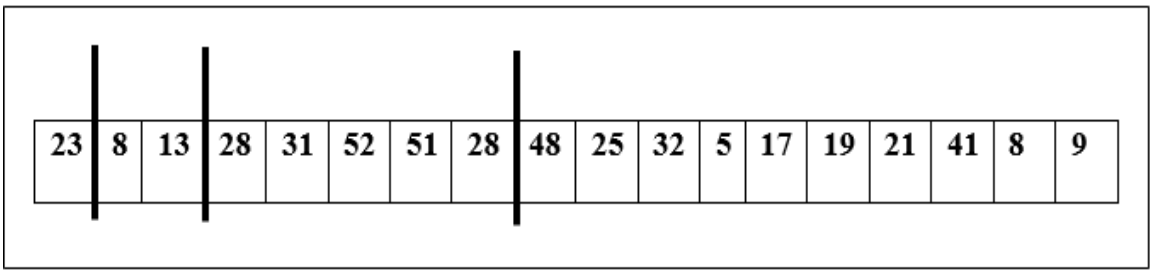


Figure 6: Running Karp-Rabin algorithm on the given array of hash values with D = 10 & R = 3. In this case, [23,13,28] is the sub-array of cut-points.

cut-points are denoted by the solid black lines) when run on a hashed version of a document. The cut-points are then used to partition the document as shown in Figure 5.

3.2 TDDD

Next, we introduce the TDDD algorithm [3]. The TDDD algorithm is structured similarly to Karp-Rabin, but with the added ability to solve Karp-Rabin's deficiency of having extremely small or extremely large chunks. Similar to Karp-Rabin, TDDD defines a hash value to be a cut-point if it meets the condition:

$$H \bmod D = R$$

where *H*, *D* and *R* are the same as described above for Karp-Rabin.

TDDD also defines a couple more parameters: *minT*, *maxT*, and *backup-hash*. *MinT* is a minimum threshold and defines the lower limit for a chunk length. TDDD ignores all the hash values if the current chunk length is less than the minimum threshold and only begins to find a cut-point after the minimum threshold has been satisfied. Similar to *minT, maxT* defines the upper limit for a chunk length, and forces a cut by allowing the *backup-hash* to be the cut-point. If there is no *backup-hash*, the

current hash value is declared the cut-point, even though it failed the condition for being a cut-point. *Backup-hash* values are defined to be cut-points that satisfy:

$$H \bmod D_2 = R$$

where D_2 is a backup divisor, and much smaller than D_1. By setting D_2 to be a smaller value, there is a higher chance of finding a cut-point that satisfies the condition.

An advantage of TDDD is that the minimum and maximum threshold values prevent the chunks from being too small or too large. A major issue of TDDD is that it forces a cut-point when no *backup-hash* is found. As a result, the current hash value is forced to be a cut-point. This violates the content-dependent property of the algorithm and does not guarantee that a different version of the file will have the same forced cut-point, thus potentially misaligning the file chunk boundaries.

We can reduce the possibility of these forced chunks by defining more backup hashes using the same condition as above, but with much smaller divisors. For example, we can define D_3 to be half of D_2, and so on. With this, we can have multiple backup hashes: *backup-hash1* defined by D_2, *backup-hash2* defined by D_3, and so on. When we reach the maximum threshold value of a chunk, we first check if we have a *backup-hash1* and use that as the cut-point. If not, we proceed and check if we have a *backup-hash2*, and so forth. Even with these multiple backup hashes, we may still have a forced cut-point.

Figure 7 shows an example running of the TDDD algorithm. Notice how small chunks and large chunks are prevented due to the extra parameters. The example is color-

coded to make it easier to follow. Red is used to denote the hash values that are ignored because the minimum chunk length is not met. Green is used for hash values that meet the criteria for being a cut-point and blue is used to denote hash values that satisfy the backup hash condition. Yellow indicates a hash value that is forced to be a cut-point. The hash value under consideration is denoted by an arrow.

3.3 Winnowing

The next algorithm we discuss is Winnowing [14]. Winnowing utilizes a different approach in determining whether a hash value is a cut-point. Winnowing defines a sliding window (or boundary) of length *b*. Similar to the sliding window used in hashing the document, the sliding boundary is slid across the whole hash array, sliding the new hash value in and the old hash value out. Initially, Winnowing defined the rightmost smallest hash value in its window and declares it a cut-point. A new cut-point is defined when a new smaller hash value slides in and the previous minimum is still in the window. If the previous smallest slides out, we find the next cut-point by taking the rightmost smallest hash value in the current window. A sample run of Winnowing is shown in Figure 8. The sliding window is denoted by the color green.

The maximum length of any chunk is *b*, which happens when the previous minimum value slides out. Therefore, an advantage of Winnowing is that it is simple and prevents long chunks. A disadvantage however, is that Winnowing does not define a limit for the minimum chunk length. Similar to Karp-Rabin, many successive hash values can be made into cut-points, resulting in numerous very small chunks.

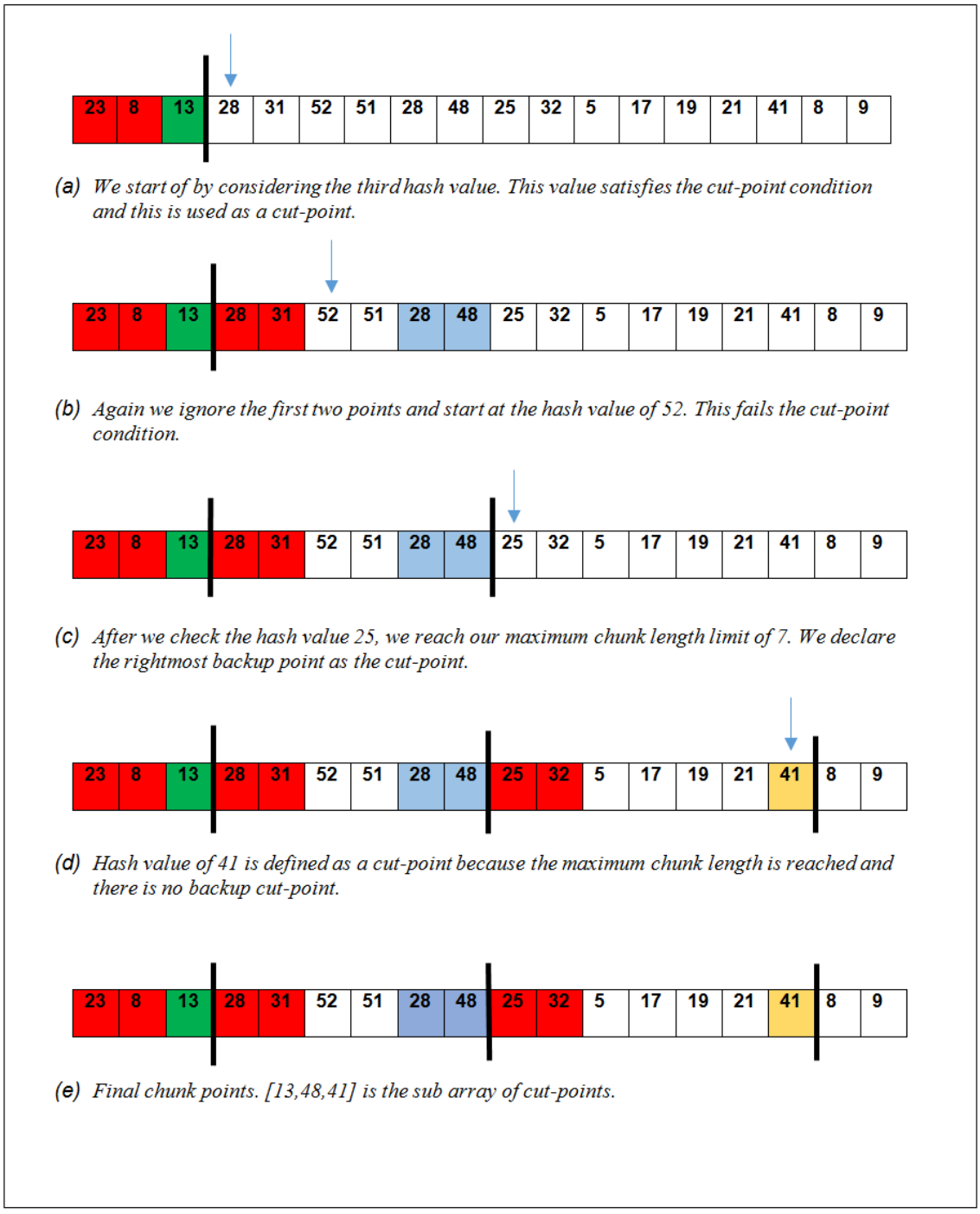


Figure 7: Running TDDD algorithm on the given array of hash values with D = 10, D_2 = 5, R = 3, minT= 2, and maxT = 7.

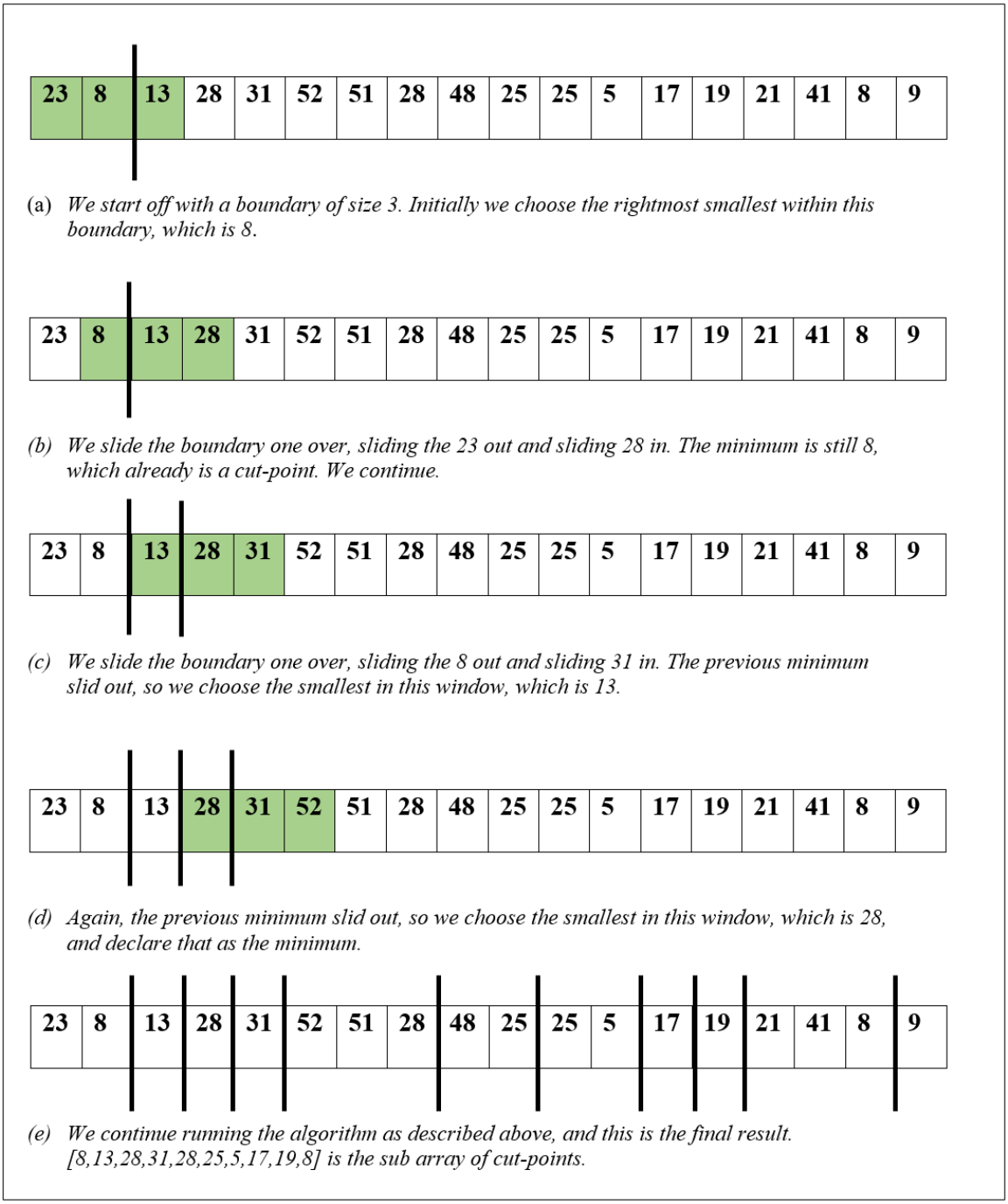


Figure 8: Running Winnowing on the given array of hash values with b = 3.

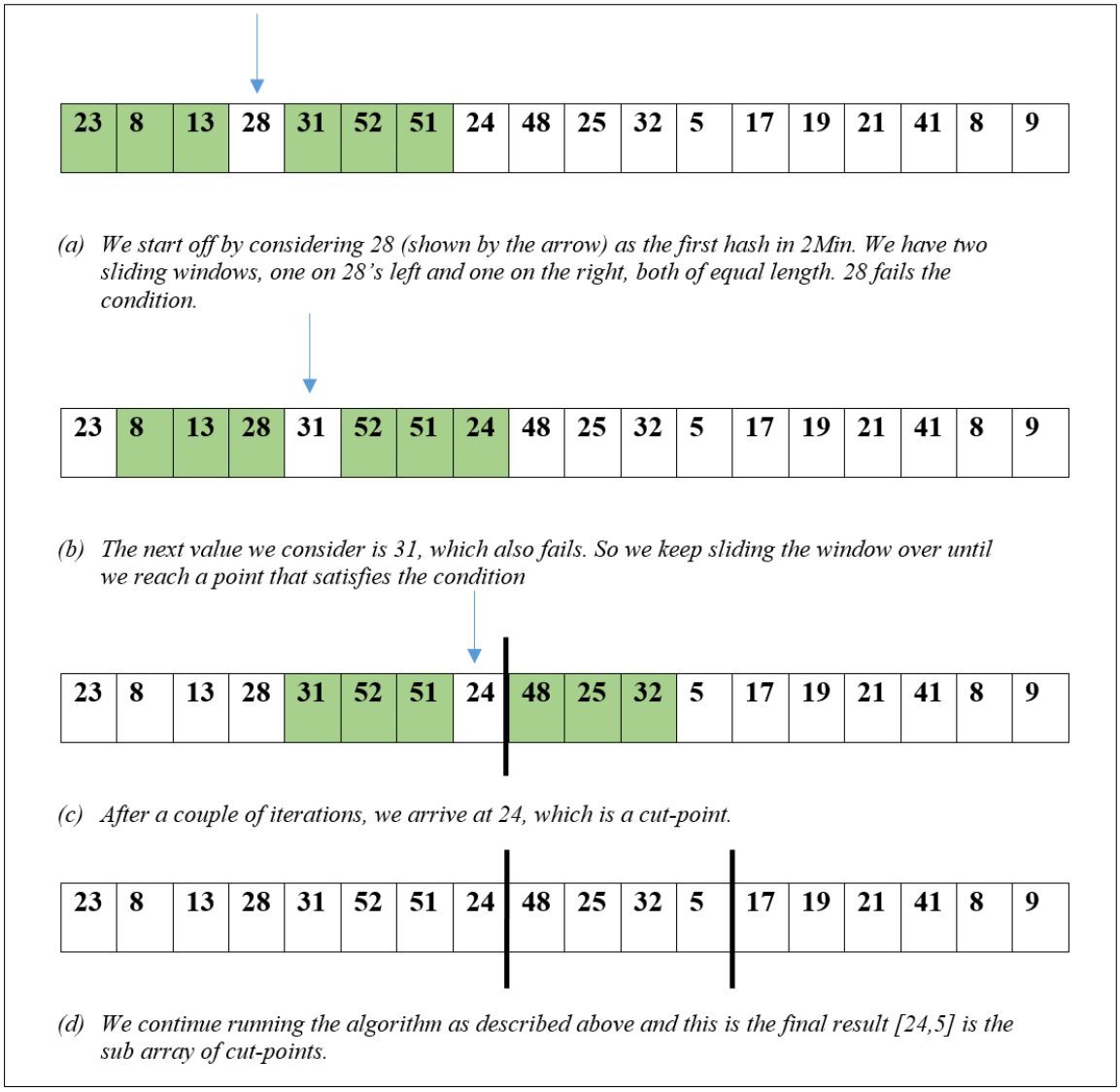


Figure 9: Running the 2Min algorithm on the given array of hash values with b = 3

3.4 The 2Min Algorithm

The final algorithm we consider is 2Min [1, 16]. 2Min also defines a boundary window of size *b*. Unlike Winnowing, 2Min's *b* is a two-way boundary window. A cut-point is defined if the hash value is less than all the hash values up to *b* steps to the left and within *b* steps to the right of the cut-point. The major advantage of 2Min is that it

prevents small chunks because it implicitly sets the lower bounds of the chunk length to *b*. This is how long we have to wait until we can actually start checking for a cut-point. The disadvantage of 2Min is that it may result in large chunks. 2Min could benefit from an upper limit for the chunks, which we propose later. Figure 9 shows an example execution of the 2Min algorithm.

4. Experiments

In this section, we report our experimental results. We ran the algorithms on four different datasets. We first discuss the experimental setup for each of the datasets. In Section 4.1, we discuss the gcc and emacs datasets. In Section 4.2, we discuss the Internet Archive dataset, and in Section 4.3 the random dataset. In Section 4.4, we discuss the results of the experiments. In Section 4.5, we discuss the speed of the algorithms on various sized datasets. In Section 4.6, we discuss the variances among the block sizes for the different algorithms.

In all cases, the parameters of the algorithms started from 100, and were varied differently for each parameter (since each algorithms' parameter affects its respective block sizes differently). The parameters for each of the algorithms were as follows: divisor (D) for Karp-Rabin, sliding boundary (b) for Winnowing, one-sided sliding boundary (b) for 2Min, primary divisor (D_1) for TDDD. For TDDD and Karp-Rabin we set *R* to be 7. We also defined two backup divisors for TDDD, $D_2 = \frac{D_1}{2}+1$, and $D_3 = \frac{D_1}{4}+1$, and set *minT* to 2* D_1 and *maxT* to 8*D_1. The sliding hash window was set to 12.

We also introduce some terms that we use as metrics for comparison. Coverage is the total amount of redundant information found in a newer version that was already present in an earlier version. Ratio is defined as the total coverage divided by the total length of the new file, and gives us a measure of what percentage of the document was already present in a previous version. Average Block Size is the expected block size of the algorithm for a given dataset and set of parameters, and is computed by dividing the total length by the number of chunks found in the newer version. We wanted to determine which of the algorithms was the best in terms of finding document coverage.

4.1 Experimental Setup for gcc and emacs

We ran all four algorithms on two different versions of gcc and emacs source files. We used gcc versions 2.7.0 and 2.7.1 and emacs versions 19.28 and 19.29. The chunking algorithms first computed and stored the chunk signatures of the previous version. The signatures were computed by partitioning the version into chunks and computing a strong checksum of each chunk using the MD5 hash. The signatures of the newer version were computed next. For each matching signature, we updated the coverage by that signatures' respective chunk length. Figure 10 shows the results for gcc and emacs.

4.2 Experimental Setup for Internet Archive Set

In this section, we present the setup for the Internet Archive dataset. We selected three random words from the English dictionary. The three random words were used as a search query in Bing and the top 20 results were selected. Only the top 10 unique results were kept and this process was repeated until we obtained 5000 unique URLS.

Then, we used the Wayback API (https://archive.org/help/wayback_api.php) to retrieve four versions of the page: current, six months ago, a year ago, two years ago. We then filtered the datasets and removed any pages that did not have 4 distinct uncorrupted versions, and were left with 4427 unique pages. The chunk signatures of each of the previous versions were compared to the chunk signatures of the current version. The experiment was repeated for all 4427 versions, and the block size and ratio values were averaged. Figures 11, 12, and 13 show the results for the Internet Archive set.

4.3 Morph Results

Morph datasets are files that are generated randomly by running a program. We first generated two random files, each of length 20MB. We then created a new file by "morphing" the two files that were generated. The morphing was done by setting parameters *p* and *q* which defined the probability of staying in state1 and state2, respectively. The states represented which of the input files we read from next. State1 corresponded to file1 and state2 corresponded to file2. If we were in state1, we stayed in state1 with a probability of *p* and switched to state2 with a probability of *1-p*. If we were in state2, we stayed in state2 with a probability of *q* and switched to state1 with a probability of *1-q*.

Depending on which state we were in, we copied byte *i* to the output morph file from the corresponding input file, where *i* is the current iteration of this step. The next state was computed after each copy as discussed above. We repeated this until we

reached the end of the smaller input file and thus, the size of the output file was *min(file1.length, file2.length).*

Figures 14, 15, and 16 display the experimental results for the morph files generated by morphing file1 and file2, with different probabilities. For the purpose of these experiments, the output file should be similar to one of the input files. That being said, one of the state probabilities should be set very high (~.99) and the other should be set low. We modelled the output morph file similar to file1 and hence, set state1 probability to be high. We discovered that for high values of *p,* the value of *q* is negligible because the state will rarely be in state2 due to state1's high probability (*p)* and state2's low probability (*q*). Thus, only graphs for varying *p* values are shown, with a fixed *q* value of 0.1.

4.4 Results

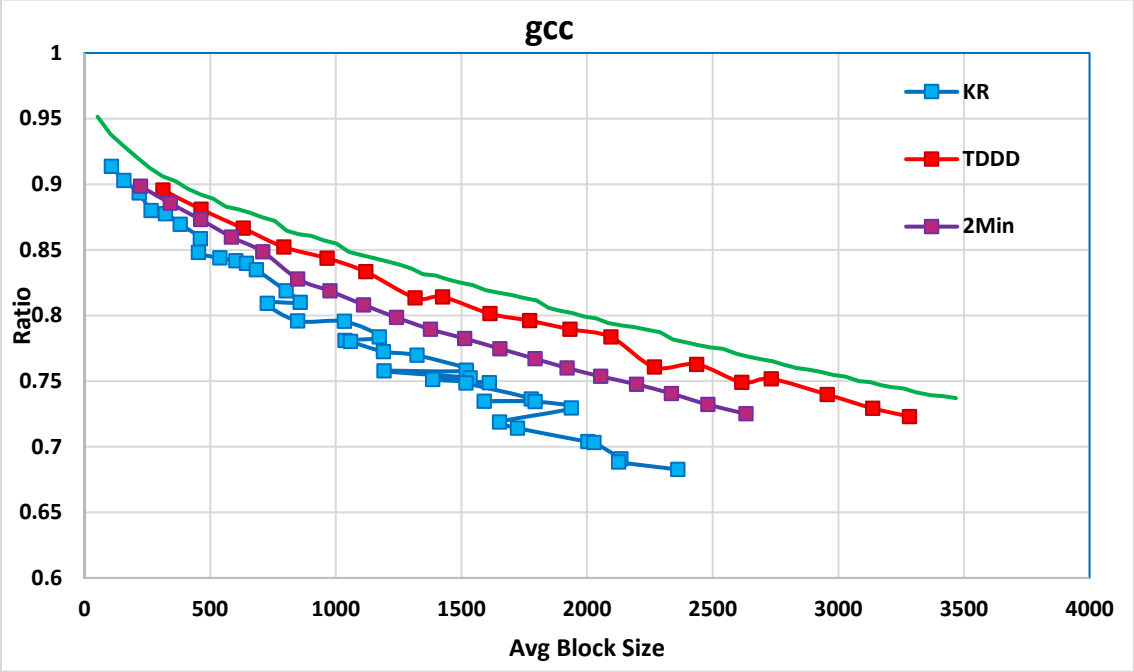
TDDD and 2Min perform better than the other chunking algorithms in terms of finding more coverage for a given average block size. Recall that bigger block sizes that match are desirable because less metadata is required for storing the chunk signatures; this reduce storage costs.

For the morph files, 2Min and TDDD outperform the other algorithms for large values of *p* as shown in Figures 15 and 16. As the value of *p* decreases (Figure 14), there is a higher probability of a random byte coming from file2, thus decreasing the chances of a matching chunk, especially for larger chunks. The algorithms behave differently on random data than on real data.

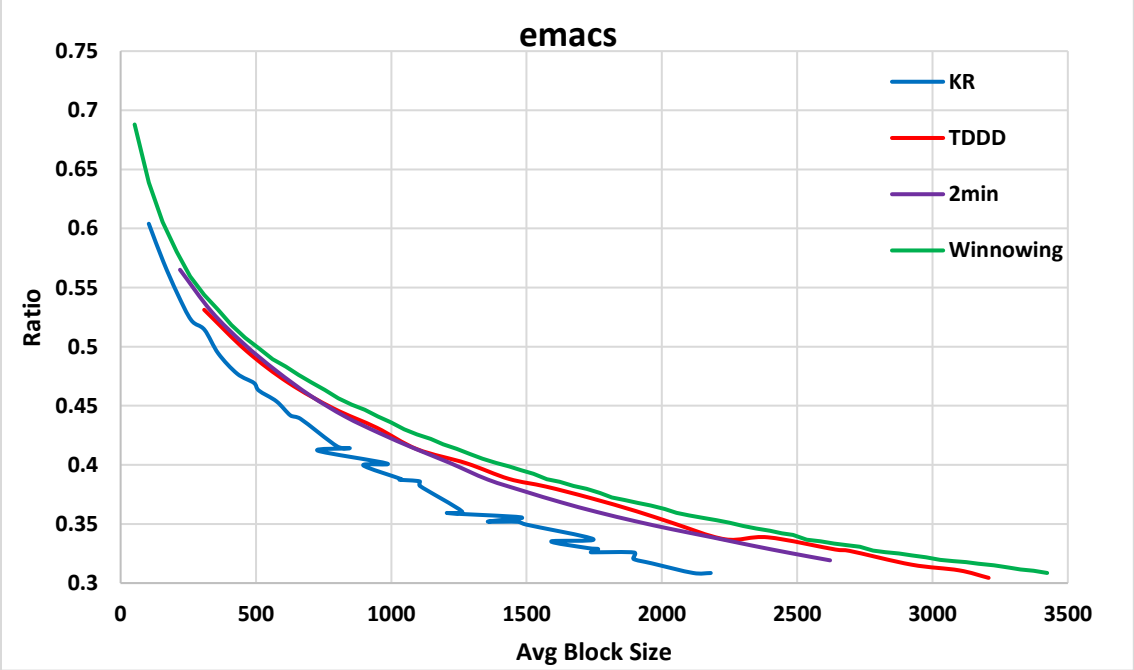
Winnowing outperforms the other algorithms on the gcc and emacs source files. This finding was shocking because TDDD and 2Min outperformed on the other datasets. TDDD outperforms 2Min and Karp-Rabin on the gcc source files. On the other hand, TDDD and 2Min perform equally well on the emacs source files. If we look closely, gcc files have more similarities between the versions compared to the emacs files, and hence, TDDD picks up more document similarity. Karp-Rabin under performed in both datasets.

For the Internet Archive dataset, TDDD does significantly better for larger block sizes rather than 2Min and Winnowing. Karp-Rabin, on the other hand, is very random; sometimes it performs equally well with TDDD and at other times, it does not. One reason for this is that Karp-Rabin does not define a minimum nor maximum limit for the chunk sizes, and may potentially have extremely large or extremely small chunks. We do not connect the points for clarity purposes. These results are different from the other datasets and we speculate two reasons for this.

One reason is that the Wayback API returns the closest version to the version requested, since it is unlikely that Wayback has all the versions for any day available. Even though the four versions were unique for a given URL, it was not guaranteed that those versions were separated out by the time period given. For instance, the Wayback may have returned a version that was crawled last week when requested for a six-month-old version, since it may not have crawled that particular website for a while.



(a) Experimental result for the gcc dataset.



(b) Experimental result for emacs dataset

Figure 10: Experimental results for gcc and emacs

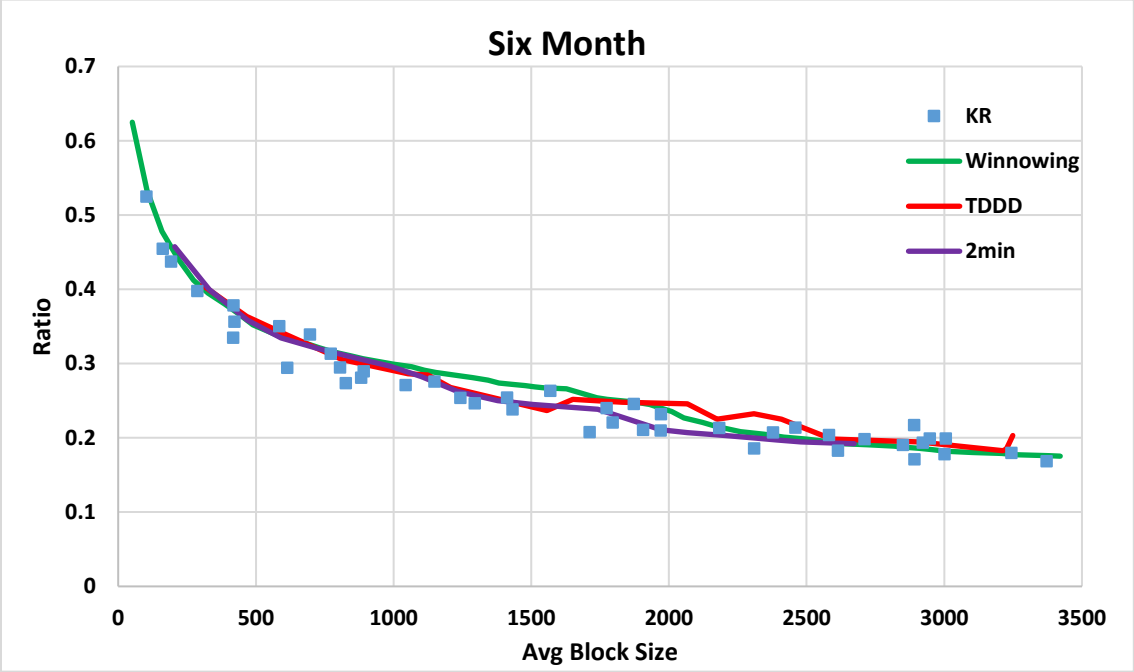


Figure 11: Experimental result for The Internet Archive set. We are comparing the current version to an approximately six-month-old version.

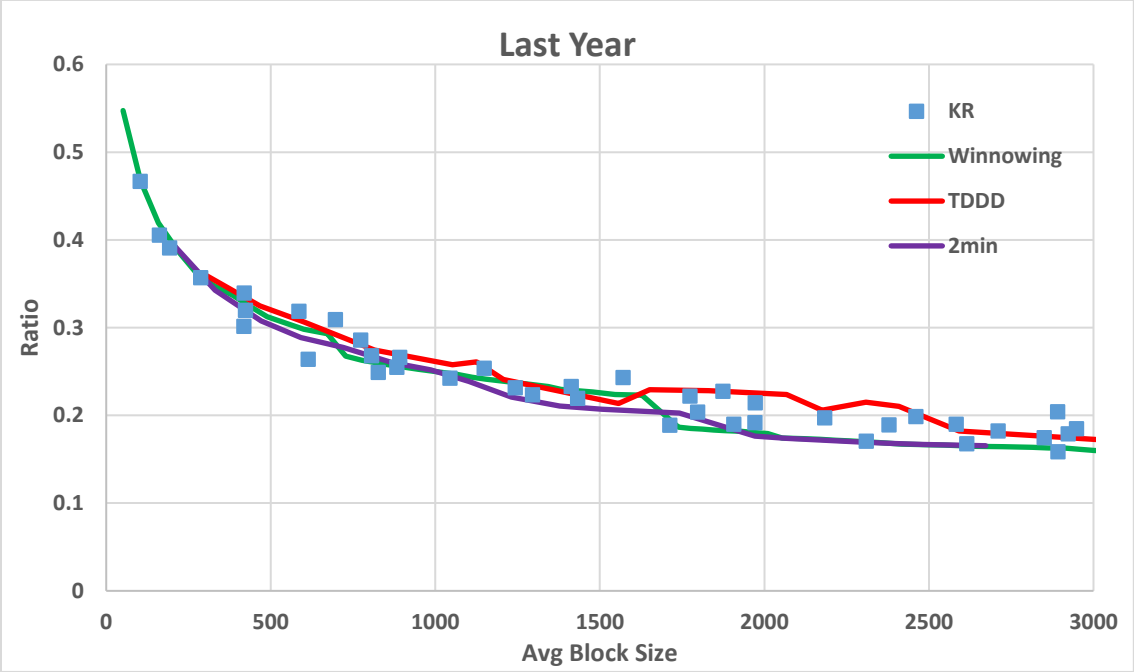


Figure 12: Experimental result for the Internet Archive set. We are comparing the current version with an approximately one-year old version.

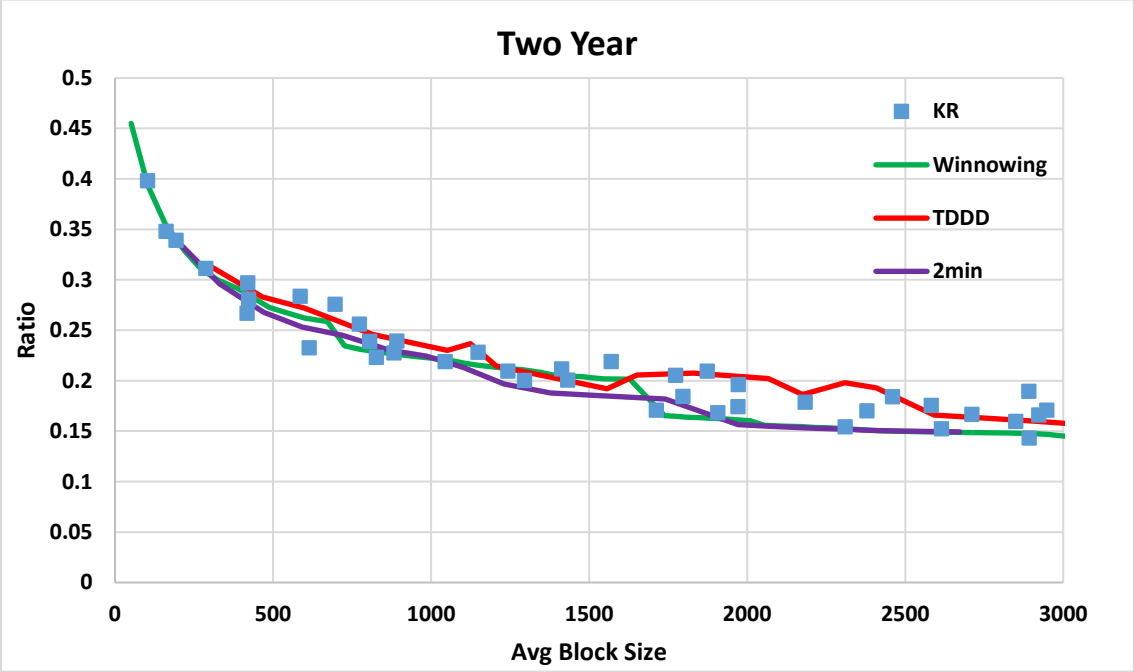


Figure 13: Experimental result for the Internet Archive set. We are comparing the current version with an approximate two-years-old version.

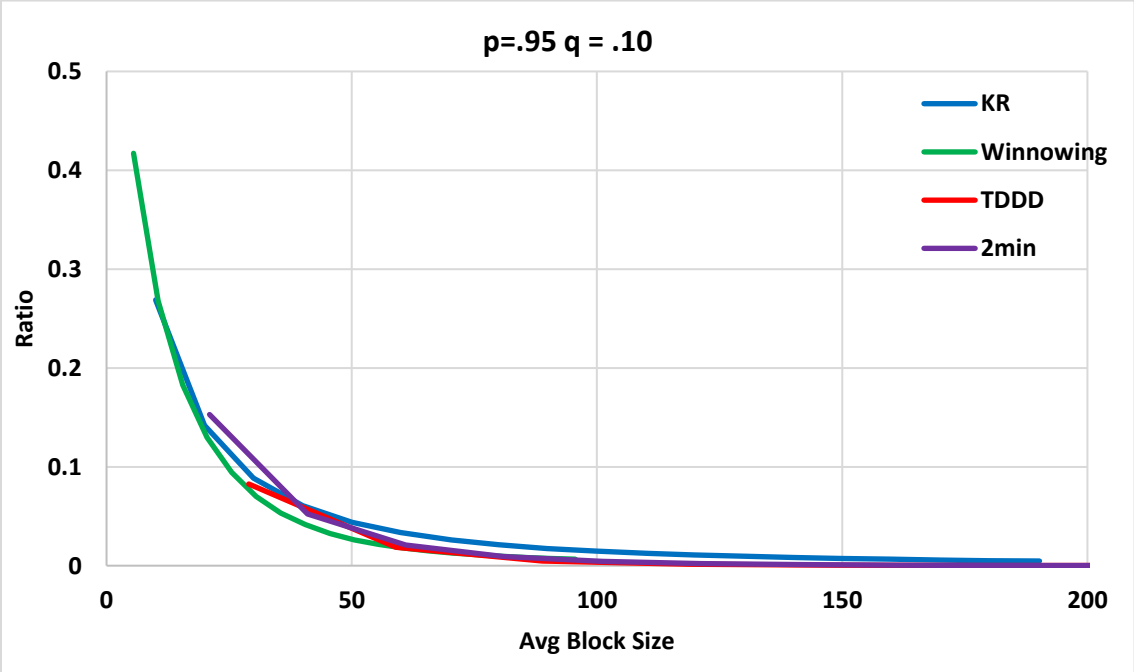


Figure 14: Experimental result for the morph dataset, generated with p = .95 and q = .10.

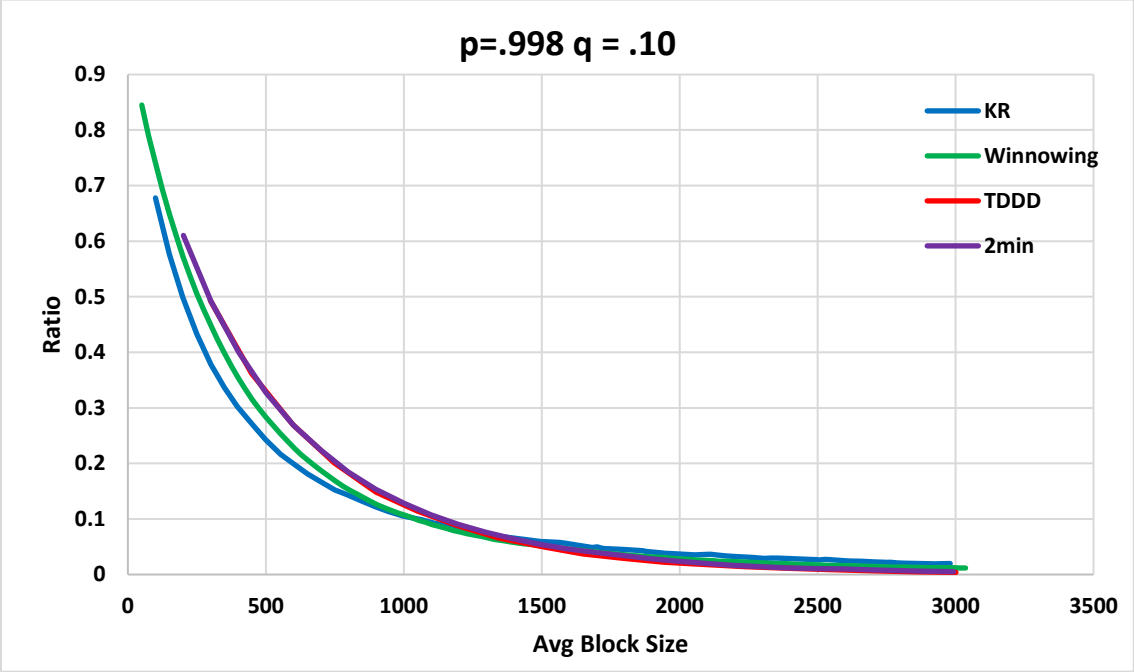


Figure 15: Experimental result for the morph dataset, generated with p = .998 and q = .10.

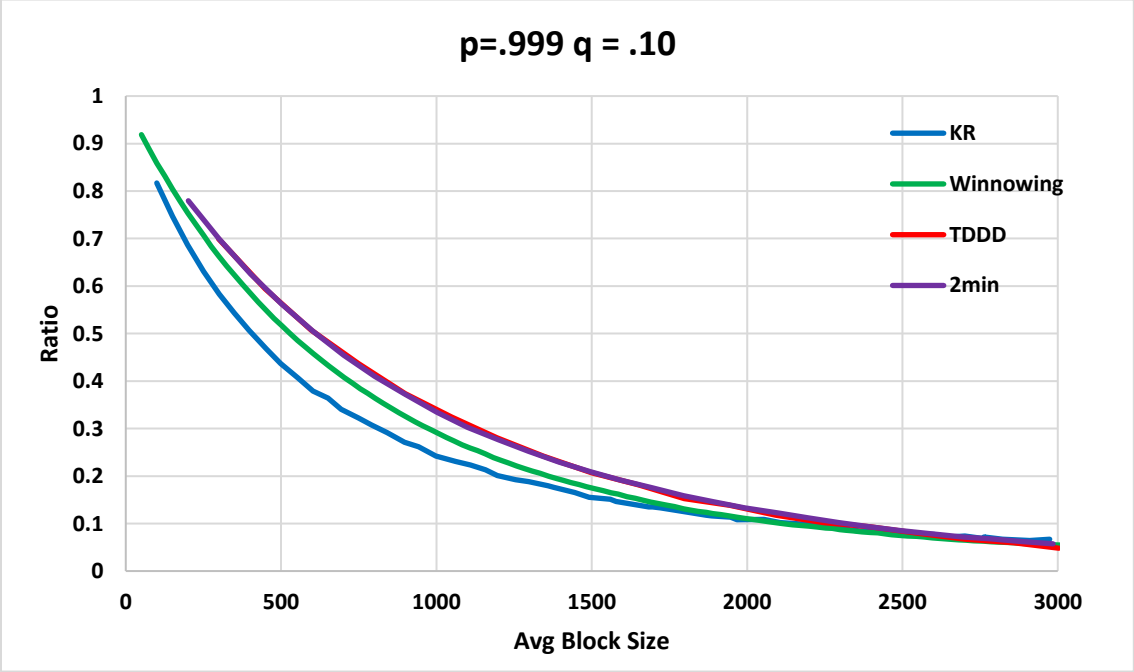


Figure 16: Experimental results for the morph dataset, generated with p = .999 and q = .10.

Another possibility is that the data's periodicity may have influenced the results. This occurs because the Internet Archive dataset is composed of HTML, CSS and JavaScript, and much of the content is repetitive, such as the tags for HTML. When hashing this periodic data, a lot of the hash values may repeat. Therefore, these chunking algorithms may behave unexpectedly. We discuss periodic data in a later section.

4.5. Timing Experiment

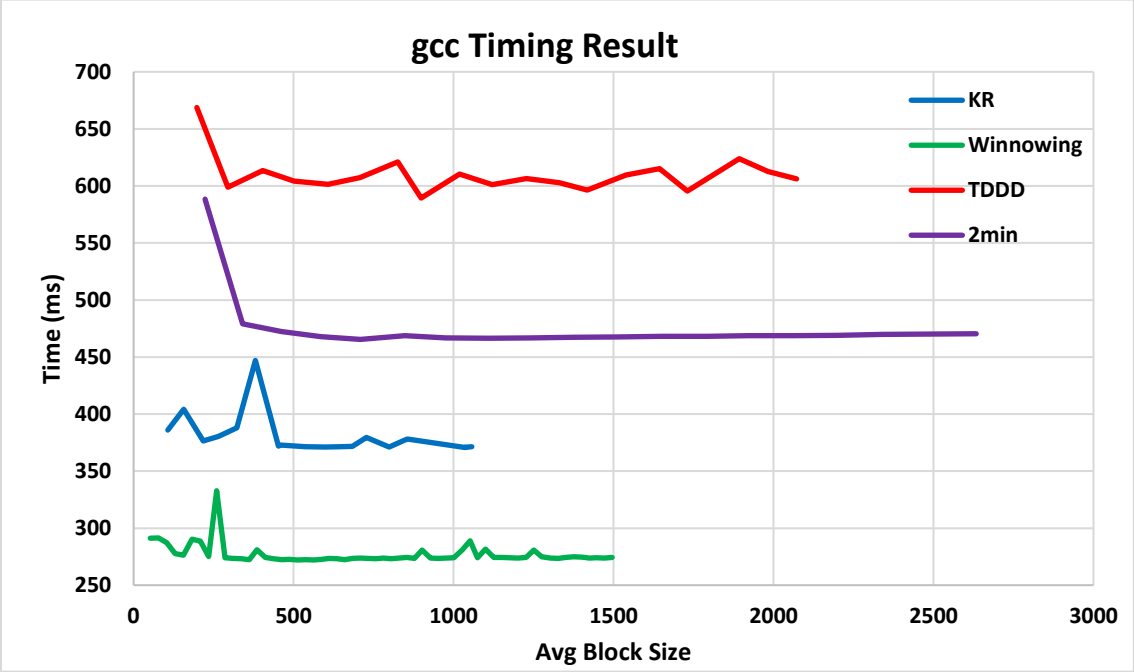
Speed is crucial when running these chunking algorithms, especially on a large dataset. We used three datasets for doing the timing: random, gcc and emacs datasets. The sizes of the datasets were 20 MB, 28 MB, and 40 MB, for the morph file, gcc and emacs, respectively. The results are shown in Figures 17 and 18.

The parameters were the same as described above. We only timed the portion of the program that took the hashed version of the file as input, and computed a list containing all the cut-points. Thus, we did not include the time it took for hashing the document, calculating the chunks from the cut-points, and computing the signature of each chunk. The time for these steps is either negligible, or uniform for all the algorithms. However, each algorithm has its own conditions for computing a cut-point, which affect the speed of the algorithm.

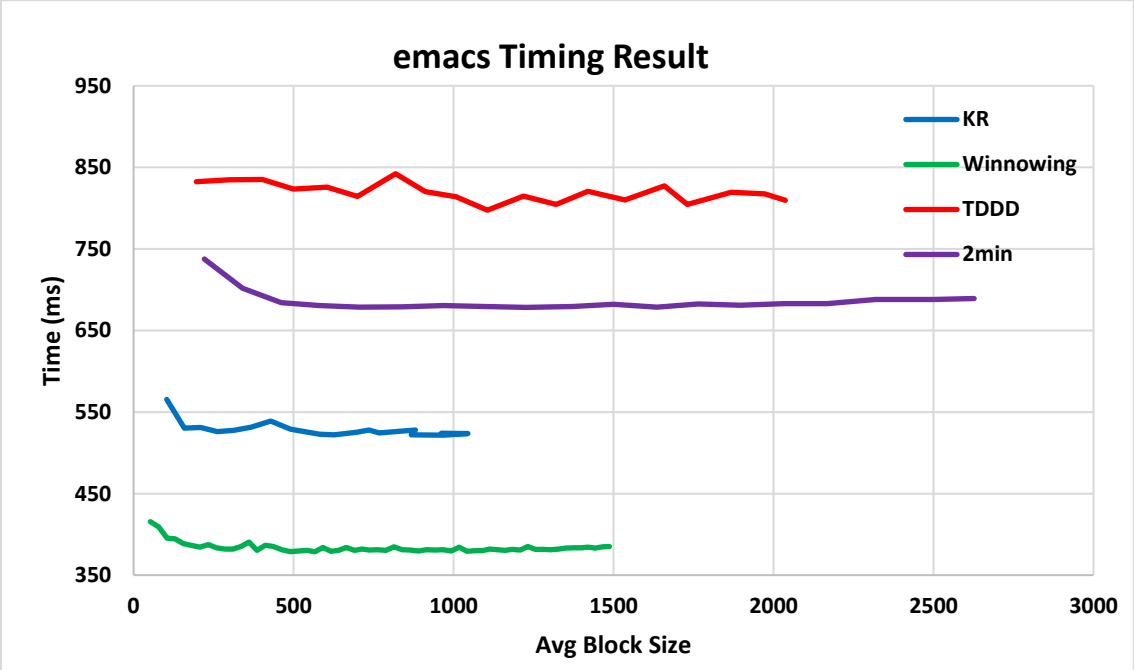
The programs were run on an x64 Windows machine running Windows 10 Pro, with an Intel (R) Core (TM) i5-2410M CPU running at 2.30GHz, using 2 Cores with 6GB of ram. All the programs were written in Java and run on Java version 1.8.0_65, with 4GB of memory allocated to it. The experimental results are shown in Figure 13.

We now briefly discuss how we optimized TDDD, Winnowing and 2Min algorithms for the timing analysis (there was nothing nontrivial to be done for Karp-Rabin). For TDDD, we immediately skip to the point where the minimum chunk condition is met, and do this every time a cut-point is declared. This saves us the time from waiting for the minimum chunk limit to be reached. For Winnowing, we initially find the smallest hash value in Winnowing's window using a for loop (which is also the first cut-point), which we will refer to as the *prevboundary*. We compare the new hash value with *prevboundary*, and declare it cut-point if it is smaller than it. We also update *prevboundary* to be this new value (since this is the current minimum within the window). If *prevboundary* slides out of Winnowing's window, then we loop through the window, find a new valid minimum, and update *prevboundary*. Unlike Winnowing, 2Min has two windows, and we keep track of each windows minimum hash value. We will call the left windows minimum hash value as *leftmin*, and the right window's minimum hash value as *rightmin.* We initially find both *leftmin and rightmin* using a for loop. Then, we compare the current hash value with both *leftmin and rightmin*, and only declare it a cut-point if it is less than both minimums. If either of the stored hash values become invalid, that is, if either hash value slides out of its respective window (2Min has two windows), we update that windows minimum hash value with a new valid hash value.

TDDD performs the slowest compared to the other chunking algorithms, despite outperforming the other algorithms in finding more document coverage. One reason that TDDD is slow is due to its extra conditions. TDDD computes a modulo of the hash using each of the back-up divisors defined, and also checks whether the minimum or maximum chunk limit is reached.



(a) Timing experiment for gcc, which is of size 27 MB.



(b) Timing experiment for emacs, which is of size 40 MB.

Figure 17: Timing experiments for gcc and emacs

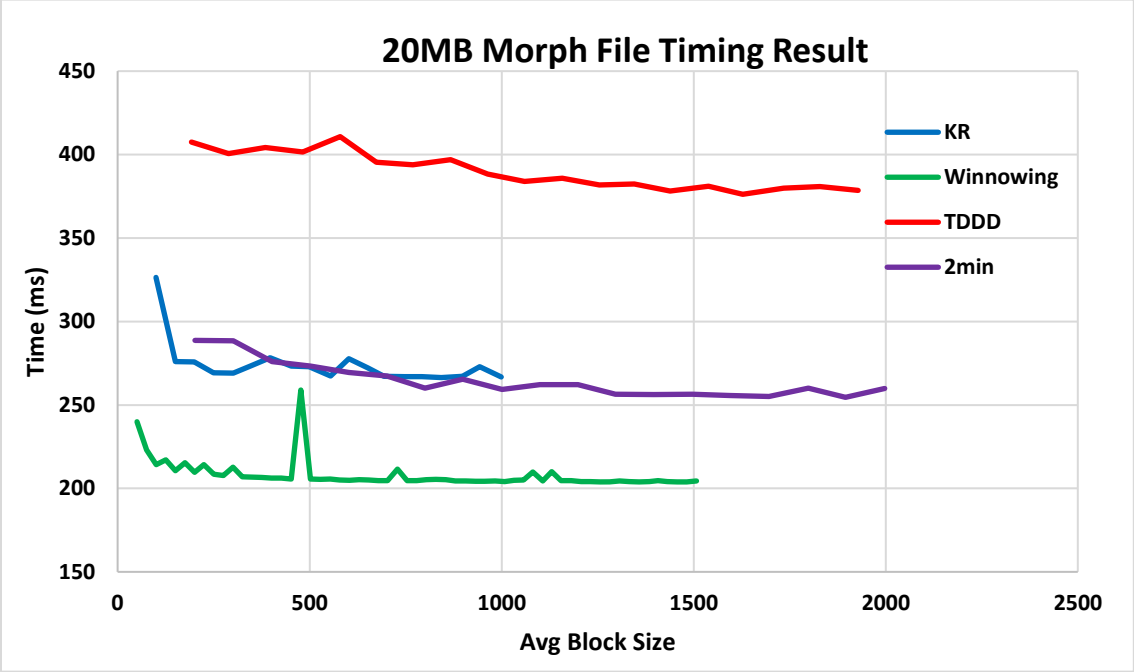


Figure 18: Timing experiment for a morph file, which is of size 20 MB.

4.6 Variances in Block Sizes

Each of the chunking algorithms has an expected block size, which is tunable by each algorithm's respective parameters. It is desirable to partition the document into chunks close to the expected size, to avoid accidentally smaller or larger chunks, as these accidental chops may cause unexpected behavior.

In this section, we present some experimental results on how the block sizes vary for a given chunking algorithm. We generated a random file of size 100MB, and outputted all the different chunk sizes along with their respective frequencies. We set the parameters of the chunking algorithms such that each algorithm had the same expected block size, for a particular file. We ran the chunking algorithms for different normalized expected block sizes and only present the graphs with the expected block size set to 1000,

since all different expected block sizes displayed similar results. The parameter of each algorithm was set to 1000, 2000, 500, and 333 for Karp-Rabin, Winnowing, 2Min, and TDDD, respectively (these values will become apparent in the upcoming sections). Recall that the parameters for each of the algorithms were as follows: divisor (D) for Karp-Rabin, sliding boundary (b) for Winnowing, one-sided sliding boundary (b) for 2Min, primary divisor (D_1) for TDDD. For TDDD and Karp-Rabin we set *R* to be 7. We also defined two backup divisors for TDDD, $D_2 = \frac{D_1}{2}+1$ and $D_3 = \frac{D_1}{4}+1$, and set *minT* to 2* D_1 and *maxT* to 8*D_1.

We present two plots, cumulative frequency and cumulative weighted frequency. The cumulative frequency plot shows the amount of block sizes that fall above and below the expected block size. The plot was generated by first running the chunking algorithms and only saving the block sizes with their respective frequencies. We obtained the cumulative frequency plot by summing the frequency of a block size with that of all the block sizes less than it. The plot is shown in Figure 20.

The cumulative weighted frequency plot shows the total fraction of the file that is contained for any block size. The plot was generated by first running the chunking algorithm and only saving the block sizes with their respective frequencies. We then calculated how much of the file was captured for a given block size, by multiplying the block size with its respective frequency. We obtained the cumulative weighted frequency plot by summing the amount of file captured by the current block size with that of all the block sizes less than it. The plot is shown in Figure 19.

4.6.1 Karp-Rabin

The expected block size for Karp-Rabin, on random files, is $\frac{1}{prob(c)}$, where *prob(C)* is the probability that a hash value is a cut-point, which is $\frac{1}{D}$ [1]. By setting *D* to be 1000, the expected block size for Karp-Rabin is 1000, which is the reason we set Karp-Rabin's parameter to this value. Looking at Table 1, we see that the average block size is close to the expected block size.

Amongst the chunking algorithms, Karp-Rabin suffers from the highest variance in the block sizes (Table 1), and has many large and small chunks. For instance, Karp-Rabin's cumulative frequency plot (Figure 20), shows that 40 percent of the block sizes are less than half the expected block size, and 15 percent of the block sizes are greater than thrice the expected block size. Recall that extremely large chunks are not desirable because they have a lower chance of matching, and extremely small chunks are not desirable because they increase the cost of storage (keeping track of chunk signatures). Looking at the cumulative weighted frequency plot (Figure 19), we see that only about 30 percent of the file is captured by block sizes up to 1000. A lot of the file is captured by large block sizes; 40 percent of the file is contained in block sizes that are greater than twice the average block size.

4.6.2 Winnowing

The expected density for a cut-point is $\frac{2}{b+1}$, where *b* is the sliding boundary length [14] , and thus the expected block size is $\frac{b+1}{2}$. Neglecting the one in the numerator, the

expected block size for 2000 is 1000. Looking at Table 1, we note that the average block size is close to the expected block size.

Looking at the cumulative frequency plot (Figure 20), we see that Winnowing has a uniform distribution of the block sizes. One advantage of Winnowing is that it does not have any extremely large block sizes, but it does suffer from some extremely small block sizes, like Karp-Rabin. Note that Winnowing never exceeds block sizes of length *b* (2000 in this case). Winnowing will be forced to find a new minimum after *b* steps because the old minimum value would have slid out.

4.6.3 The 2Min Algorithm

2Min declares a cut-point with a probability of $\frac{1}{2b+1}$, where *b* is the single sided boundary length [1]. The expected block size is $2 * b$ for large values of *b*. Recall that *b* is the single-sided boundary window size. 2Min compares the current hash value with the hash values that are *b* steps to the left and *b* steps to right, and only declares a cut-point if the hash is less than all the values. Looking at the cumulative frequency plot, we see that one major advantage of 2Min is that it prevents small sized chunks, and the majority of the chunk sizes are close to the expected block size. Looking at the cumulative weighted frequency plot, we see that about 50% of the file is covered by block sizes up to the 1000, and almost all of the file is captured by block sizes up to twice the average block size.

4.6.4 TDDD

TDDD both upper bounds and lower bounds the chunk length sizes and as a result, we can always modify the frequencies by simply changing these parameters. For instance, if

it is desirable to prevent a lot of small chunks or a lot of larger chunks, we can set the minimum threshold to the desired minimum length and set the maximum length to the maximum desirable block size. We experimentally determined the average block size for our parameters, and found that the average block size is three times the main divisor, thus we set D to be 333 (Table 1). Similar to 2Min, a large portion of the block sizes are near the average block size, as shown in the cumulative frequency and cumulative weighted frequency plots (Figure 19).

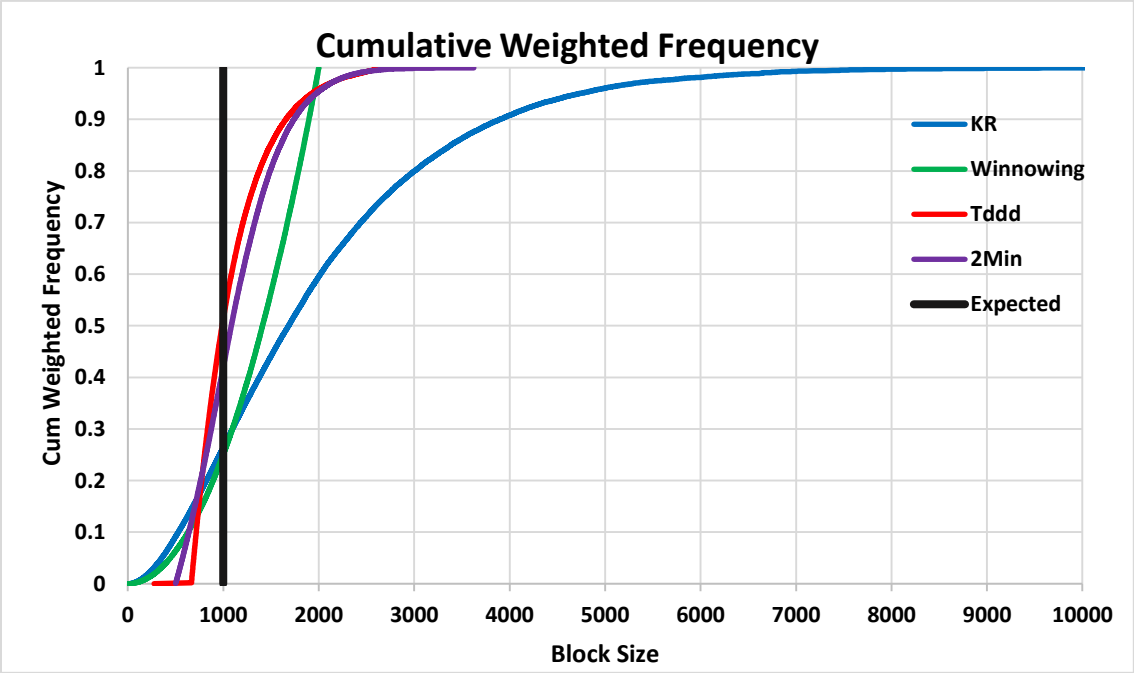


Figure 19: Cumulative weighted frequency plot for all the algorithms. The expected block size for all the algorithms is 1000.

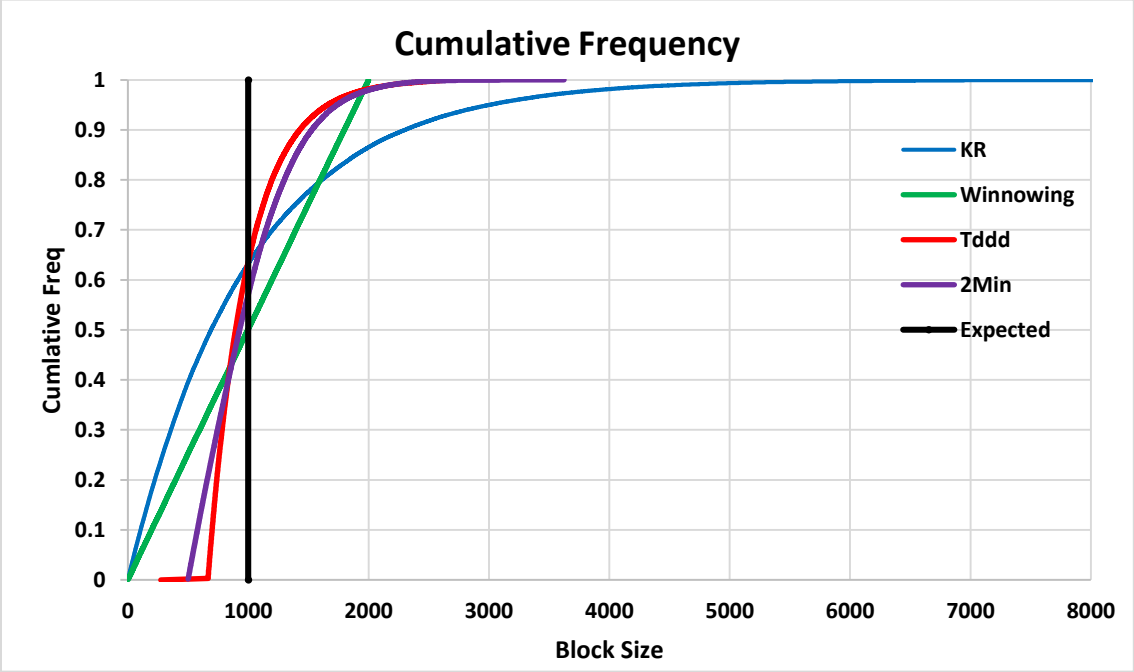


Figure 20: Cumulative frequency plot. Block sizes are shown on the x-axis and frequency on y-axis.

	Karp-Rabin	*Winnowing*	*2Min*	*TDDD*
Avg Block Size	1000.42	997.6356	1003.875	995.5995
Variance	1000467	333722.45	147958.2	105898.1
STD Dev	1000.233	577.68716	384.6533	325.4199

Table 1: Average block size, variance, and standard deviation of the block sizes for of the chunking algorithms, on a 100 MB random file.

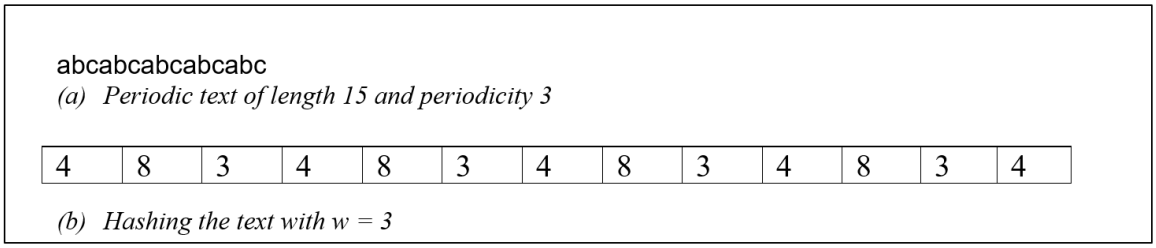


Figure 21: Example of how hash values repeat for periodic text

5. Periodic Data

These chunking algorithms do not work well for all types of the data. An exception is periodic data, where the content is repeated every couple of bytes. This may occur because a lot of the content is recurring, as in the case of block tags for HTML and JavaScript, and hence the hash values of the file are repeated. Figure 21 shows a simple case of periodic text and how the hash values repeat.

Since the hash values repeat, the chunking algorithms may produce either very small or very large chunks. Either the condition of these chunking algorithms will be met multiple times, depending on how often the content is repeated, or it will not be met at all. TDDD is an exception because it prevents small and long chunks. Winnowing only prevents long chunks, but may suffer from many small chunks.

TDDD will force a cut-point even on periodic data, but will then suffer from the boundary shifting problem if the maximum threshold is reached and no backup hashes are found. This may affect TDDD's ability to find coverage. Winnowing on the other hand, will always partition the document into chunks in a content-dependent way. In this section, we show experimental results on a periodic dataset, and discuss the impact on the performance of these algorithms.

5.1 Periodic Datasets Setup

We generated periodic datasets similarly to the way we generated the morph datasets. The key parameters for the periodic datasets are the periodicity interval and the actual content that is repeated. The dataset was generated in the following way:

Step 1: We first generated multiple files with varying periodicity intervals, *t*. The code randomly generated content equal to *t* and then copied the content repeatedly until we had a file of at least 10MB.

Step 2: We repeated the above step for each *t* value with a different random seed, which generated different data until we had 100 files for each *t*. The reason we did this is because whether the cut-point condition was met or not depends on the actual content itself. Hence, we had 100 files with the same periodicity but different content.

Step 3: We generated another file, which was not periodic, of size 10 MB. We used this file to create the morph file, which is explained in the next step.

Step 4: For each periodic file, we created a morph file by combining the periodic and nonperiodic file (generated in step 3) with $p = .998$ and $q = .70$. State1 corresponded to reading from the periodic file, and state2 corresponded to reading from the nonperiodic file. At the end of this step, we had 100 morph files for each *t*.

Figures 23, and 24 show the results. The algorithms displayed the same behavior for different *t* values, and thus we only present the graphs for files with *t* of 10 and 30. Karp-Rabin's output is shown in Table 2. We display Karp-Rabin's output in a table to better

show the randomness of Karp-Rabin on the periodic dataset (this was not apparent when graphed). We now discuss how each algorithm is affected by the periodic content.

5.2 Karp-Rabin

Karp-Rabin declares all hash values as cut-points if they satisfy the condition:

$$H \ mod \ D = R$$

where *H*, *D* and *R* are hash value, divisor and remainder, respectively. For periodic data, the hash values will either meet the condition successively in short intervals (depending on how often the data is repeated and thus produce many cut-points), or fail to meet the condition entirely, and output very large chunks. For instance, Figure 22 shows an example of Karp-Rabin being run on the periodic dataset. In Figure 22 (a), Karp-Rabin declares many small sized chunks because the condition is satisfied each time the data is repeated. In Figure 22 (b), Karp-Rabin fails to find chunk points and ends up hashing the whole document in a single chunk. It depends on the parameter settings for Karp-Rabin and the file content. We fixed Karp-Rabin's *R* parameter to 7, and only modified the *D* parameter, shown Table 2.

In our experiments, Karp-Rabin behaved randomly on the periodic dataset; sometimes it found document coverage, and other times it did not. Karp-Rabin either produces many small chunks, in which case it will find document coverage, or produce extremely large chunks, in which case it will not find any document coverage. In Section 4.6, we saw that Karp-Rabin's expected block size depends its *D* parameter. However, this is not the case for periodic data; the expected block size depends on the periodicity of

the periodic data. The result for Karp-Rabin on periodic data is shown in Tables 2. This result was similar throughout the rest of the periodic dataset and thus, we only show the output for a couple of files. We do not plot the data for clarity purposes.

5.3 Winnowing

Winnowing declares a cut-point if the hash value is the rightmost minimum within its boundary length, *b*. Unlike Karp-Rabin, Winnowing is always guaranteed to partition the document into multiple chunks because it upper bounds the chunk length. Winnowing is the most robust to periodic data because it will always partition the document into chunks in a content-dependent way. Looking at Figures 23, and 24 we see that Winnowing performs well on the periodic data, and unlike TDDD, it always finds coverage in the periodic data.

5.4 The 2Min Algorithm

2Min declares a cut-point if the hash value is strictly less than all the hash values within *b* steps from the left and right of the current hash value. Unlike Winnowing, 2Min ignores the hash value if the hash value has ties. As a result, 2Min may fail to output cut-points if the value of *b* is greater than or equal to the periodicity of the data. For instance, the periodicity of the data in Figure 18 is 3; 2Min will fail to declare a cut-point for all values of *b* greater than 2 because it will fail to find a local minimum.

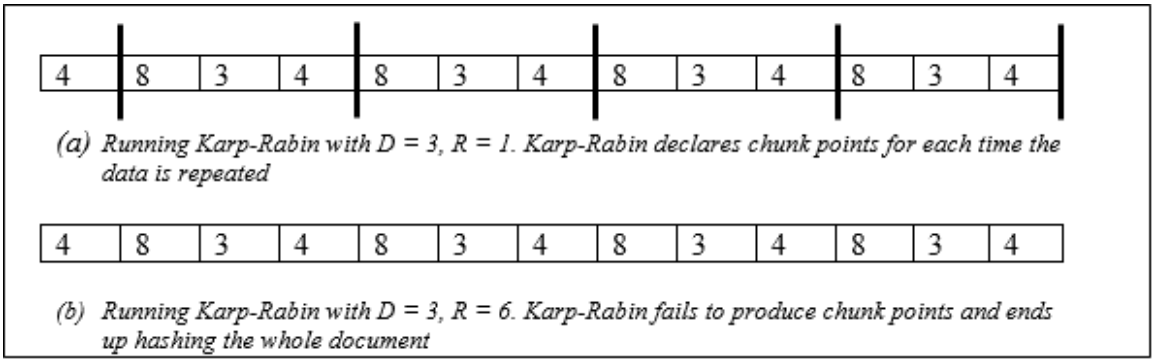


Figure 22: Example of Karp-Rabin being run on periodic data

If we run 2Min with *b* equal to 3 for the data shown in Figure 21, the smallest value |3| will always collide with the next |3| and this will keep occurring until we reach the end of the file.

Taking a closer look at the results, we conclude that 2Min fails to find cut-points when its boundary window is greater than or equal to the periodicity of the data. For instance, in Figure 23, 2Min fails to find cut-points because the local boundary starts at 10, which is not less than the periodicity of 10. In Figures 24, 2Min finds cut-points for all local boundary values that are less than the periodicity |30|, and zero cut-points otherwise.

A simple modification to 2Min by allowing a cut-point to be declared if the hash value is less than or equal to all the values within its neighborhood solves this problem. As a result, 2Min performs significantly better on periodic data while showing little to no change on other datasets. Figures 24 and 25 show an example. Unlike TDDD, 2Min always partitions the document in a content-dependent way, and hence always finds coverage between the documents.

5.5 TDDD

TDDD's condition to declaring a cut-point is similar to Karp-Rabin's but with additional constraints, as discussed in Section 3. Unlike Karp-Rabin which suffers from the periodicity of the data and ends up declaring too many chunks or too few, TDDD does not have these problems. TDDD prevents small chunks because all chunks have to meet the minimum threshold value, and prevents large chunks because it forces a cut-point when the chunk size reaches the maximum threshold value. TDDD also has backup conditions which are more relaxed, and thus has a higher chance of finding a content-dependent cut-point. This is shown in Figures 23 and 24. The major down-side of TDDD is that TDDD may force a cut-point in a content-independent way, which can cause the boundary shifting problem and prevent chunks from matching. This is seen in Figure 24 (b), where TDDD does not find any coverage for some of its parameter settings. TDDD was more robust to periodic data than Karp-Rabin in our experiments due to its extra conditions.

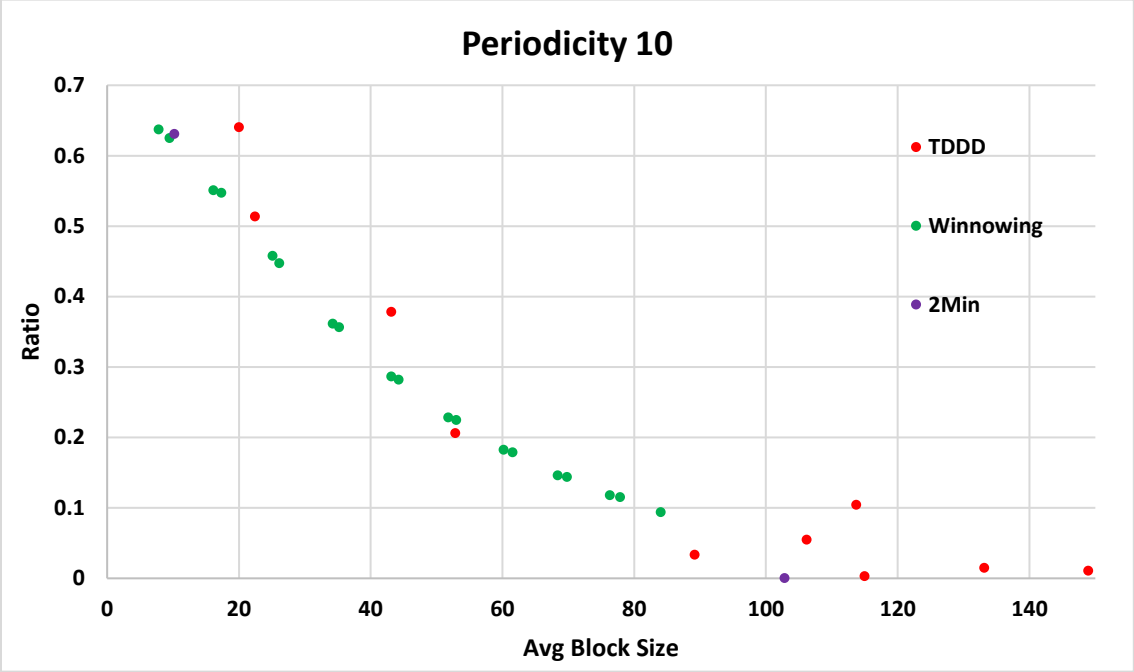
Divisor	File 1		File 2		File 3		File 4	
	Avg Block Size	Ratio	Avg Block Size	Ratio	Avg Block Size	Ratio	Avg Block Size	Ratio
10	9.993125	0.635372	5.702672	0.677888	5.7133	0.682848	3.999066	0.666007
15	60.16521	0	4.131105	0.706894	5.998861	0.650026	10.91013	0.622046
20	11.39789	0.628179	79.60896	0	6.150054	0.677095	11.42592	0.620345
25	99.83727	0	11.74205	0.624321	11.75859	0.622556	100.1663	0
30	120.1533	0	6.306554	0.672478	12.00182	0.621189	11.98719	0.617916
35	139.2176	0	138.4907	0	139.7722	0	12.16464	0.617521
40	160.87	0	158.2579	0	12.30085	0.619625	159.65	0
45	181.5442	0	12.3954	0.620355	12.41306	0.61896	180.2939	0
50	202.6876	0	12.46325	0.620054	203.1983	0	201.3288	0
55	12.55337	0.620361	12.53065	0.617896	6.470097	0.671104	226.2136	0
60	238.7661	0	239.2802	0	12.6294	0.617346	235.2166	0
65	257.4003	0	257.938	0	253.6976	0	12.67329	0.614943
70	280.9778	0	277.0006	0	279.6577	0	12.71808	0.61526
75	301.0507	0	12.75097	0.618282	12.76373	0.616291	302.0418	0
80	315.9558	0	315.4972	0	315.189	0	328.2132	0
85	334.0125	0	326.4453	0	343.2887	0	341.4368	0
90	362.227	0	12.84594	0.617556	364.7638	0	358.8345	0
95	378.6301	0	385.7876	0	392.2492	0	388.259	0
100	401.7839	0	391.0986	0	412.3031	0	405.0223	0

(a) Different outputs for Karp-Rabin on the periodic dataset, for files with periodicity of 10.

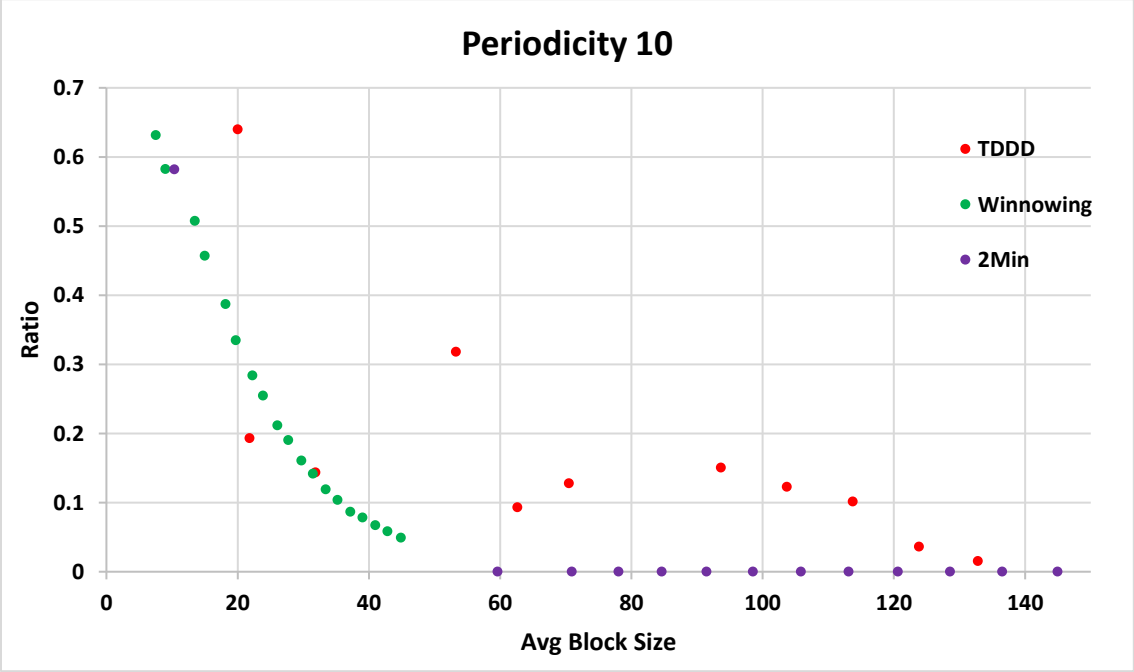
Divisor	File 1		File 2		File 3		File 4	
	Avg Block Size	Ratio	Avg Block Size	Ratio	Avg Block Size	Ratio	Avg Block Size	Ratio
10	6.663508	0.589749	9.996062	0.559371	5.704222	0.643246	10.0032	0.618708
15	14.99233	0.538491	60.36545	0	24.00199	0.421391	24.00665	0.41968
20	8.88379	0.580224	80.26665	0	7.256235	0.635687	11.41466	0.613537
25	16.66753	0.507419	16.64983	0.496324	28.53797	0.418414	16.65202	0.530815
30	30.02153	0.413144	120.0353	0	29.93283	0.41711	121.029	0
35	31.11781	0.414532	12.15874	0.618512	31.05828	0.417644	31.10136	0.415831
40	17.76594	0.493927	161.4286	0	12.28228	0.598047	17.75496	0.55474
45	17.97317	0.534045	181.0352	0	32.65825	0.41571	32.68696	0.414866
50	18.1699	0.50549	18.14286	0.494283	33.22233	0.415357	203.3351	0
55	33.81595	0.412833	33.85041	0.413009	18.28117	0.48281	219.6696	0
60	34.29315	0.411545	240.7497	0	34.22519	0.414745	239.2807	0
65	34.64434	0.412585	262.9093	0	34.61221	0.4147	262.4471	0
70	35.01089	0.412148	18.64687	0.53475	34.92347	0.415097	283.5116	0
75	303.2331	0	298.8203	0	35.27394	0.414113	35.19733	0.413723
80	35.53705	0.411353	319.4997	0	35.53465	0.413913	18.79484	0.552363
85	340.1136	0	35.78119	0.411847	335.5599	0	347.7663	0
90	35.99616	0.410918	358.565	0	35.90491	0.413631	364.074	0
95	384.5718	0	381.651	0	384.9271	0	389.2118	0
100	19.03062	0.504188	399.4097	0	36.23366	0.413609	401.6879	0

(b) Different outputs for Karp-Rabin on the periodic dataset, for files with periodicity of 30.

Table 2. Output of Karp-Rabin on some files from the periodic dataset. Karp-Rabin is very random.

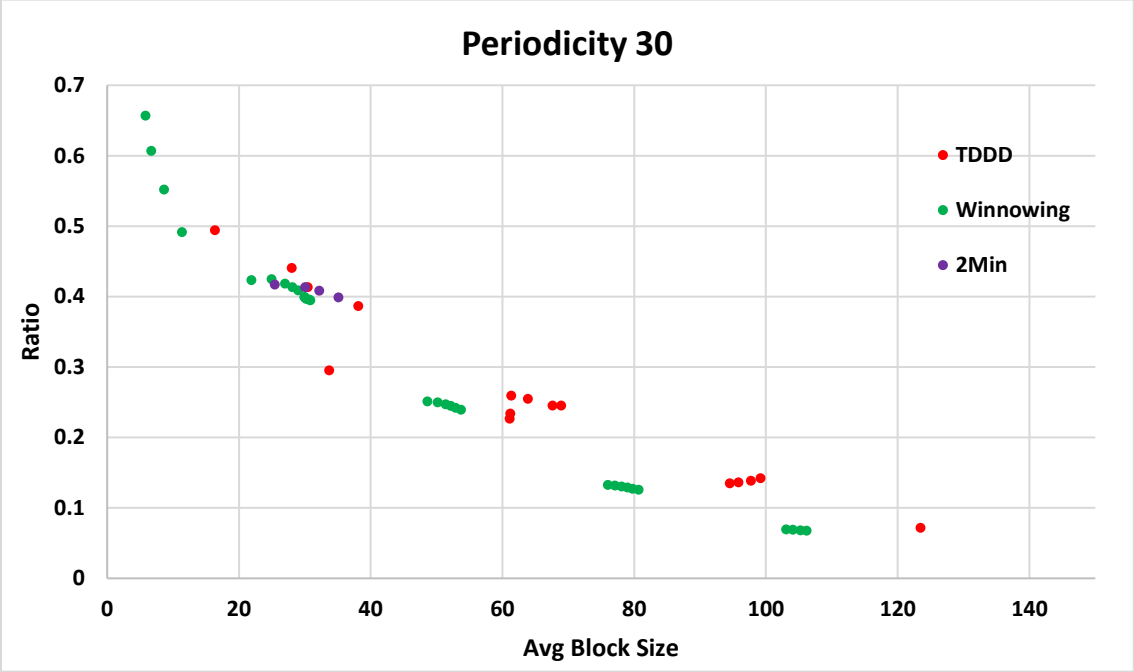


(a) Experimental results for a particular file from the periodic dataset.

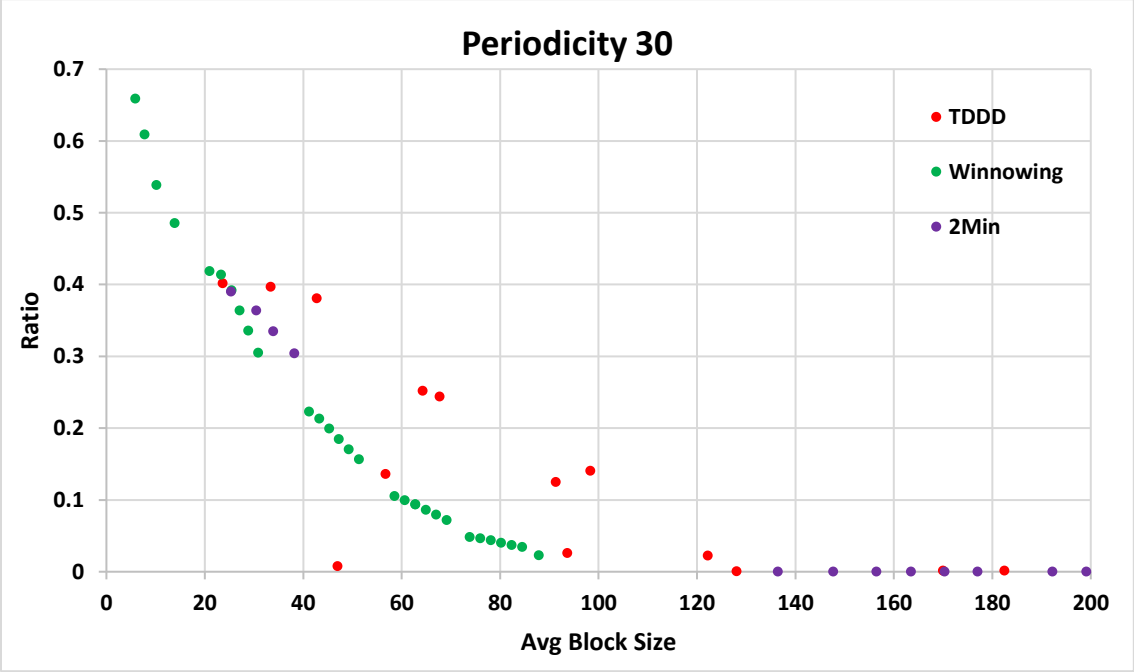


(b) Experimental results for a particular file from the periodic dataset.

Figure 23: Experimental results for periodic data with periodicity of 10

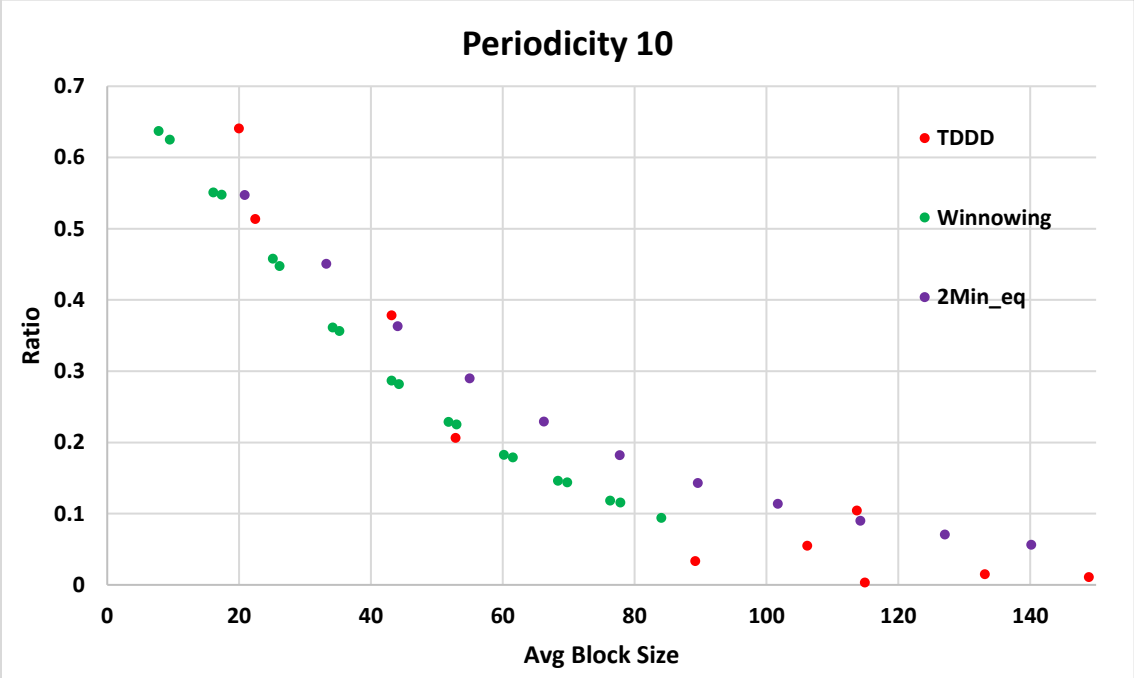


(a) Experimental results for a particular file from the periodic dataset.

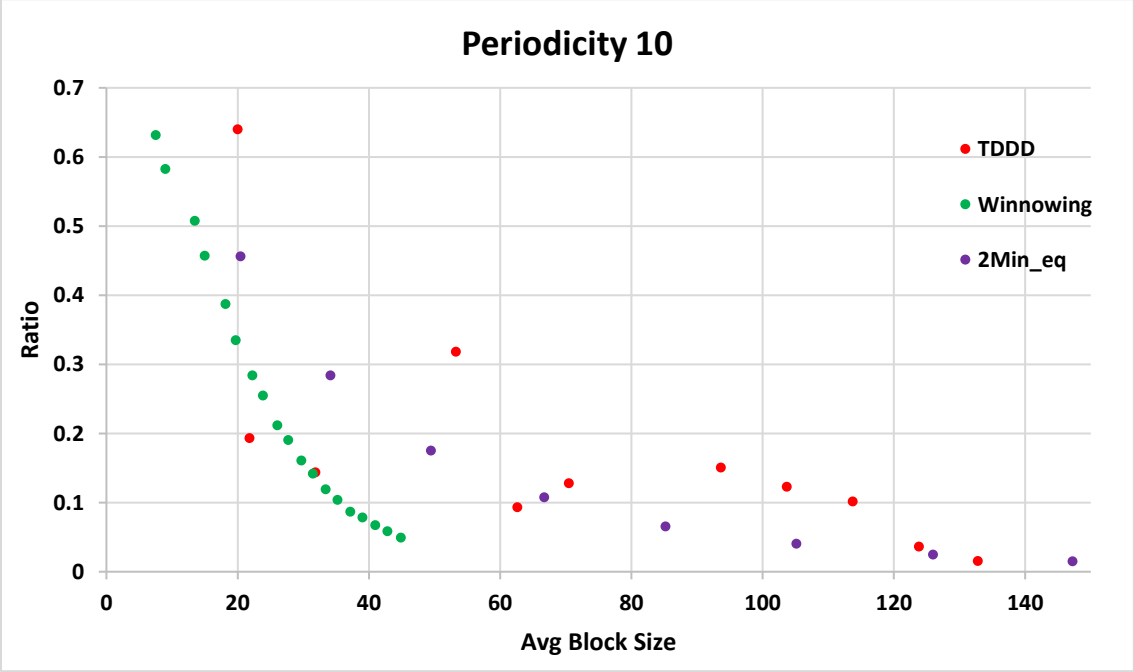


(b) Experimental results for a particular file from the periodic dataset.

Figure 24: Experimental results for periodic data with periodicity of 30

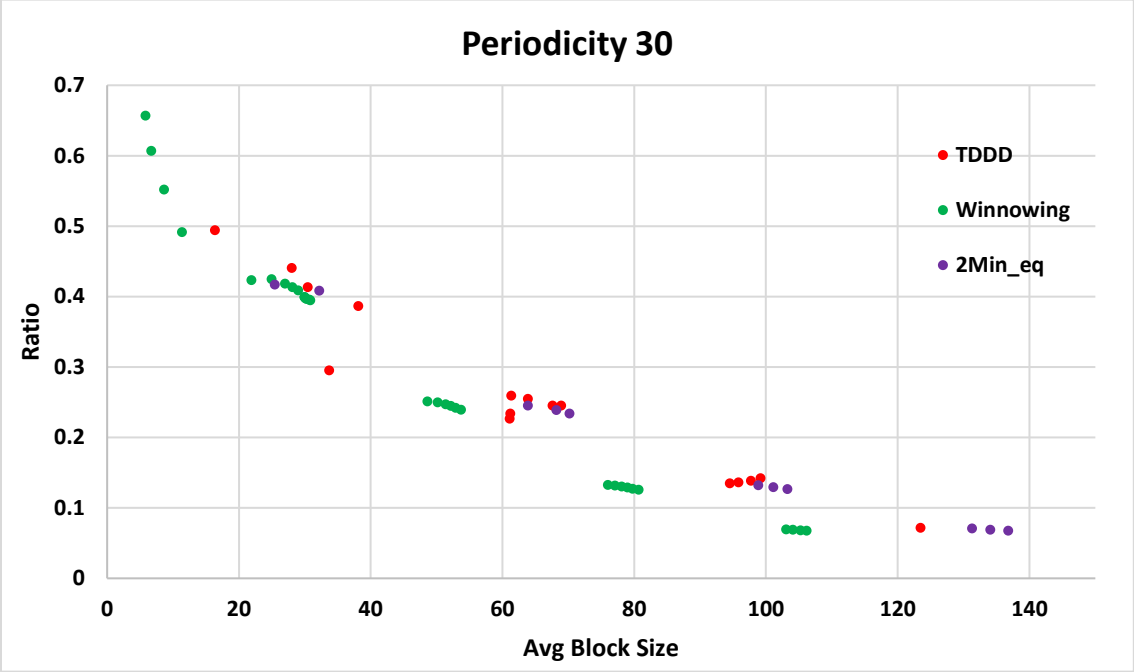


(a) Experimental results for a particular file from the periodic dataset.

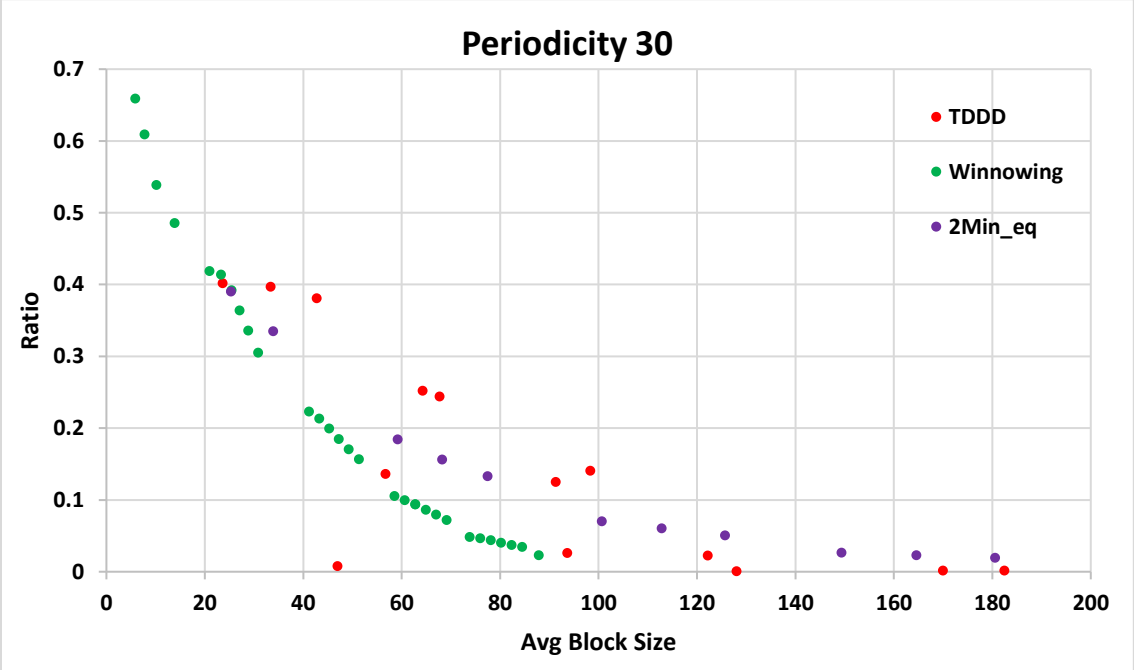


(b) Experimental results for a particular file from the periodic dataset.

Figure 25: Experimental results for periodic data with periodicity of 10. Here we are setting 2Min's condition to less than or equal to.



(a) Experimental results for a particular file from the periodic dataset.



(b) Experimental results for a particular file from the periodic dataset.

Figure 26: Experimental results for periodic data with periodicity of 30. Here we are setting 2Min's condition to less than or equal to.

6. Modified Versions of 2Min

In this section, we present two modifications of the 2Min algorithm. The shortcoming we aimed to correct for 2Min is its potentially large block sizes. Block sizes much larger than the expected size are undesirable because they have a lower probability of matching. Our approach is inspired by TDDD, and how it improved Karp-Rabin by imposing an upper and lower limit on the chunk size. When the upper limit of the chunk is reached, TDDD either uses the backup cut-point, or forces the current point as a cut-point. One downside to TDDD's approach is that forced chops may not align properly. 2Min already has a lower limit for the chunk sizes, and thus only impose an upper limit for the 2Min algorithm. We always select a cut-point content-dependently, even when the upper limit for the block length is reached. The method by which the cut-point is determined when the upper chunk limit is reached differs between the two algorithms we propose.

6.1 The 2Win Algorithm

The first algorithm we propose is 2Win. The algorithm runs the same way 2Min does, which was previously discussed, but imposes an upper chunk limit. The upper limit prevents larger sized blocks. When the upper limit is reached, we stop the 2Min algorithm and attempt to find a cut-point. The cut-point is determined by selecting the smallest hash value in the interval (*previous*, *current*); where *previous* is the hash value after the previous cut-point, and *current* is the current hash value that was under consideration. If there are ties, we select the rightmost minimum hash value (closest to

current). We then start running 2Min after this point. Figure 27 shows an example of running 2Win on the data range with $b = 3$ and the maximum threshold set to $2*b$.

6.3 Backup2Min

The second algorithm we propose is called Backup2Min. When the upper limit is reached, just as the name suggests, we use a backup point and declare that as a cut-point. The backup point is declared by running a more relaxed version of 2Min. A hash value is defined to be a cut-point in 2Min if that the hash value is less than all the hash values up to b steps to the left and within b steps to the right of the cut-point. We define a hash value to be a backup cut-point if the hash is the second smallest in 2Min's two-sided window (2Min would require this to be the smallest). We only store and use the rightmost backup hash. When we reach the upper chunk limit, we see if we have a backup hash and declare that as a cut-point. If we do not have a backup hash, we continue running 2Min until we find a cut-point by either the 2Min criteria or by our backup criteria. Figure 28 shows an example running for the Backup2Min, with $b = 3$ and maximum chunk length set to $2*b$.

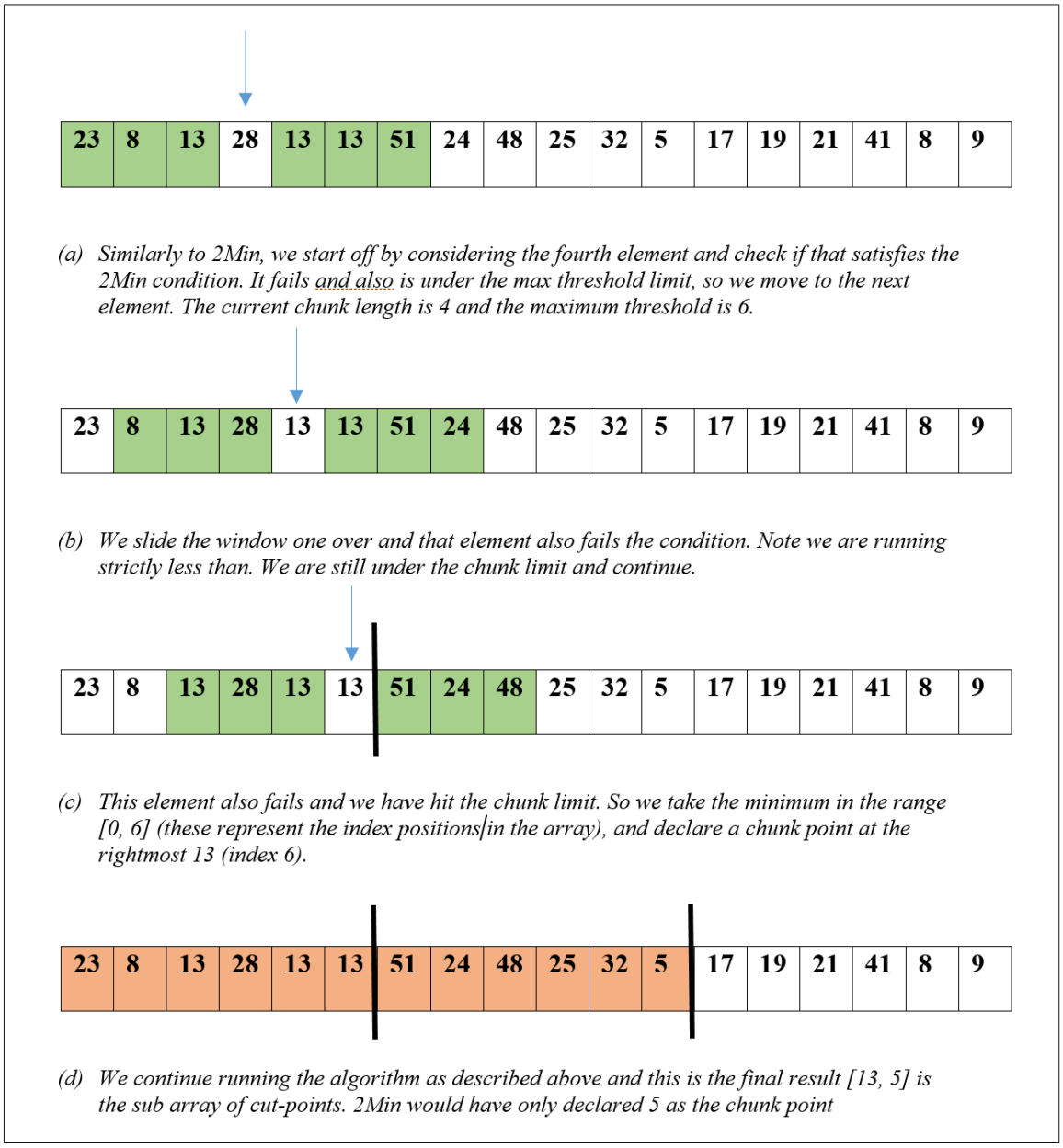


Figure 27: Example of 2Win with b = 3, with max chunk size = 2*b (6)

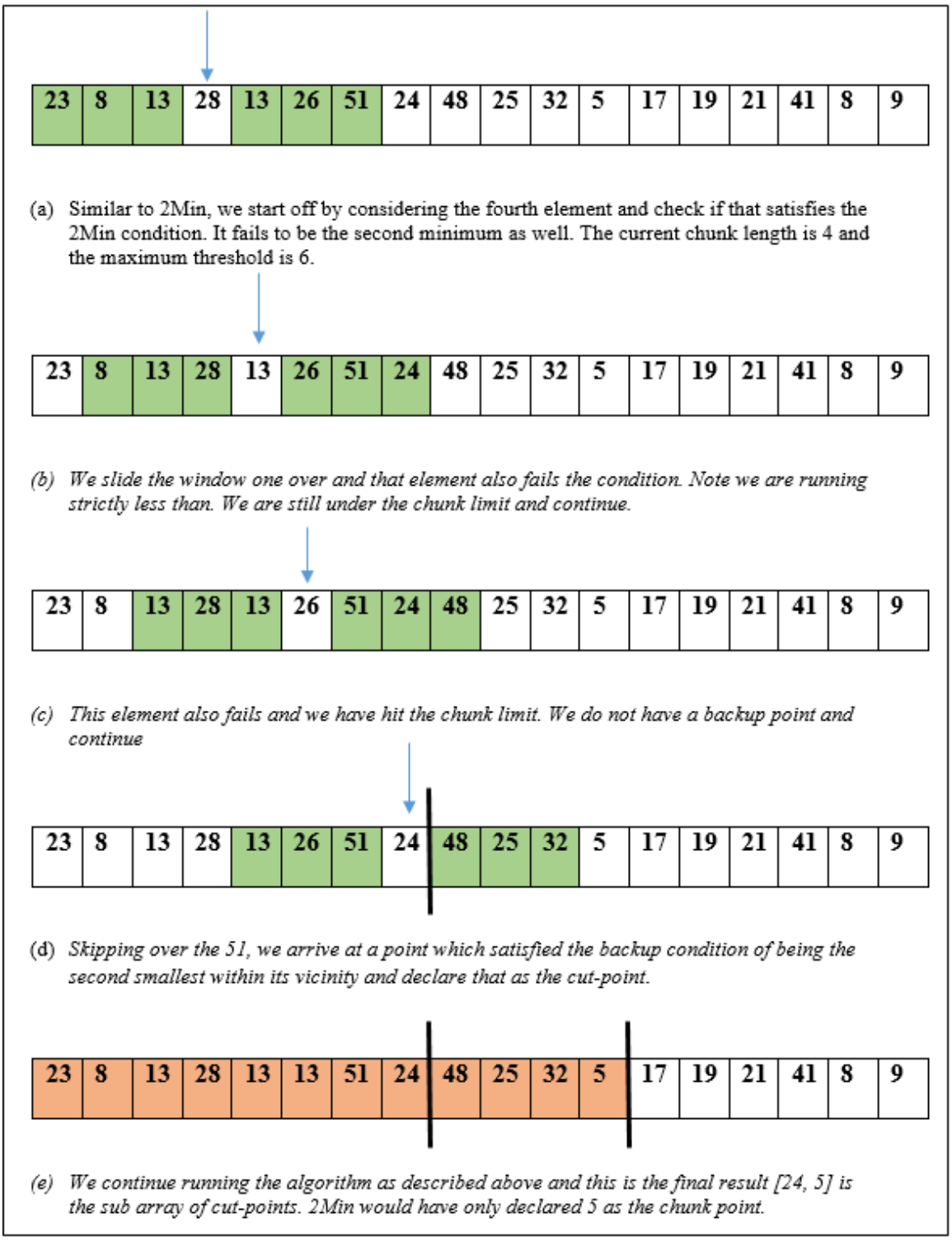


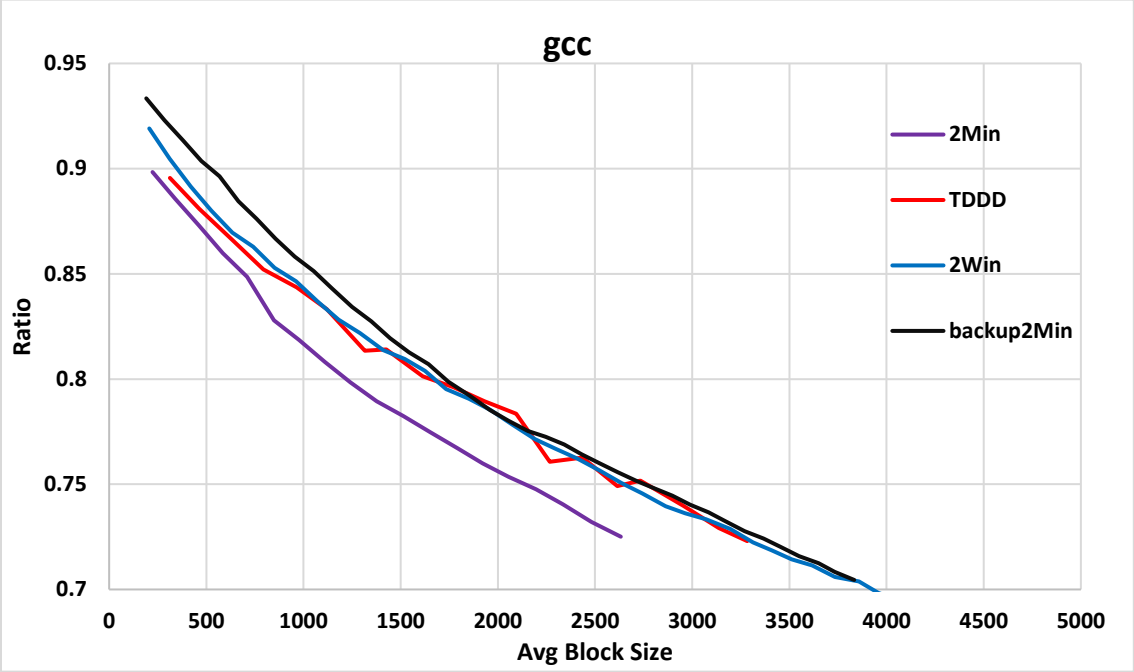
Figure 28: Example of Backup2Min with b = 3, with max chunk size = 2*b (6)

6.4 Results

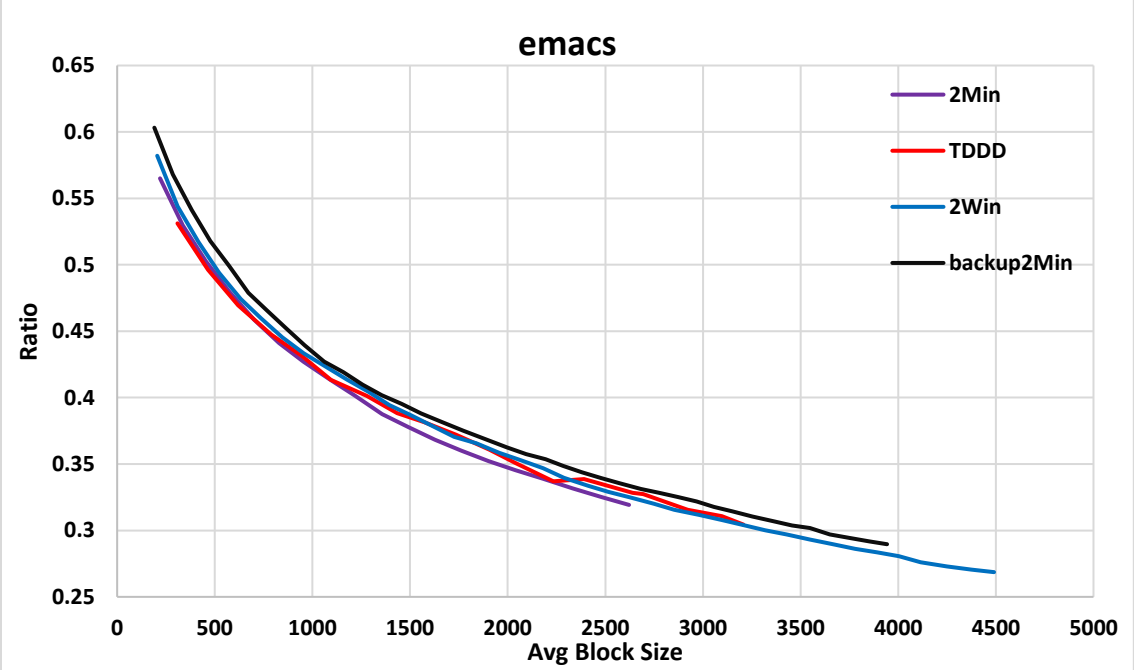
Backup2Min and 2Win were run on the gcc, emacs, morph, Internet Archive and periodic datasets. These datasets were the same as in the previous experiments. Both Backup2Min's and 2Win's max boundary was set to 4*b. The results for gcc and emacs are shown in Figure 29. The results for the Internet Archive dataset are shown in Figures 30, 31, and 32 , and results for periodic data are shown in Figure 33. We only show the periodic data for t set to 10, since the results were similar for other values of t. We do not show the results for the morph dataset because the results were identical to the original 2Min.

The modified versions of 2Min outperform the original in terms of finding more coverage. Looking at Figure 29, we see that the both Backup2Min and 2Win find more coverage in gcc and emacs datasets, just as good as TDDD. On the Internet Archive dataset (Figures 30, 31, and 32), Backup2Min performed better than 2Win and 2Min. Backup2Min outperformed TDDD for medium block sizes, whereas TDDD outperformed Backup2Min for large block sizes.

The modified versions did almost the same as the original 2Min in terms of coverage for the morph dataset. 2Min does not have large chunks for the morph dataset, as seen in Figure 20 and thus, the added chunk limits does not make a difference. However, this is not true in real data, as seen in Figures 29, 30, 31, and 32. The modified versions of 2Min perform better than the original for periodic data, as shown in Figure 33. Unlike 2Min, the modified versions always find document coverage.



(a) *gcc experimental results for the modified versions of 2Min with the original*



(a) emacs experimental results for the modified versions of 2Min with the original

Figure 29: Experimental results for modified 2Min on gcc and emacs

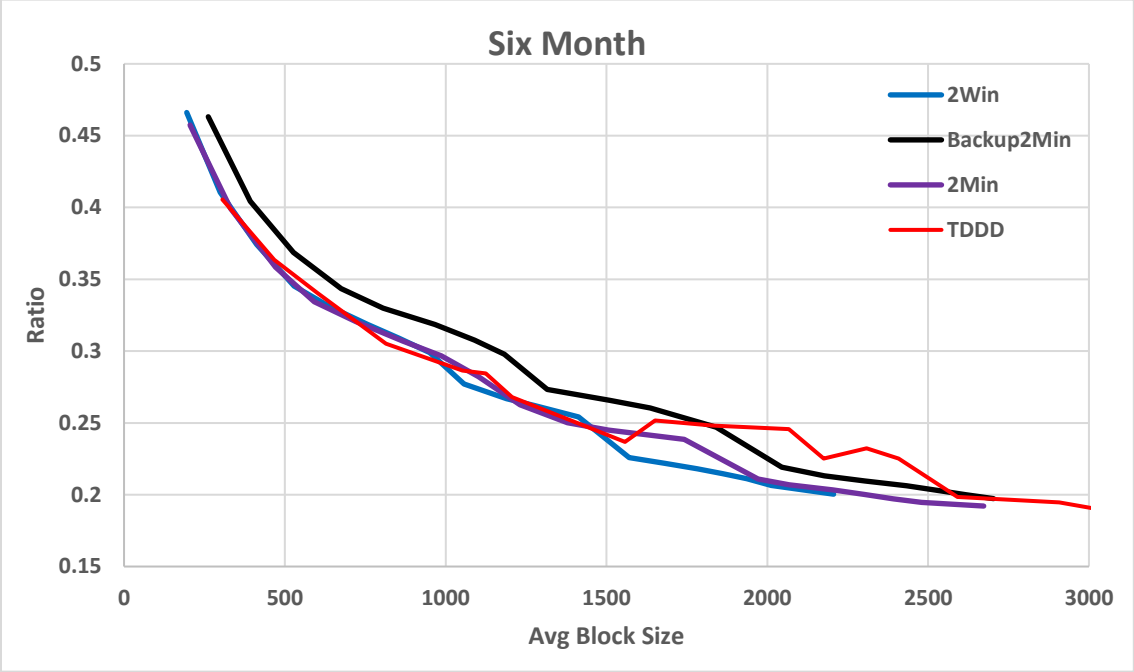


Figure 30: Experimental result for the Internet Archive set. We are comparing the current version with an approximately 6-month-old version.

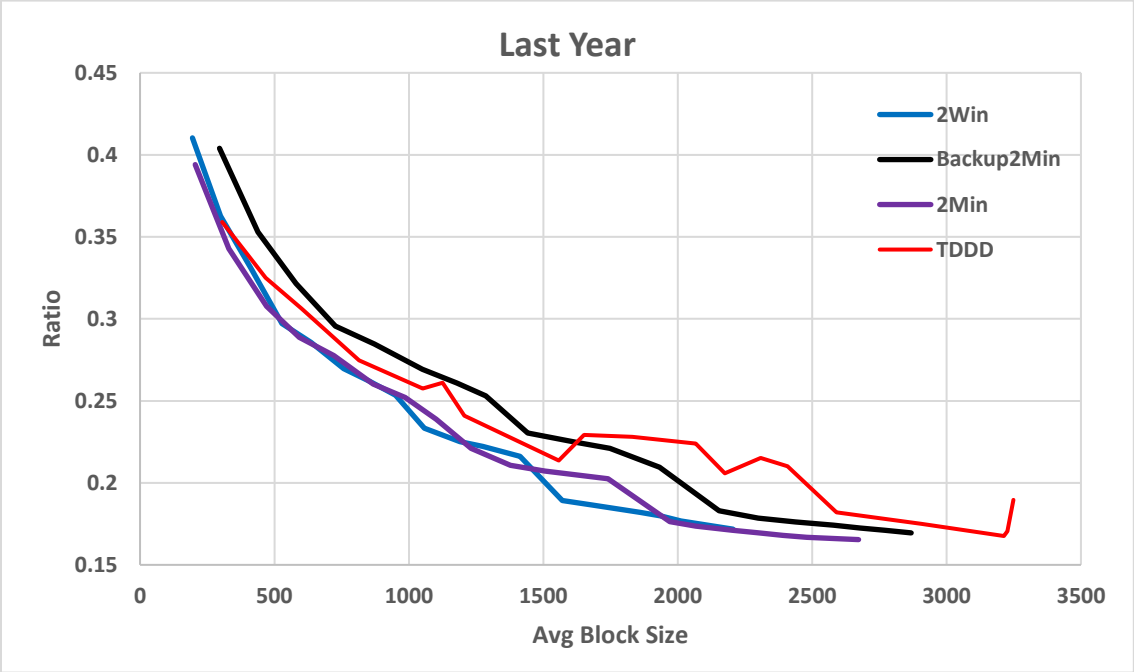


Figure 31: Experimental result for the Internet Archive set. We are comparing the current version with an approximately one-year old version.

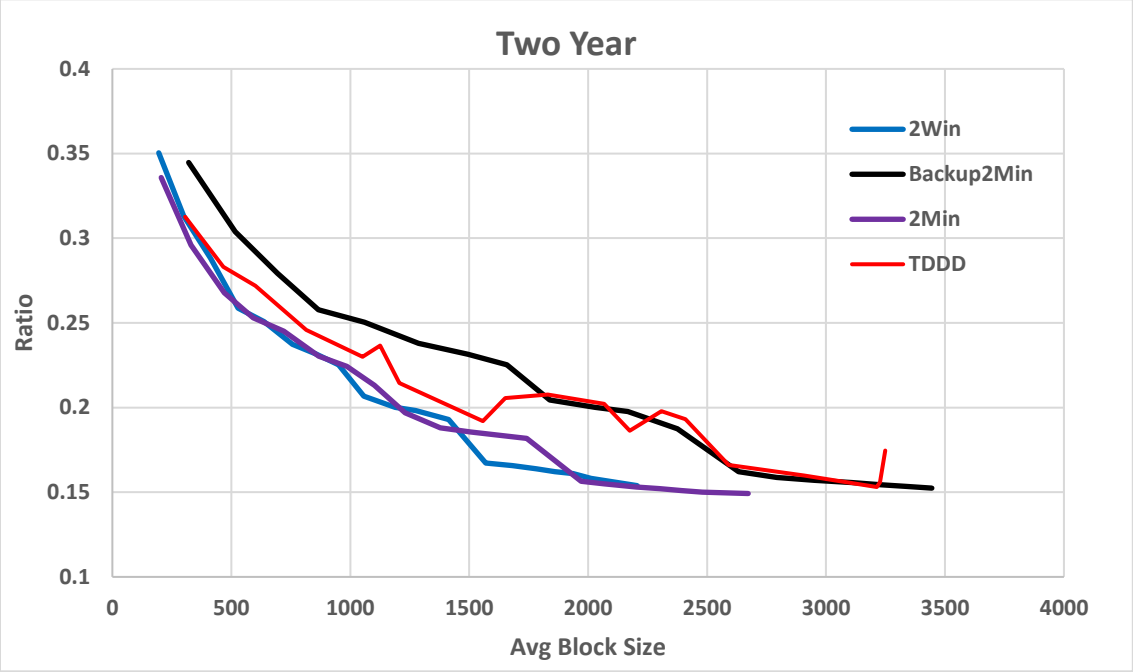


Figure 32: Experimental result for the Internet Archive set. We are comparing the current version with an approximately two-year old version.

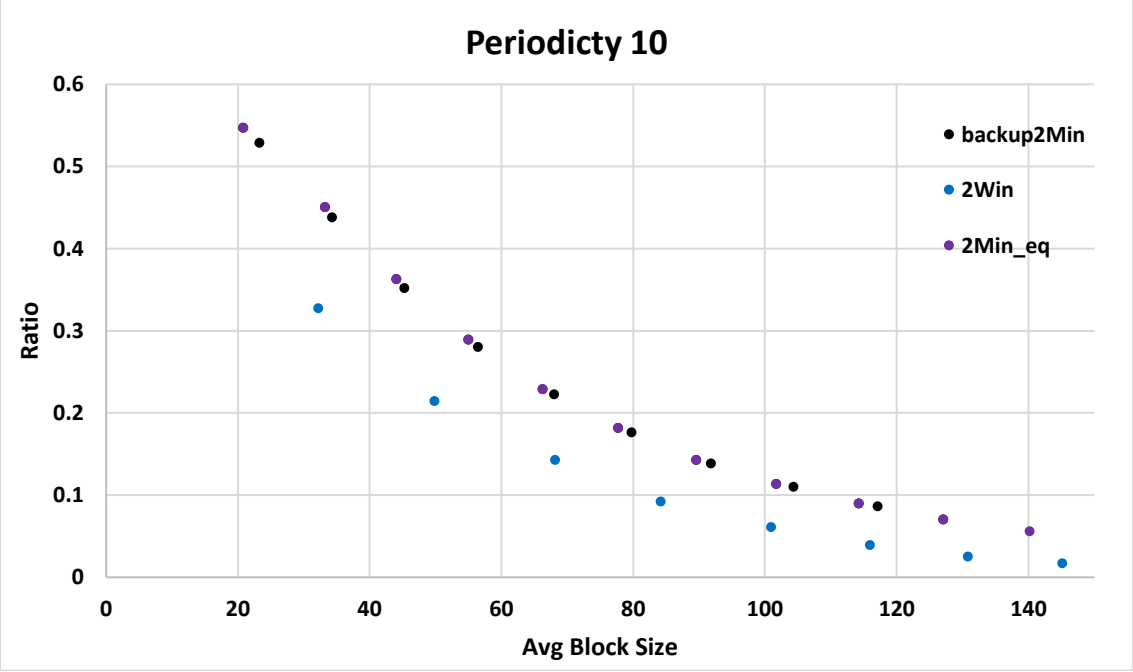


Figure 33: Experimental result for modified 2Min on periodic data

6.5 Other Work

We also had other ideas that we tried, but did that not perform well in experiments. We discuss a few here so the readers will know which ones we implemented. We do not include the experimental results and simply explain how the algorithms work.

One idea was to modify 2Min such that when the maximum chunk size is reached, we decrease the window size until we find a chunk point that meets 2Min's condition. After every fail, we decrease the boundary window by a constant factor such as .8. Once we find a cut-point, we reset the window size and continue.

Another related approach was that once 2Min reaches it maximum chunk length, we would stop and re-run 2Min in the interval again. Every time we failed to find a chunk point within that interval, we decremented the boundary window by 1, until we eventually found a cut-point.

7. Applications

Content-dependent chunking algorithms have many practical uses. In this section, we describe some of the common uses in applications.

7.1 Saving Network Bandwidth

One application where these chunking algorithms are used is in saving network bandwidth, when transferring files between two machines. [11] introduces the Low-Bandwidth Network File System (LBSF), a network file system which uses the Karp-

Rabin algorithm to transfer only the differences of a common file between a client and a server.

Traditional network file systems, such as AFS, a client has a locally cached version of a remote file, which is on the server, and the server and the client exchange updates of the file by sending the file in entirety. LBSF on the other hand, exploits the similarities between the files and only transfers the differences. Both the client and the server partition its respective files into chunks (using Karp-Rabin), and store the chunk hashes (hashed using SHA-1 hash) and chunk metadata (such has the file and offset) in a database. Instead of sending the file entirely, the client and the server only communicate the portions of the file that are missing from the others' respective chunk database, along with the chunks' metadata.

LBFS uses the Karp-Rabin method to partition the files into chunks. In fact, any chunking algorithm may be used to partition the documents. [15] introduces its own variant of a protocol to transfer files which uses the 2Min algorithm.

Another application where content-based chunking algorithms are used to save network bandwidth is in Value-Based Web Caching [12]. Here, the client has an arbitrary data stream in its cache, such as web pages, and issues an update request on that data stream (over HTTP). The request goes to the proxy, which fetches the content from an origin server. The proxy partitions the content into chunks and stores the chunk signatures. The proxy knows which of the blocks the client already has, and would only send the chunk signatures to the client for blocks that are already present in the client's cache. If a block is not present, the proxy will send the block and the block signature, which the client stores in its local cache. Note, the blocks do not have to be from the

same file. The proxy has an arbitrary cache of everything the client had requested for and would reuse any block, even if the block appears in a different file.

7.2 Plagiarism Detection

Another useful application of chunking algorithms is in plagiarism detection [14]. Online plagiarism tools such as MOSS detect plagiarism by comparing the fingerprints [14] or cut-points of the documents. The cut-points of the file, which is produced by running a chunking algorithm, are stored in a database. Therefore, plagiarized documents can be found by querying the database for matching cut-points.

7.3 Data Deduplication

The final application that we will discuss in this paper is in data deduplication [9, 17]. Data deduplication is the idea of reducing data redundancy within a file system to save storage space. The new file is chopped using a chunking method and a signature is computed for each chunk using a low probability hash function such as MD5. The new file's chunks are compared with the stored chunks and only the new chunks are inserted in and a reference is stored for the redundant chunk.

This application is especially useful for a versioned file set, such as Wikipedia. Many of the Wikipedia articles are minor edits such as fixing grammatical/spelling errors or adding new content. It would be expensive to store all these versions when the majority of the content is the same. Thus, it is less expensive to reuse the redundant data and only

store the new information. Other applications in which these algorithms are used include [2, 4, 5, 18].

8. Conclusion

In this paper, we evaluated and enhanced a couple of content-dependent chunking algorithms, in particular Karp-Rabin, Winnowing, 2Min and TDDD. The chunking algorithms are used to partition the files into chunks, which are used to eliminate redundancy between different files. The files are first hashed using the sliding window approach, and then the content-dependent algorithms are run over the hashed version to determine cut-points. The cut-points are used to partition the file into chunks. The algorithms are referred to as content-dependent due to the way the cut-points are discovered; based on the data itself.

We showed experimental results based on different datasets which include gcc, emacs, morph files, and the Internet Archive. We discovered that TDDD outperformed the other algorithms on gcc, emacs, and Internet Archive dataset. TDDD and 2Min perform equally well on the morph dataset. The algorithms behave unexpectedly on the Internet Archive dataset, and we believe it may be due to the periodicity of the content, such as the HTML tags. We generated periodic data and ran the chunking algorithms on the sets. 2Min performed the worse among the algorithms due to its strict condition for locating a cut-point. Recall that 2Min declared a cut-point if the cut-point was strictly less than all the other values within its vicinity. If we relax that condition to less than or equal, 2Min performs much better than the original on the periodic dataset while still keeping

the coverage ratio the same amongst other datasets which was shown in Section 4. Winnowing was the most robust on periodic data, since it will always find cut-points in a content-dependent way.

One problem we discovered with 2Min was that it may generate large block sizes, much larger than the expected block size. This is undesirable because large block sizes have a lower probability of matching and therefore, the whole block size may be wasted. We corrected this by placing an upper limit on the chunk size, such that when the chunk size limit is reached, we either relax the condition or go back and find a cut-point. Unlike TDDD, which forces a cut-point if no back up hash is found, our algorithms always find a cut-point content-dependently. The two algorithms we propose are 2Win and backup2Min. The two algorithms differ in how a cut-point is determined when the limit is reached. 2Win goes back and finds the rightmost smallest value in the range and declares that as a cut-point. Backup2Min, on the other hand, stores a backup cut-point which is determined by a backup condition which is less strict than the main 2Min condition. Once the maximum chunk limit is reached, we declare a chop using the backup cut-point. If there is no backup value, then we continue until we find a chunk either via the main condition or the backup condition. The modified versions of 2Min outperformed the original in terms of finding coverage, and also did not suffer from periodic data.

Both TDDD and 2Min perform better than the other chunking algorithms in all the different types of datasets we experimented on, with the exception of the periodic dataset. Due to the documents' periodicity, the algorithms behave unexpectedly. If these algorithms are applied to little to none periodic data, TDDD or 2Min may be the optimal choice in the algorithm since these algorithms find more coverage. If the data repeats in

vast amounts, a modified version of 2Min, such as equals2Min, backup2Min or 2Win, can be used. In a nutshell, a modified version of 2Min may be used with all types of data and still provide near optimal coverage.

The block sizes can be chosen according to the desired number of chunks or ratio. Choosing a larger block size will reduce the number of chunk signatures needed to be transferred and stored, but will result in a lower coverage ratio. On the other hand, choosing a smaller block size will increase the number of chunk signatures needed to be transferred and stored, but will result in an increase in the coverage ratio.

Another trait that is interesting amongst the algorithms is the smoothness of the algorithms. For instance, looking at the experimental results of gcc, emacs and Internet Archive set, we see that the curves for both Karp-Rabin and TDDD are not smooth whereas 2Min and Winnowing are. Both Karp-Rabin and TDDD may declare any hash value as a cut-point and therefore there may be many small or large chunks. 2Min has a stricter condition which places a lower bound on the chunk limit and thus will always discover a cut-point content-dependently.

Some more datasets that these experiments would have benefitted from are some real life versioned document sets such as the Wikipedia. The chunking algorithms all use a sliding window to hash the entire document. All of these chunking algorithms use the sliding window approach to hash the entire document. In all our experiments, we had the value of the window set to 12. It would be interesting to determine the effect of the window size on the chunking algorithms performance. We leave this as an open problem.

Finally, we conclude this thesis by pointing the reader toward some additional chunking algorithms, which we encountered while finishing this thesis. The algorithms are introduced in [7, 8, 13, 17], and future work may also compare these approaches.

References

[1] Bjørner, N., Blass, A., & Gurevich, Y. (2010). Content-dependent chunking for differential compression, the local maximum approach. *Journal of Computer and System Sciences*, *76*(3), 154-203.

[2] Eaton, P., Ong, E., & Kubiatowicz, J. (2004). Improving bandwidth efficiency of peer-to-peer storage. *Proceedings. Fourth International Conference on Peer-to-Peer Computing, 2004* (pp. 80-90).

[3] Eshghi, K., & Tang, H. K. (2005). A framework for analyzing and improving content-based chunking algorithms. *Hewlett-Packard Labs Technical Report TR*, *30*, 2005.

[4] He, J., & Suel, T. (2011). Faster temporal range queries over versioned text. In *Proceedings of the 34th international ACM SIGIR Conference on Research and Development in Information Retrieval* (pp. 565-574).

[5] He, J., & Suel, T. (2012). Optimizing positional index structures for versioned document collections. In *Proceedings of the 35th international ACM SIGIR Conference on Research and Development in Information Retrieval* (pp. 245-254).

[6] Karp, R. M., & Rabin, M. O. (1987). Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, *31*(2), 249-260.

[7] Kruus, E., Ungureanu, C., & Dubnicki, C. (2010). Bimodal Content Defined Chunking for Backup Streams. In *Proc. 8th USENIX Conference on File and Storage Technologies* (pp. 239-252).

[8] Lu, G., Jin, Y., & Du, D. H. (2010). Frequency based chunking for data de-duplication. In *2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems* (pp. 287-296).

[9] Meyer, D. T., & Bolosky, W. J. (2012). A study of practical deduplication. *ACM Transactions on Storage (TOS), 7*(4), 14.

[10] Minsky, Y., Trachtenberg, A., & Zippel, R. (2003). Set reconciliation with nearly optimal communication complexity. *IEEE Transactions on Information Theory, 49*(9), 2213-2218.

[11] Muthitacharoen, A., Chen, B., & Mazieres, D. (2001). A low-bandwidth network file system. In *ACM SIGOPS Operating Systems Review* (Vol. 35, No. 5, pp. 174-187).

[12] Rhea, S. C., Liang, K., & Brewer, E. (2003). Value-based web caching. In *Proceedings of the 12th International Conference on World Wide Web* (pp. 619-628).

[13] Romański, B., Heldt, Ł., Kilian, W., Lichota, K., & Dubnicki, C. (2011, May). Anchor-driven subchunk deduplication. In *Proceedings of the 4th Annual International Conference on Systems and Storage* (p. 16).

[14] Schleimer, S., Wilkerson, D. S., & Aiken, A. (2003). Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data* (pp. 76-85).

[15] Teodosiu, D., Bjorner, N., Gurevich, Y., Manasse, M., & Porkka, J. (2006). Optimizing file replication over limited bandwidth networks using remote differential compression. *Microsoft Research TR-2006-157*.

[16] Venish, A., & Sankar, K. S. (2016). Study of Chunking Algorithm in Data Deduplication. In *Proceedings of the International Conference on Soft Computing Systems*.

[17] Yu, C., Zhang, C., Mao, Y., & Li, F. (2015). Leap-based content defined chunking—theory and implementation. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)* (pp. 1-12).

[18] Zhang, J., & Suel, T. (2007). Efficient search in large textual collections with redundancy. In *Proceedings of the 16th international conference on World Wide Web* (pp. 411-420).