1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0
0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1
0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0
1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0
0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0
0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0
0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1
1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0
0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0
0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0
0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0
0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1
0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0

Improved Pattern Generation for Bloom Filters with Bit Patterns

Optimizing bit patterns for use in blocked Bloom filters

Master's thesis in Computer Science: Algorithms, Languages and Logic

BJÖRN HEDSTRÖM & IVAR JOSEFSSON

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2018

Master's thesis 2018

Improved Pattern Generation for Bloom Filters with Bit Patterns

Optimizing bit patterns for use in blocked Bloom filters

BJÖRN HEDSTRÖM & IVAR JOSEFSSON

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2017

Improved Pattern Generation for Bloom Filters with Bit Patterns
Optimizing bit patterns for use in blocked Bloom filters
Björn Hedström & Ivar Josefsson

Supervisors: Peter Damaschke & Alexander Schliep, Department of Computer Science and Engineering
Examiner: Christos Dimitrakakis, Department of Computer Science and Engineering

Master's Thesis 2018
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Front page picture depicts a pattern design for small filters using the MCRS-algorithm described in this thesis.

Typeset in LaTeX
Gothenburg, Sweden 2018

Improved Pattern Generation for Bloom Filters with Bit Patterns
Optimizing bit patterns for use in blocked Bloom filters
BJÖRN HEDSTRÖM & IVAR JOSEFSSON
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Set-membership is a commonly occurring problem in many areas of computing, from networking to database applications. A popular data structure for this problem is the Bloom filter: a small, hash-based probabilistic structure which guarantees no false negatives, but can result in false positives. Recently they have been used as an important tool in bioinformatics where the data sets are huge, and as a consequence the filters also need to be large. Blocked Bloom filters with bit patterns have been suggested as an alternative to cope with the deteriorated cache- and hash-behaviour in these cases.

It was recently discovered that optimal pattern design for use in these structures is linked to two-stage group testing. There has also been some recent partial results that indicate a certain structure of optimal patterns. This thesis concerns itself with investigating these structural properties to find a better pattern design for use in Blocked Bloom filters with bit patterns. Our main result is a new, deterministic, algorithm for pattern generation used in these structures based on the Chinese Remainder Theorem. The results indicate that this construction improves the false positive rate for all our testing scenarios. As a side-result we also propose a modification to a known combinatorial design used in group testing which significantly reduces the needed number of tests for high number of defectives.

Keywords: Bloom filter, genomics, non-adaptive group testing, disjunct matrix, thesis.

Acknowledgements

First and foremost we sincerely thank our supervisors Peter Damaschke and Alexander Schliep for the large amount of guidance and feedback they have provided. Their vast knowledge of both the theoretical and practical aspects of this work together with their fast response times when questions emerged has been crucial in the completion of the project.

We also would like to thank the kind people at Omegapoint Gothenburg for generously offering us a place to work.

Finally, we wish to thank all the people who have read through our text and given us feedback regarding grammar as well as content. Their assistance has lead to a higher quality thesis and helped us gain further insight into the problems.

Björn Hedström, Gothenburg, June 18, 2018
Ivar Josefsson, Gothenburg, June 18, 2018

Glossary

Antichain A subset without any pairs $y < x$ in a partial order.. viii

Hamming distance The number of indices where the bits differ in two binary vectors.. viii

Hamming weight The number of 1's in a vector.. viii

Level The number of 1's in a vector. Synonym for Hamming weight. viii

Support The support of a discrete probability distribution is the set of all elements with a non-zero probability.. viii

Acronyms

BBFBP Blocked Bloom Filter with Bit Patterns. viii, 3–5, 7, 9–11, 13–15, 17, 20, 23, 27, 34, 37, 38, 42, 43

CPU Central Processing Unit. viii, 10

ECC Error Correcting Code. viii, 18, 37

FPR False Positive Rate. viii, 3, 4, 8–11, 13, 14, 16, 19, 20, 23–28, 30, 31, 33–36, 38, 41–43

LCG Linear Congruential Generators. viii, 24, 31

Contents

List of Figures

1
Introduction

To see if an item is a member of a set is one of the most commonly occurring problems in computer science. The problem is deceptively simple: Given an item and a collection of several items, does the given collection contain the provided item? Examples of this problem can be found all around in many different types of applications from web stores to computationally heavy research software. Consider a service that requires authentication through login. Then for the user to access the service it is required that the user is registered in the user database, which the software needs to check before granting the user access. Avoiding the database query for mistyped usernames would speed up this.

More specifically set-membership can be used to speed up database queries. A database query can be quite costly in terms of computational power, and to spend that time to try to fetch an item which does not exist can be an expensive operation. An idea to speed up this process can be to check if the key is actually present in the database before querying it. This can be done by having some structure hosting only the existing keys, which in turn may be so significantly smaller than the entire database that the structure fit in local memory. This allows for a fast "filtering" of queries that would otherwise just result in a negative response, and may in turn speed up the application significantly.

Another application of set-membership is to detect duplicate data in large datasets efficiently. In this case, an application may iterate over a dataset and test each item against a different, initially empty set and insert the item into the set if the query is negative. Hence if the query returns positive, then the item has been encountered previously and can as such be reported as a duplicate.

The above examples illustrate just a small subset of applications of set-membership, but regardless of application they have one thing in common: There must be some data structure for the set. A popular structure for this is a *hash table* which assigns each item to a unique key and allows one to fetch/check the item in constant time by providing the corresponding key. But as the set grows larger, so does the memory requirements for the hash table. When the memory usage reaches a critical point it might be impossible to accommodate the table on a quick-to-access memory such as the cache, which may result in costly operations just to query the table.

However, for specific applications, this problem can be avoided by using a *probabilistic data structure* instead. In these cases, one can often significantly reduce the amount of memory needed by allowing the structure sometimes to claim that an item is a member of a set when it is not. While this is not always applicable, for example, one would never want to claim that a user *might* have administrative rights on one's system, in many cases a small number of false positives can be tolerated to allow the structure to fit on a quicker access memory.

To illustrate this, we yet again consider the example of speeding up database-queries. If the database contains a large number of keys, it might be impossible to accommodate the entire set-structure in the cache memory. In those cases to see if the key exists requires a computationally expensive disc-read, which might only slow the overall application down. However, by allowing a small rate of *false positives*, the application might be able to filter out the vast majority of queries that would not give any result, while still being relatively small. The time wasted searching the database for the non-existing keys that slipped through the filter would then be dwarfed by the speed-gains of successfully filtering the vast majority of non-existing keys.

1.1 Background

One of the most well-known probabilistic data structure for set-membership problems is the Bloom Filter. Named after their author Bloom which devised the structure in 1970, Bloom filters are a fast, small and probabilistic hash-based structure which allows false positives but guarantees no false negatives [1]. There are drawbacks to Bloom filters even when disregarding the false positive possibility, in particular, they require up to 44% more space than the information theoretical minimum and in their basic form does not support deletions. However, when used correctly the disadvantages are overshadowed by the computational gains due to space-efficiency. For example, Georganas et al. [2] use a Bloom filter with a 5% error-rate while using 1-2% of the space required by a hash table in their recent genome assembl software. Uses for Bloom filters exist for such diverse tasks as distributed caching [3], reducing unnecessary disk lookups [4], identifying malicious URLs, word-hyphenation [1] and speeding up database joins [5] among many other examples.

Since its conception in 1970 plenty of research has been devoted to improving the original design. Today there are several variants of Bloom filters with different trade-offs such as compressed Bloom filters and counting Bloom filters [3]. Different fields tend to adapt Bloom filters to suit their requirements specifically, resulting in several specific variants of the typical Bloom filter implementation.

Recently these structures have become an important tool in bioinformatics. For example when counting frequent k-mers it is not unusual to have queries to the filter number in the hundreds of billions [6, 7, 8], and store millions of data points. In these cases, even the comparatively small Bloom filter might become too large to be efficiently handled, which can lead to unwanted behaviour in the form of cache misses. A cache miss may slow down the computation by a factor of a hundred, which in genomic applications can be devastating. Another issue is the time needed to compute the hash functions for such a large amount queries since it can prove to be a bottleneck for the program in settings where the computational needs of the hash functions may be a scarce resource. As such another approach is sought in these kinds of applications.

Blocked Bloom Filter with Bit Patterns (BBFBP)s have been suggested as an alternative to regular Bloom filters in problems where the issue of cache- and hash-behaviour is of importance. In contrast to regular Bloom filters, BBFBPs are cache- and hash-efficient, albeit with some additional storage requirements. The idea is to speed up the queries to the filter by assigning an item to some precomputed bit-pattern stored on the cache, which leads to substantial performance gains. However recently there has been research indicating that the current standard of designing these patterns is not optimal for minimising the rate of false positives [9].

1.2 Problem description

Since BBFBPs are cache-efficient, the most significant performance problem for these structures comes from two sources: overhead from computing the hash functions and receiving a false positive. A false positive can be a costly operation for an application since it will typically lead to unnecessary extra computation. Therefore a reduction in the risk for false positives will improve the running time of the program that uses the filter.

Damaschke and Schliep showed that there is a mathematical connection between optimal bit patterns and non-adaptive group testing [9]. This result links optimal pattern designs to *almost disjunct matrices*, a combinatorial design known from the group testing literature. Group testing is a mathematical field interested in finding a subset of defective items in a larger population in as few tests as possible by testing multiple items as one sample. An important question is thus if there exists a design for matrices that gives lower False Positive Rate (FPR). Since the field of group testing has existed for a lengthy time, with many algorithms for disjunct matrix construction, it is interesting to see if any of these algorithms might provide better performance than the current standard. They also produced a partial result which suggests that some mixture of patterns with specific attributes are optimal [9], which should be investigated in practice. Any evidence for or against this notion of optimal attributes may give insights into a better construction of bit patterns or possible improvements to existing group testing algorithms.

Another issue is the space required to store patterns when using a BBFBP. Since all patterns need to be stored in the cache, there is a hard limit on the number of patterns that can be stored. If there was a way to construct patterns when they are needed this hard limit on the number of patterns could be bypassed. This run-time construction would, in turn, allow for more patterns, which should lower the FPR. Hence an intriguing question is if there exists some algorithm for constructing these patterns at run-time with just a small amount of memory. If this is the case, it might provide an interesting alternative to regular Bloom filters.

The purpose of this thesis is divided into one primary goal with three related sub-goals:

Main goal: Improve the FPR of BBFBPs by constructing better bit-patterns.

1. **Sub-goal:** For finite numbers of patterns, evaluate the performance of existing non-adaptive group testing designs as patterns in a BBFBP.
2. **Sub-goal:** Gather evidence to check whether the conjecture found in [9] regarding the optimal construction of bit patterns holds.
3. **Sub-goal:** Investigate deterministic designs for constructing patterns at run-time for memory-management without severely worsening the performance.

These goals share the same end purpose, a good deterministic design for patterns may provide a way to generate patterns at run-time, which in turn should provide better performance than a regular Bloom filter. We have opted only to consider Bloom filter performance in terms of FPR, primarily since we are looking at the generation of patterns which has a minimal impact on running time, something that will be explained more in-depth in coming chapters. While this thesis seeks to improve the construction of bit-patterns, it does not strive to construct *optimal* bit-patterns. Hence no comparison with lower bounds will be made throughout this text, only with the current standard of generating bit-patterns.

1.3 Related work

There are other methods than Bloom filters for probabilistic membership. Cuckoo filters are one such structure that while theoretically worse, it has in practice achieved better FPR than Bloom filters [10]. Compared to BBFBPs however, Cuckoo filters are significantly slower in throughput. Hence it is still important to analyse Blocked Bloom filters, and its variant with patterns since improvements in FPR can lead to even better performance comparatively. Another improvement to the classical Bloom filter was made by Pagh et al. [11], which gave a new structure with constant lookup independent of the FPR, together with a lower order term on the space usage. A common trait of this method and Cuckoo filters is the support of deletions from the set, an operation which the original filter does not allow.

A closely theoretical-related work is the recently proposed EGH-filter [12]. This filter is also heavily inspired by combinatorial group testing-designs and can guarantee no false positives for specific, restricted, parameters. This type of filter utilises the deterministic construction of a specific combinatorial design which does not require the filter to save the patterns. However, these are not designed for cache-placement and are therefore not constrained by the cache line-size, which is what drives this study. The EGH-filter also only guarantees no false positives for very few stored items, and for any larger number it has worse performance than a regular Bloom filter. Hence this type of filter is only usable in edge-cases for particular problems, where we seek to find a more general improvement to BBFBPs. It also places substantial constraints on the available universe, something that most genomic applications cannot allow.

1.4 Ethical considerations of this work

There are some ethical considerations that should be made when using Bloom filters. In particular, the false positive possibilities of Bloom filters could if used improperly cause severe problems. Consider a scenario where a probabilistic data structure such as a Bloom filter is used to find criminal citizens. A false positive in this setting would mean that a law-abiding citizen has been flagged as a criminal, which can have fatal consequences if no follow-up check is conducted.

This scenario is relatively improbable, but still shows an extreme of the possible ethical problems. However, there are many additional cases where false positives cannot be tolerated. Even when false positives can be tolerated there is always a balancing act between efficiency versus risk. However, even if the false positive rate seems negligible and the allure of high efficiency seems promising it is essential to stress that these data structures are not a viable alternative when false positives cannot be tolerated. Probabilistic data structures should only be used if the false positive results in an unnecessary search or a similar, harmless, computation.

2
Theory

This chapter goes into detail about some of the theoretical aspects of Bloom filters, BBFBPs and the theory behind non-adaptive group testing. It also reviews the connection between pattern design in BBFBPs and group testing, and list a selection of algorithms for non-adaptive group testing. The chapter finishes with a brief introduction to binary constant-weight codes and how they relate to the field of group testing.

2.1 Preliminaries

Here we define some terms that come up recurrently throughout the text. We assume that all hash-functions are perfect and uniform. The **operator** $\leq$ on vectors denotes *containment* of an vector inside another vector, i.e. $x \leq y$ states that each index containing a 1 in x also contains a 1 in y. The **union operator** $\cup$ is used to denote bitwise-or of two vectors of equal length. **Hamming weight** refers to the number of 1's in a vector, while **Hamming distance** refers to the number of different bits when comparing two vectors. We will sometimes refer to a pattern's Hamming weight as its **level**, and throughout the text pattern and vector will be used interchangeably. A vector always refers to a binary vector in this context. When a pattern is **inserted** into a block, it means that the block is set to be the union of the current block and the pattern. The **support** of a discrete probability distribution denotes the set of all elements with a non-zero probability. These concepts can also be found in the glossary. Together with this, some variable shorthand symbols are used for mathematical purposes. These will be described in greater detail in the coming sections.

- m = The number of bits in a block or pattern.
- n = The number of patterns.
- d = The number of inserted items in the filter.
- b = The number of blocks in the filter.
- k = The number of hash functions in the filter.

When discussing matrices for non-adaptive group testing we will refer to their dimensions as m x n instead of the t x n notation that is more commonly found in the group testing literature.

2.2 Bloom filters

2.2.1 Standard Bloom filters

A regular Bloom filter contains a set of m bits and a set of k hash functions. Whenever an item is added to the set, k hashes in the range $\{1,..,m\}$ are computed and the corresponding bits set to 1. When testing if an item belongs to the set the same k hashes are computed and instead it is checked if the bit at every index is 1. If so the query will be positive, if any of the bits are 0 the query will instead be negative. This implementation makes the filter probabilistic since it is possible for the filter to compute every hash for an item not in the set to indices already set to 1, thus generating a false positive. For an example of this, see Figure 2.1. It has long been a known fact that to minimise the FPR in Bloom filters the amount of 0's and 1's in the filter should be about equal, which is achieved by taking $k = \frac{m}{d} * \log 2$ [3]. Since many items can map to the same index, it is not possible to remove an item from the filter, because a Bloom filter guarantees no false negatives. However, extensions such as counting Bloom filters solves this problem at the expense of additional space.

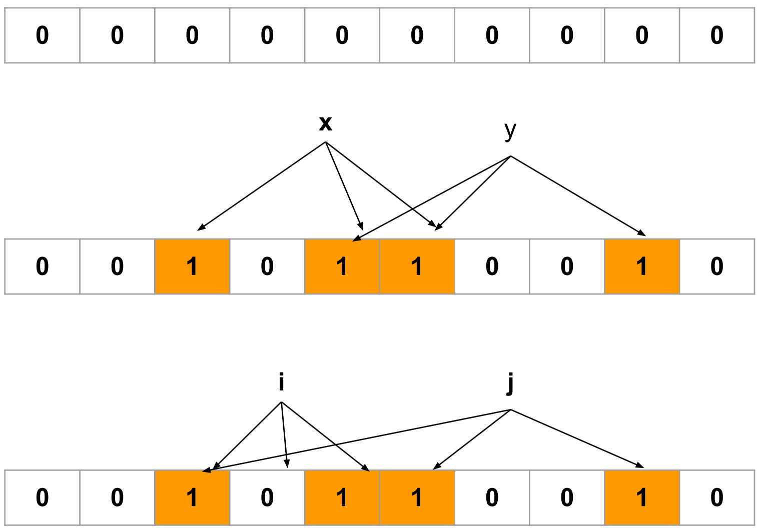


Figure 2.1: An example bloom filter with $m = 10$. Here item x and y are inserted into the filter by $k = 3$ hash-functions, flipping each bit in the mapped indices to 1. Item i and j is then tested. Since one of the hashes for item i points to an index with a 0 this item cannot be a part of the set. However since the hashes for item j all points towards indexes with value 1 the test returns positive, thus giving a false positive. This example also illustrates why items cannot be removed since both x and y have two indices in common. Removing one would thus result in a false negative when testing the other for membership.

The standard Bloom filter implementation quickly becomes unsuitable when storing a large number of items while trying to maintain a low FPR, despite its relatively small size. This unsuitability is due to a combination of two factors, the first of which is the hash overhead, since up to k hashes may need to be calculated for each query. The second, maybe more pressing issue, is the cache behaviour when the filter cannot fit on the cache. In this case, each hash function will result in a so-called cache miss, a term used to refer to an action the program takes which requires it to access some other storage than the significantly faster cache memory. These cache misses comes from the application having to access a bit not currently on the cache. The impact of cache misses for negative queries is not as severe, since the expected number of hashes computed before rejection is 2 given that the filter consists of equal amounts of 1's and 0's [13]. On the other hand for positive queries, there will be k cache misses. For usage cases which expects a large number of positive queries, there are adapted variants of Bloom filters that avoid this problem. Precisely at what point a Bloom filter becomes suboptimal compared to more cache-efficient varieties depends on the parameters of the system with the expected percentage of positive queries.

2.2.2 Blocked Bloom filters with bit patterns

BBFBP is a variant of Bloom filters that can be used when the issue of cache- and hash-behaviour is of importance. In this implementation, the filter is divided into a number of blocks b and provided with a table consisting of n patterns. The length of a block and a pattern is the same, namely m bits, which should be exactly small enough to fit on either an L1 or L2 cache line. This implementation allows the CPU to store both patterns and blocks on the cache, which is crucial for the filter to be quick enough. A pattern is a precomputed vector representing where the hash functions would have inserted a 1 into the filter. The suggested way to construct this pattern-table is by constructing n vectors of length m with exact Hamming weight k by random sampling [13]. By computing this beforehand, we can achieve significant speed improvements which will be described later. However, the number of available cache lines limits the number of patterns n and the number of blocks b. This limitation means that the number of blocks (b), the number of patterns (n) and the number of bits in a block (m) is restricted by the hardware on which the filter is run.

The implementation only uses two hashes that are calculated for every item that is either added or queried. The first hash determines what block the item belongs to and the second hash determines to which pattern an item corresponds. This hash-scheme results in significantly fewer hash functions than a regular Bloom filter in most cases, making the structure hash-efficient. There exist variations of this scheme called multiblocking BBFBPs, however these structures require a larger amount of hashes and is thus out of scope for this study. Adding an item to the filter means that the pattern is added to the block, a query instead consists of checking if the pattern is contained within the block. Since a block and a pattern consists of the same amount of bits, **OR** & **AND** instructions can perform each of these operations.

SIMD instructions can be used to speed up the comparison of a pattern against a block or adding a pattern to a block. These instructions are generally available for any generic Central Processing Unit (CPU) that is relatively modern even without hand-written assembly code. SIMD instructions can perform boolean operators on whole vectors in just a few CPU cycles, which gives significant speed gains compared to the naive method of iteratively comparing each index. This quick comparison between vectors is significantly faster than comparing k separate indices, which makes BBFBPs a significantly faster data structure overall compared to a regular Bloom filter. Since everything is designed to fit on the cache the structure is also cache-efficient since memory outside of the cache will not need to be accessed. While this structure is fast, it does suffer in terms of space usage compared to regular Bloom filters with the need to store the patterns. While in general the memory required to store the patterns is dwarfed by the actual filter this is still a price to pay to achieve the additional speed compared to computing several hash-functions. However, the introduction of patterns also results in a higher FPR.

When querying a BBFBP a false positive can occur for two different reasons: either a so called *collision*- or a *containment*-error.

A **collision**-error occurs when the two calculated hashes are exactly equivalent to another item that was previously added to the filter. In the specific case of there being only one block, $b = 1$, the probability of a collision error is $1 - (1 - (\frac{1}{n}))^d$ [13]. When there are multiple blocks, one can use collision resolution mechanisms to get roughly equal amounts of items in each block. This results in the expected FPR for a collision being $1 - (1 - (\frac{1}{n}))^{d/b}$. This collision probability is not an issue for regular Bloom filters and will thus increase the FPR for the structure compared to the standard approach.

There is a minimal amount of patterns needed for the BBFBP to be useful. For any $n \leq m$, we get a collision probability bounded by $1 - (1 - (\frac{1}{m}))^{d/b}$, which is worse (or equal) to the expected error probability from just a single hash function, which is vastly superior in terms of space usage. Hence we assume that we have $n > m$. If this is not the case and we insist on using precomputed patterns, then the n first columns of the $m \times m$-identity matrix are always optimal.

A **containment**-error, on the other hand, is the probability that the selected pattern has not been added to the block, but the filter still reported a false positive. This false positive comes from other items being added to the filter, and the resulting block has indices set to 1 at each index that the selected pattern has. For an example of this, see Figure 2.2. Mathematically this can be described as the selected pattern being y, and all previously added patterns added to the block is $x_1..x_d$. The condition for a containment error to occur would then be $y \leq x_1 \cup ... \cup x_d, y \neq x_1..x_d$. This probability is, as opposed to the collision-probability, highly dependent on the actual construction of the patterns.

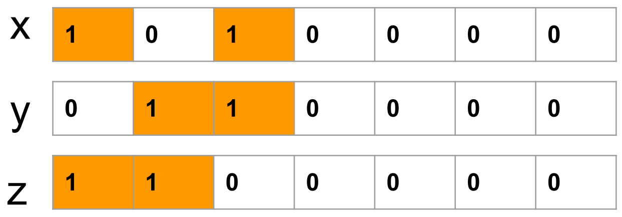


Figure 2.2: Above is an example of a possible containment error in the pattern design. Consider the case where **x** and **y** have been inserted into the filter. Should any new item hash to pattern **z** then this will result in a false positive since $z \leq x \cup y$. Also, note that this is not the only possible error in this design since both $x \leq z \cup y$ and $y \leq x \cup z$.

2.2.3 A conjecture regarding neighbouring levels

As mentioned previously, the traditional approach to both regular Bloom filters and BBFBPs is to use k hash functions or construct patterns of exact Hamming weight k with sampling, and choose from these with a uniform distribution. However, Damaschke and Schliep indicated that this should not necessarily produce optimal FPR [9]. Rather, they provide the following result:

Consider probability distributions on the set of bit vectors of length n. We say that a distribution **dominates** another distribution if, when the vectors are used as patterns in a BBFBP, the first one results in lower or equal FPR. Damaschke and Schliep show that the following Theorem holds:

Theorem 2.1. *For every m, every probability distribution on the m-bit vectors is dominated by some probability distribution whose support is a weak antichain.*

A weak antichain refers to a distribution where we only allow patterns in a $y \leq x$ relation as long as they differ in only one bit. While not conclusive, this indicates that the support which obtains optimal FPR has a combination of patterns on two neighbouring levels, k and $k + 1$ [9].

2.3 Group testing

Given the connection found between pattern design and non-adaptive group testing, it is necessary to give a small introduction to the field of group testing. The section begins by outlining the theory of non-adaptive group testing before presenting the relevance of this field to the design of optimal patterns in a BBFBP.

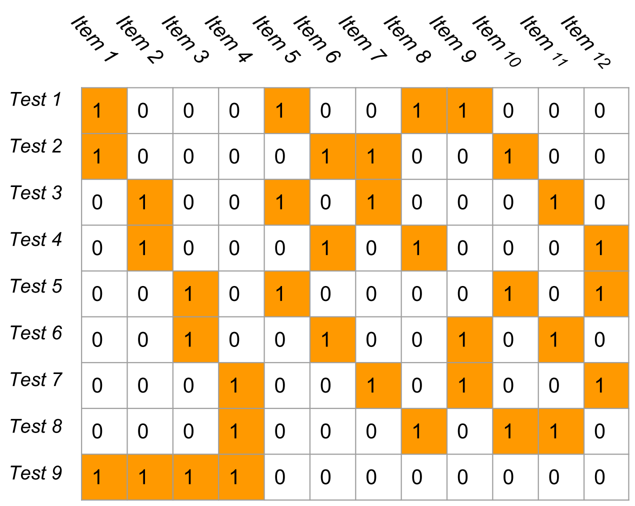
Group testing has the goal of finding the *defective* individuals in a population while doing fewer tests than the total population count. This search is performed by grouping individuals together and testing multiple individuals using the same test.

The problem was initially formulated during World War 2 as an attempt to minimise the number of tests required to determine which individuals had syphilis [14]. Today, group testing is used in a multitude of different applications such as DNA sequencing [15] and data compression [16]. The main idea is this: assign the entire population into groups, where an individual can be a member of several different groups. Then each group is tested as a single unit. Thus if any item in the group is defective, then the entire group will be reported as such, and it is a candidate for further study to find the defective element in this much smaller group. Conversely, if the group is reported as non-defective, then it is assured that none of the items in the group is defective. As an example, consider the application of syphilis detection. Instead of testing each soldiers sample independently, a substantial amount of soldiers sample could be mixed into one sample and tested. If the sample produced a negative result, then it is assured that no soldier in that group has syphilis. This relatively simple approach, somewhat similar to binary search, can significantly reduce the number of tests needed to locate defect items in a set or population.

2.3.1 Non-adaptive group testing

Group testing can be either adaptive or non-adaptive, with adaptive group testing running different tests depending on the results of previous tests while non-adaptive group testing, on the other hand, has a fixed set of tests regardless of the outcome of any test. Non-adaptive group testing is highly parallelizable, which enables significant computational gains compared to adaptive group testing. From an application standpoint, it also has the advantage that all groups are predetermined which can ease the logistics of the tests. Constructing a non-adaptive group test can be approached by constructing a binary matrix of dimensions $m \times n$ [17]. In this case, each row in the matrix represents a test and each column an item. If there is a 1 in the i-th row and j-th column, then the j-th item participates in the i-th test. The outcome can then be represented as a 0-1 vector of size m, where 1 denotes that the test contains at least one defective, otherwise 0.

The set of defective individuals is denoted D. A requirement for a testing matrix is that if $D_1 \neq D_2$ then the outcome vector should be different, meaning that no combination of tests "shadow" another test (column) in the matrix. If $|D| = d$ then such a matrix where no $y \leq x_1 \cup ... \cup x_d$ (with y distinct from $x_1..x_d$) is called a *d-disjunct* matrix. If such a containment happens with a probability at most ε then the matrix is called $(d, \varepsilon) - disjunct$, or *almost* disjunct. Note that a $(d, 0) - disjunct$ matrix is equivalent to a $d - disjunct$ matrix. This relaxation is sometimes referred to as two-stage group testing, as reported defectives are typically reexamined one more time using some other algorithm to make sure that they are labelled correctly. This form of testing can, in turn, reduce the number of tests needed to just a constant factor above the information-theoretical minimum [18]. If the matrix is $d-$disjunct the procedure of labelling items is easy. Since a defective item will never be a part of a negative test, all items that were part of a negative test are removed as non-defective, and the remaining d items are then labeled as the defective ones. Informally, this can be summarised as follows: A d-disjunct ma-



	Item 1	Item 2	Item 3	Item 4	Item 5	Item 6	Item 7	Item 8	Item 9	Item 10	Item 11	Item 12
Test 1	1	0	0	0	1	0	0	1	1	0	0	0
Test 2	1	0	0	0	0	1	1	0	0	1	0	0
Test 3	0	1	0	0	1	0	1	0	0	0	1	0
Test 4	0	1	0	0	0	1	0	1	0	0	0	1
Test 5	0	0	1	0	1	0	0	0	0	1	0	1
Test 6	0	0	1	0	0	1	0	0	1	0	1	0
Test 7	0	0	0	1	0	0	1	0	1	0	0	1
Test 8	0	0	0	1	0	0	0	1	0	1	1	0
Test 9	1	1	1	1	0	0	0	0	0	0	0	0

Figure 2.3: Pictured is an example binary matrix used for non-adaptive group testing. Since the property $y \nleq x_1 \cup x_2, y \neq x_1, x_2$ holds for all columns the matrix is 2-disjunct, and can as such identify 2 defective items from the 12 items in 9 tests.

trix of dimensions m x n allows us to identify d defect items from a population of n items with m tests. The same goes for a (d, ε)-disjunct matrix, but with probability $1 - \varepsilon$ of success, which can significantly reduce the number of tests needed (m).

A number of different algorithms have been developed to construct these types of non-adaptive tests such as the randomised algorithm COMP [19] and its extension DD [20] for two-stage group testing, and several random and deterministic designs for *d-disjunct*-matrices [17, 18, 21]. However, for almost disjunct matrices deterministic designs are generally lacking.

2.3.2 Relationship to Bloom filters

The similarities between group testing matrices and Bloom filters have been noticed previously in literature [18], but it was recently proven to be connected [9]. Damaschke and Schliep state that any $(d, \varepsilon) - disjunct$ matrix of dimensions $m \times n$ enables a BBFBP with m bits and n patterns, where the FPR is bounded by

$$1 - (1 - 1/n)^d + (1 - 1/n)^d \varepsilon$$

if the distribution over the set of bit patterns is uniform. What this entails is that constructing optimal bit patterns is equivalent to constructing optimal $(d, \varepsilon) - disjunct$ matrices, with ε being directly equivalent to the containment probability. A naive lower bound on the needed number of tests would then entail only a collision probability, which can be achieved when using a d-disjunct matrix, bounded by

$$1 - (1 - 1/n)^d.$$

This property was used in the recently proposed EGH-filter [12]. A more general lower-bound on the FPR for BBFBPs sadly does not exist since group testing bounds typically focus on a fixed n and not m. However, Damaschke and Schliep note that the space requirements for a completely $d - disjunct$ matrix, which again would eliminate the containment probability, quickly become unsuitable as d grows. A workable alternative is $(d, \varepsilon) - disjunct$ matrices since they grow much slower in terms of tests (m).

There are however other types of matrices with even better properties for the use in BBFBP. When a so called $(d, f) - \textit{resolvable}$ matrix's columns are used as patterns, the FPR decreases for growing n [9]. A $(d, f) - \textit{resolvable}$ matrix is a special type of (d, ε)-disjunct matrix where $y \leq x_1 \cup ... \cup x_d$ holds for fewer than f columns y that are different from all x_i. Coincidentally these types of group testing matrices when translated to patterns for BBFBP are exactly equivalent to the standard way of generating patterns [18].

Using a $(d, \varepsilon) - disjunct$ matrix to create the patterns in a BBFBP simultaneously makes it easier to reason about FPR. For example, consider a small example of a filter with $m = 9, n = 12$ and using the patterns from Figure 2.3. Since the pattern design is 2-disjunct, we can store up to two items in the filter without any risk of containment. This comes from the simple fact that no union of two columns (i:e insertion of two items in the filter) can be constructed so that $y \not\leq x_1 \cup x_2, y \neq x_1, x_2$, When testing an item not already inserted we thus only risk *collision* errors.

2.4 Group testing algorithms

Since it is now known that patterns for a BBFBP can be formed from a non-adaptive group testing matrix, it is important to evaluate the standard way of constructing the patterns to other pre-existing algorithms for almost disjunct matrices. This section outlines selected algorithms for constructing these matrices.

While deterministic constructions of (d, ε)-disjunct matrices usually come from codes, algorithms using a random construction typically belongs to one of four categories [17]:

1. **RID** = *Random Incidence Design*, each index is set to 1 with some probability
2. **RkSD** = *Random k-set Design*, each item is a participant in k tests.
3. **RDkSD** = *Random Distinct k-set Design*, same as RkSD but each column is distinct.
4. **RrSD** = *Random r−size Design*, each test contains r items.

The algorithms presented below are selected to give a fair representation of each of these categories. We have opted not to consider matrices from codes since they often require the user to look up existing codes with good length m, which is unsuitable for data structures which should be much more dynamic.

2.4.1 COMP - Combinatorial Orthogonal Matching Pursuit

The COMP-algorithm is one of the most widely used algorithms for generating group testing matrices and a good representative of a RID. This popularity can be attributed to its simple, and completely parallel, implementation. Originally proposed by Chan et al. [19], the algorithm works by setting each index in the matrix to 1 with probability $1/d$. In order to accommodate the block-structure of BBFBPs the algorithm is slightly altered and the probability of setting an index is $\frac{d}{b}$ instead of $\frac{1}{d}$. Hence this modification assumes that there is an equal spread of items over the blocks, which can be achieved by collision resolution mechanisms.

Algorithm 1 Combinatorial Orthogonal Matching Pursuit

```
1: M := $m$ x $n$ matrix
2: set every index in M to 1 with probability $\frac{b}{d}$, 0 otherwise
3: return M
4: **End**
```

The above algorithm is a relatively simple way of generating a $(d, \varepsilon) - disjunct$ matrix. However, it is useful to analyse its properties shortly. The first thing to note is the expected Hamming weight of a vector. Calculating the standard pattern designs Hamming weight k in the traditional way [3] yields:

$$k = \frac{mb}{d} * ln(2) \Rightarrow d = \frac{mb * ln(2)}{k} \tag{2.1}$$

We denote by X the Hamming weight of any pattern created by this algorithm. This gives in a few steps:

$$E[X] = m * \frac{b}{d} = m * \frac{b}{\frac{m*ln(2)}{k}} = \frac{1}{ln(2)} * k \approx 1.44k \tag{2.2}$$

However, we note from [9] that the suggested distribution of patterns yields vectors with a Hamming weight of either k or $k+1$. It is also well known from Bloom filters that k is the optimal number of hashes. Hence this algorithm will lead to a higher weight than the suggested, which will lead to the filter more quickly filling up with 1's. This, in turn, will likely lead to a higher rate of containment compared to the standard approach, which is described in more detail below.

2.4.2 CHE - Cache- Hash- efficient

This design is the standard pattern design suggested in [13], as well as the design for resolvable matrices from [18]. The algorithm is also fairly simple: construct each column of the matrix so that their Hamming weight is exactly k by means of random sampling, i.i.d of the rest of the matrix.

This approach of constructing a matrix also closely resembles the random k-sets design proposed in [22], as well as being an excellent representation of a RkSD. It is also worth to note the similarities between the fixed Hamming weight patterns and non-adaptive schemes from constant weight codes [23]. While this seems promising we note that random vectors with fixed Hamming weight cannot be optimal. This can be shown by a small counterexample with $m = 2$ and $d = 2$ [9]. However since this is the current standard it is included for comparative purposes.

Algorithm 2 Cache- Hash- efficient

```
   M := $m$ x $n$ matrix
2: for $i$ = 1 to $n$ do
       V := vector with $k$ bits set to 1, rest $m - k$ bits 0.
4:     M[:, $i$] = scramble(V)
   return M
6: End
```

2.4.3 RrSD - A random r-size design

The random r-size design constructs a matrix by, as previously mentioned, initialising each row with exact Hamming weight r independently of all other rows. While RrSD is typically only used when the amount of individuals in a given test is limited, it is still interesting to see how this design behaves concerning the FPR. It has also been noted previously in the literature that these designs tend to under-perform compared to RkSD or RDkSD [17]. However, the performance may change when d grows. Sadly, this type of design does not immediately give any insight into how patterns can be constructed at run-time since their construction is row-based and not column-based.

Since we know from [9] that vectors of weight k construct high-performing matrices, we attempt to set a value r (the weight of a row) so that the expected Hamming weight of each column is k. While this seems to be just a RkSD with extra steps, we note that this is just the expected weight, and that vectors can thus end up on two, or more, different levels. As such, this algorithm may help to gain support for/against the conjecture for neighbourhood levels in the vectors.

Let X denote the expected Hamming weight of a column. To get $E[X] = k$, each index must be set to 1 with probability $\frac{k}{m}$. Given the calculation for $k = \frac{mb}{d} * ln(2)$, we get a probability needed for an index as:

$$p = \frac{b * \log 2)}{d} \tag{2.3}$$

Then it is easy to see that to get this probability we need to have

$$r = p * n = \frac{b * n * \log 2)}{d} \tag{2.4}$$

The algorithm thus becomes:

Algorithm 3 Random r-size design

```
   M := $m$ x $n$ matrix
2: r := $\frac{b*n*\log 2}{d}$
   for $i$ = 1 to $m$ do
4:     V := vector with $r$ bits set to 1, rest $n - r$ bits 0.
       M[$i$, :] = scramble(V)
6: return M
   End
```

2.4.4 CRS - Chinese Remainder Sieve

This method of deterministic generation of patterns is taken from the Chinese remainder sieve introduced in [18]. The Chinese Remainder Sieve is a deterministic method for constructing d-disjunct matrices. While it does not perfectly suit our needs since it is designed for small values of d, it might still be useful in special cases for BBFBPs.

The original construction goes as follows and depends on the value d for which we want the algorithm to be d-disjunct: Using a sieve, choose a number of primes (or powers thereof)m p_i, so that their product is above n^d. For each prime p_i chosen in this manner, construct a submatrix of dimensions $p_i \times n$ with all entries set to 0. For each column in each submatrix, set the index to 1 so that the column number modulo the prime is zero. When this is done for each matrix, construct the final matrix by arbitrarily concatenating the submatrices vertically.

A thing to note here is the difference from previous algorithms in that this algorithm constructs a matrix where m is heavily dependent on n, and the assumption is that n is known. Since our case consists of a known m we alter the algorithm somewhat to the following construction:

Algorithm 4 Chinese Remainder Sieve

```
   M := $m$ x $n$ matrix with zero at every index
2: Choose a number of powers of primes $p_i$ so that $\sum_{i=1}^{k} p_i^{e_i} = m$
   for $i$ = 1 to $k$ do
4:     for $x = 0$ to $p_i - 1$ do
           for $j = 0$ to $n - 1$ do
6:             if $x == j \% p_i$ then M[$x + \sum_{l=1}^{i-1} p_l^{e_l}$][j] = 1
   return M
8: End
```

Analyzing the performance of this algorithm is interesting in regards to how d-disjunct matrices behave as patterns for higher values of d than they were designed for. However it is valuable to notice that the algorithm produces vectors with Hamming weight equal to the number of powers of primes chosen, and these are typically selected by some sieve. Hence the number of powers of primes grows quite large for even modestly sized filters, which will result in poor performance since it certainly will be larger than the optimal value k. This is not a new observation on our part: The EGH-filter, which used this algorithm at run-time instead of computing hash-functions, has noticeably deteriorated performance as d grows.

2.5 Binary constant-weight codes

Constant-weight codes and the special instance binary constant-weight codes are an Error Correcting Code (ECC) design known from coding theory. ECC is a way to code data redundantly so that errors can be found, and in some cases also corrected. This redundancy enables it to be transmitted over, for example, noisy channels which might introduce errors in the message. While this thesis does not delve into coding theory, the properties of codes have recently been used for analysing almost disjunct matrices, and thus it is important to introduce the terminology.

A (M, N, h, w)-constant-weight binary code $\mathcal{C}$ is a set of N binary vectors of size M with a pairwise Hamming distance of at least h and Hamming weight w. These types of codes are of particular interest in specific areas of coding theory. Here we use h for the Hamming distance instead of the traditional representation d to avoid confusion with the Bloom filter terminology. A heavily researched topic is how to construct these codes to detect as many errors as possible without adding too much redundancy, see for example [24, 25].

The relationship between binary codes and non-adaptive group testing were discovered quite early where the authors created a d-disjunct matrix from a code $\mathcal{C}$ by using each code-word as a column in a testing matrix [23]. Other examples of these constructions come from Porat et al. [26] by the use of Linear ECC, among others. However just recently explicit deterministic constructions of (d, ε)-disjunct matrices from ECC where presented in [27, 28] where the parameters can be estimated by analysing the properties of the codes. This new-found construction makes constant-weight codes an essential topic for analysing pattern designs which generate vectors with a fixed weight, even if we will not delve into their "real" application here.

3
Method

In this section, we briefly describe the evaluation methods. This content includes how the FPR is calculated for both the neighbourhood levels and analysis of pattern designs. It also contains a brief text of how the real-world data is processed for testing.

3.1 Neighbourhood level analysis

To study the conjecture regarding neighbourhood levels [9], we use a separate method not dependent on any pre-constructed patterns. Instead, patterns are constructed at run-time using the CHE-algorithm using various values for k. The desired k level determines the mix between the neighbourhood levels. If the desired k level is for example 6.1 then there is a 10% chance a pattern with k level of 7 is constructed, otherwise a pattern with k level of 6 is constructed. This means that the average k level used is equal to the desired k level. This testing environment allows us to study the dynamics as the different ratios of levels changes. All hashes performed are simple modulo-hashing schemes since these schemes are typically used in genomic applications where cache-efficient filters are used.

3.2 Experiment on precomputed patterns

To get an estimate of the FPR of a pattern design, for both theoretical and realistic data, the following procedure is performed:

1. Generate patterns using the designated pattern design.
2. Enter d unique items into the filter.
3. Select a fixed number of patterns and test these for entry in the filter. Report all positive matches which are not true positives as false positives.

For each pattern design this procedure is repeated several times to make sure that the reported FPR is an average of multiple sets of patterns. Estimations of the FPR consists of 100.000 tests for each parameter combination to get an accurate number. When experiments are conducted on real data, only data points not already entered into the filter contributes to the FPR. We only insert d *unique* items, but these items can of course hash to the same block and pattern as a previously entered item. All hashes are again simple modulo-hashing schemes, which is representative of hashing-schemes used in genomic applications.

3.2.1 Theoretical testing

For the theoretical testing, a random number generator is used. We seed a Mersenne-twister [29] with the current Unix time which is then used as the source for generating random numbers. Whenever we select a random number it is used as an index for selecting a pattern or selecting a block. Thus, for insertion of a pattern, we generate a random number that is in the range of the number of patterns and insert the corresponding pattern into the filter. The same goes for testing for membership in the filter, a random number is generated to select a block, and then another random number selects a pattern. The block is then checked against the corresponding pattern for membership. If a random number is generated to the exact number an inserted item was generated to we consider this as a collision-error and also report it as a false positive.

3.2.2 DNA-sequence testing

To make sure that the testing with random numbers does not somehow affect our results we also test using real-world data from DNA-sequences. These sequences have to be converted to numbers before the filter can use them. The DNA-sequences are originally represented by long strings consisting of *adenine (A), cytosine (C), guanine (G)* and *thymine (T)*. Every letter in a string is converted into a uniquely corresponding binary number, which when concatenated results in a binary string. This binary string is then converted to a decimal base 10 number and multiplied by a prime before used in the testing system. To ensure that the used data is representative of real applications, we use the first 62 letters of each line from a *.fastq*-file. As such, our data set is equivalent to data used to count frequent 62-mers in genomic applications.

3.3 Runtime generated patterns

An increased number of patterns typically result in a lower FPR since the collision probability decreases. Thus, we are also interested in significantly increasing the number of patterns and improving the FPR. This would be possible if patterns could be generated on demand, using a low amount of auxiliary memory. Informally described: Is there a deterministic approach to pattern generation which can outperform the standard Bloom filter implementation? Here we identify three key aspects:

1. Performance should be *better* or equal to a regular Blocked Bloom filter in terms of FPR
2. Memory usage should be significantly lower than a BBFBP.
3. If the performance is equal to a standard Bloom filter the computation time must be better or equivalent to a standard Bloom filter.

This question has frequently been approached in the literature previously, however it gains new relevance with the knowledge that there is a link between optimal pattern construction and strong non-adaptive matrix design.

4
Results

In this chapter we present the FPR for the different group testing designs when used as patterns in a BBFBP. We also propose a new algorithm for pattern generation based on known properties of high performing pattern designs. This chapter also presents results for different neighbourhood levels as well as two algorithms for run-time construction of patterns.

4.1 Experimental outline

We have no results where runtime of the programs are an important part of the result. As such the exact specs of the systems used is not important, but all results were produced using personal computers.

4.2 MCRS - Modified Chinese Remainder Sieve

Motivated by the knowledge that vectors with a fixed Hamming weight k have a good chance of producing a (d, f)-resolvable matrix with good properties [18], we propose a modification to the Chinese Remainder Sieve Algorithm to produce vectors with an exact Hamming weight k.

The construction goes as follows: we begin by selecting *exactly* k unique primes (or powers thereof) so that their sum is as close as possible, but below, our value m. The value k is set to $m * b/d * ln(2)$ as in the CHE-algorithm. For each power of prime p_x chosen we construct a submatrix with dimensions $p_x \times n$, consisting entirely of zeroes. For each column in such a submatrix, enumerated by their index i, we set the value to 1 at the index $i \ mod \ p_x$. When this procedure is finished for every column, we vertically concatenate the matrices in an arbitrary order. This entire procedure can of course in practice be performed in a single pass for every submatrix as the pseudocode outlines.

Given the algorithms construction, each vector will get a Hamming weight equal to the number of primes chosen. From [18] we know that a matrix constructed using the Chinese Remainder Sieve is d-disjunct up to d for values chosen so that:

$$\prod p_j^{e_j} \geq n^d \tag{4.1}$$

Hence for fixed values of m and n, as in our case, we seek to chose exactly k powers of primes as to maximize $\prod_{j=1}^{k} p_j^{e_j}$ but still be constrained such that $\sum_{j=1}^{k} p_j^{e_j} \leq m$.

If the product of these powers of primes is large, it will allow a higher value for d, which improves the disjunct properties of the matrix. This property gives a matrix construction where for small values d the matrix will only result in collision errors, and for larger d its performance should be equivalent to or better than the resolvable matrices introduced in [18]. We note here the relaxation that the powers of primes do not necessarily need to sum to exactly m, it is enough that the sum is sufficiently close. Using powers of primes it is relatively simple to get a sum close to m without loosing any performance. However, a downside to this algorithm is that it might be impossible to choose k primes for all parameters. Consider for example a filter of size $m = 512$ and $d = 2$. Then $k = 512/2 * ln(2) \approx 177$, which will be impossible to select since the sum of the 19 first primes is already larger than 512. In this case, the algorithm will have to suffice with the largest number of primes it can accommodate without their sum exceeding m.

Algorithm 5 Modified Chinese Remainder Sieve

```
   M := $m \times n$ zero matrixx
2: Choose $k$ powers of primes $p_i$ so that $\sum_{i=1}^{k} p_i^{e_i} \leq m$, where each $p_i$ is unique.
   for $i$ = 1 to $k$ do
4:     for $x = 0$ to $p_i - 1$ do
           for $j = 0$ to $n - 1$ do
6:             if $x == j\%p_i$ then M$[x + \sum_{l=1}^{i-1} p_l^{e_l}]$[j] = 1
   return M
8: End
```

4.3 Run-time pattern construction

We have looked at two different algorithms for runtime generation of patterns. The first is based on an Linear Congruential Generator and the second is based on the Chinese Remainder Sieve. In this chapter we present the exact details of the algorithms and then compare them against eachother as well as a standard Bloom filter.

4.3.1 Linear Congruential Generators

Linear Congruential Generators (LCG) are one of the oldest ways of generating pseudo-random numbers from some seeded value and are therefore a prime target for study, using the generated hashes as seed values. LCGs also has the added advantage that they can construct a column independent from the rest of the matrix, thus potentially resulting in better performance. LCGs are frequently used in a myriad of different contexts due to their simple implementation and seemingly good random spread. Given an item to insert or test we can calculate the pattern in the following manner:
It is important to choose the values for a, c and mod carefully. Badly chosen values can lead to clustering of 1's in the vectors, leading to a higher FPR. We chose to use the parameters with $c = 0$ which is what Park and Miller [30] uses and this

Algorithm 6 LCG-runtime pattern generation

```
   k := $ln(2) * m/d$
2: b := Vector with $m$ bits set to 0.
   seed := hash(item)
4: for $i$ = 1 to $k$ do
       seed = $(a * seed + c)\%mod$
6:     b[seed $\%m$] = 1
   return b
8: End
```

gives us a Lehmer random number generator. These parameters are also used by, for example, the c++ *minst_random*-function and Haskells repa-library.

It should be noted that the general complexity of designing a pattern in this fashion requires $2 * k$ modulo operations, as well as several multiplications. Computing the indices can sadly not be performed in parallel. Thus for this approach to be practical it would be a requirement that the FPR is significantly lower than for a regular Bloom filter.

4.3.2 Chinese remainder methods revisited

Using any of the two algorithms based on the Chinese remainder sieve it is possible to construct independent columns over the universe of n, where the columns belong to a *(d,ε)*-disjunct matrix. In this scenario the value n is not constrained by the cache-size but by the function [18]:

$$\prod p_j^{e_j} \geq n^d \tag{4.2}$$

The deterministic construction that we use is the same as the one recently used in the EGH-filter paper [12] with an exception. Unlike the standard Chinese Remainder sieve where a fixed number of primes is always used, we select exactly k numbers. This is very similar to the MCRS-algorithm that we used for the standard pattern generation.

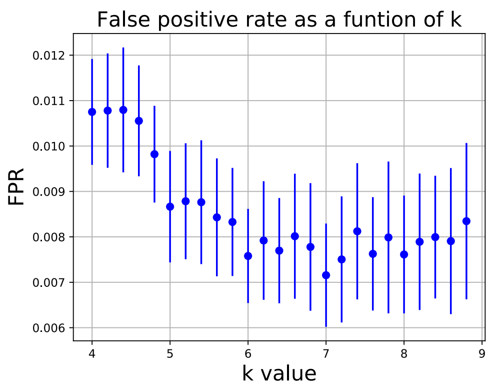
To maximise the value of n we set $d = 1$, knowing that when using resolvable matrices as patterns the FPR decreases for growing n [9]. Now, given a set of k powers of primes, we can construct a vector of Hamming weight k using $k+1$ modulo operations and one hash function. The construction goes as follows:

Algorithm 7 MCRS-runtime pattern generation

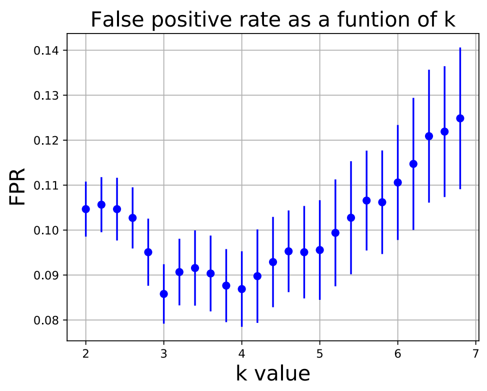
```
   $n = \prod_{i=1}^{k} p_i^{e_i}$
2: b := Vector with $m$ bits set to 0.
   $primes$ := the set of $k$ primes.
4: seed := $hash(item)$ % n
   **for** $i = 1$ to $k$ **do**
6:     index = seed % $primes$[i] + $\sum_{j=1}^{i-1} primes[j]$
       b[index] = 1
8: return b
   **End**
```

4.4 Neighbourhood level experiment

The results of the neighbourhood level experiments can be seen in Figure 4.1. Patterns were randomly generated at the desired k level. From the graphs it can clearly be observed that using any mixture of two different levels only results in higher, or equal, FPR for all problem instances compared to rounding the value k to the nearest integer. Optimal k here refers to $k = m/d * \log 2$.



$m = 512$, $d = 50$ Optimal $k = 7$

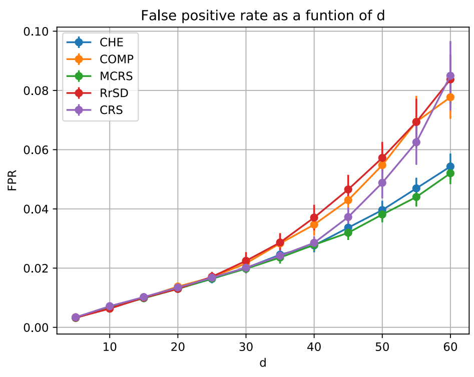


$m = 512, d = 100$ Optimal $k \approx 3.55$

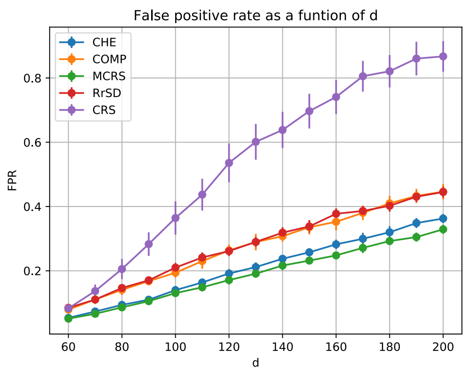
Figure 4.1: FPR for neighbourhood levels of different k with various fixed values d. *tests=1 000.000*, *filter_trials=100*, *m=512*

4.5 FPR comparisons for group testing algorithms

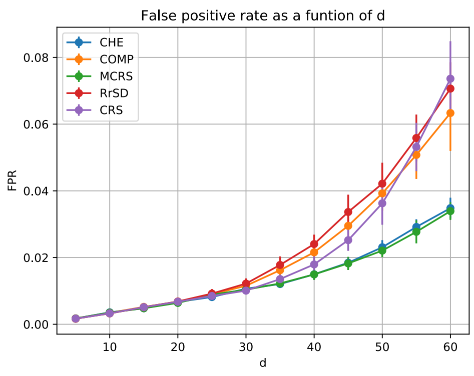
In this section we compare existing group testing designs in terms of the FPR. Since each pattern design is redone multiple times for accuracy, we have opted to preselect the primes needed for the MCRS-algorithm. This simplification is done in both an effort to save valuable computation time and to ensure that the performance of the algorithms is not hindered by a poorly implemented algorithm selecting the primes.



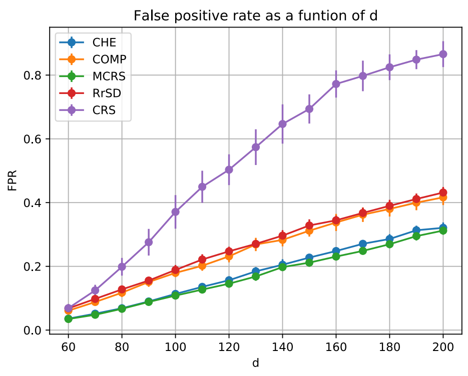
(a) n = 1500



(b) n = 1500



(c) n = 3000



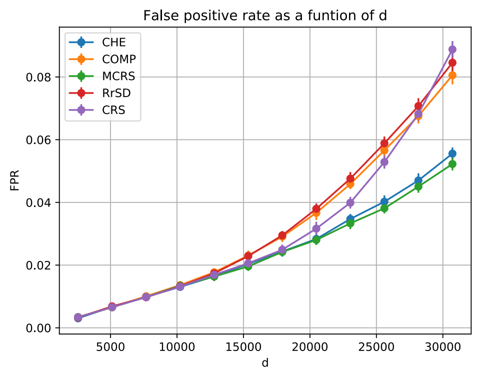
(d) n = 3000

Figure 4.2: FPR for a single block for various n. *m=512*

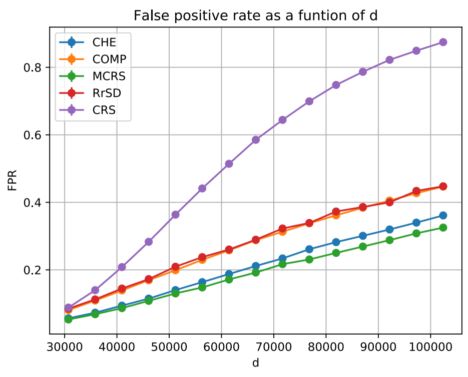
4.5.1 FPR for a single block

Here we present how the different pattern designs behave for a single block. Since a BBFBP with just a single block will obviously have all items inserted in the same block, we can get a good overview for how the pattern designs fare in comparison to each other.

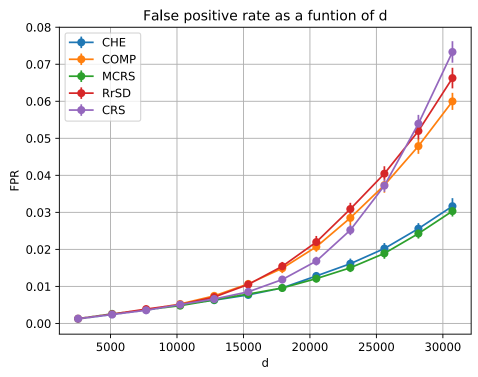
In Figure 4.2 we see the resulting FPR for the designs discussed in chapter 2 for different amounts of patterns. Figure (a) and (c) details how the designs behave for small amounts of inserted items, Figure (b) and (d) details for larger amounts of insertions. The amounts of tested patterns are relevant to practical usecases: 3000 patterns is enough to almost fill the entire L2 cache of a consumer computer. One important thing to note is that the MCRS-algorithm has the lowest FPR out of all the approaches, especially for a small amount of patterns.



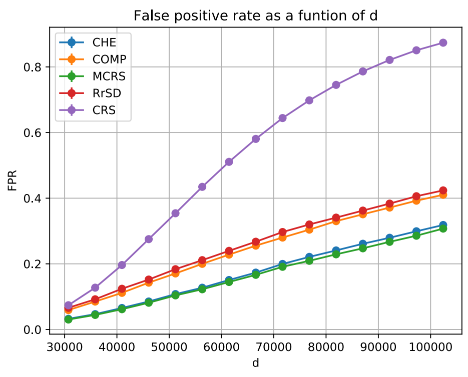
(a) n = 1500



(b) n = 1500



(c) n = 4096



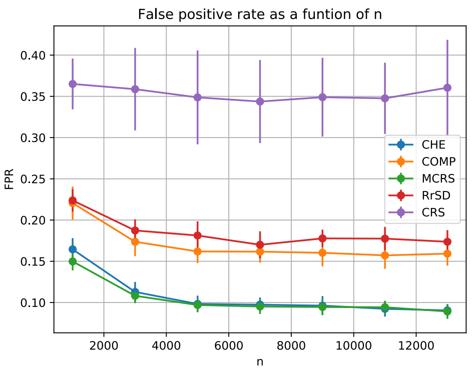
(d) n = 4096

Figure 4.3: FPR for large filters for various n. *m=512, b=512*

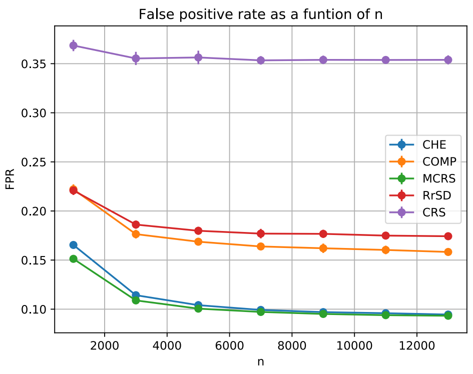
4.5.2 FPR for multiple blocks

In this section we show how the FPR behaves for larger Bloom filters with a higher amount of items inserted. By using multiple blocks we can also see if there is any difference when the pattern designs are used in a more realistic use case scenario. The different pattern generation methods generally behave the same as in the tests using only a single block. The MCRS-algorithm still slightly outperforms the CHE-algorithm, however, the difference between the two methods decreases as n grows.

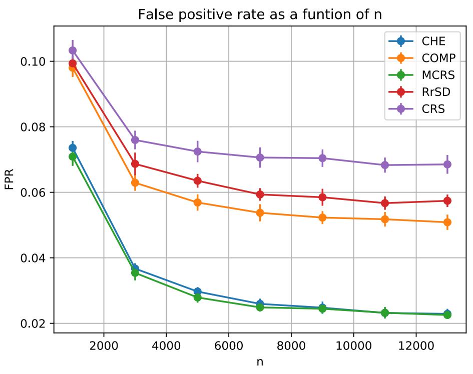
The Figure 4.4 FPR comparison for growing n closely resembles how the collision probability for patterns behaves. The initial drop in FPR by adding more patterns is also the same regardless of the number of insertions into the filter. The graph also indicates that there seems to be a point when adding additional patterns does not decrease the FPR in any significant way.



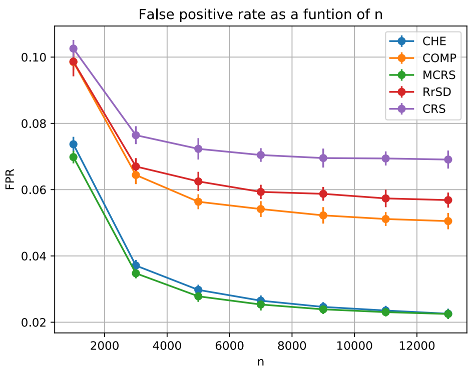
(a) d = 100 b = 1



(b) d = 51200, b = 512



(a) d = 60 b = 1

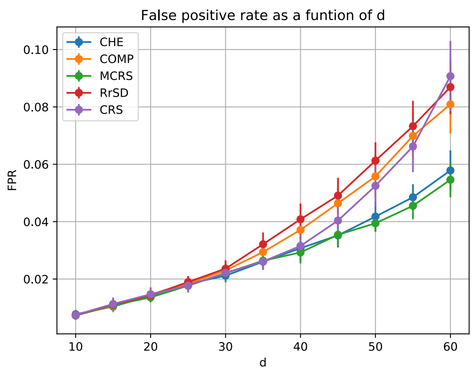


(b) d = 30750, b = 512

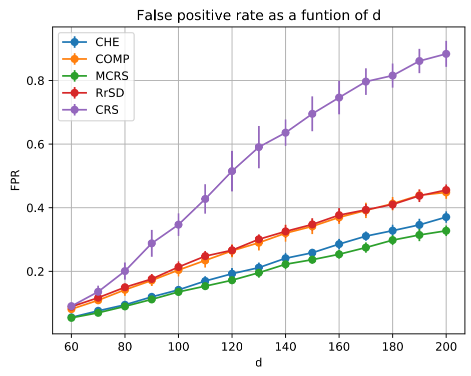
Figure 4.4: FPR for patterns as n increases, $m = 512$

4.5.3 FPR for real-world data

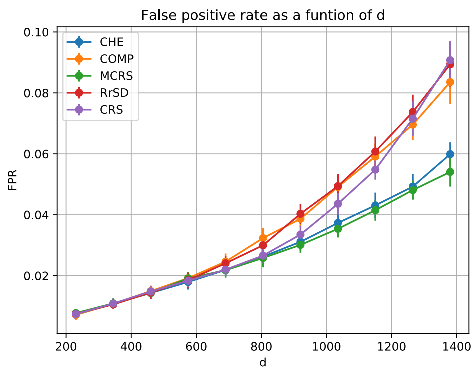
In realistic applications it is not at all certain that items will be uniformly distributed among the blocks as they tend to be from a random value generator. For real world applications, the distribution over the blocks tends to leave some blocks overcrowded while others are relatively empty, thus affecting the FPR [13]. In this section we see how the FPR varies for different group testing algorithms when using DNA-data, for both single and multiple blocks.



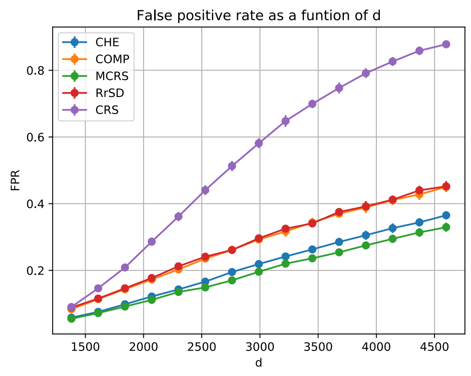
(a) n = 1500, b = 1



(b) n = 1500, b = 1



(c) n = 1500 b = 23



(d) n = 1500, b = 23

Figure 4.5: FPR for patterns of various b, m = 512. The data is taken from the *Babesia Bovis* single cell parasite genome.

The graphs show that the relative performance of the group testing designs are about the same for realistic and theoretical data.

4.6 FPR comparison for run-time generated patterns

The FPR for the LCG- and MCRS-runtime pattern generators can be seen in Figure 4.6. To ensure a better spread over blocks and patterns we set m and b to values that are primes for the LCG scheme. Figure 4.6 is divided into two to more accurately display the differences in FPR for small values of d which would otherwise be dwarfed when d grows.

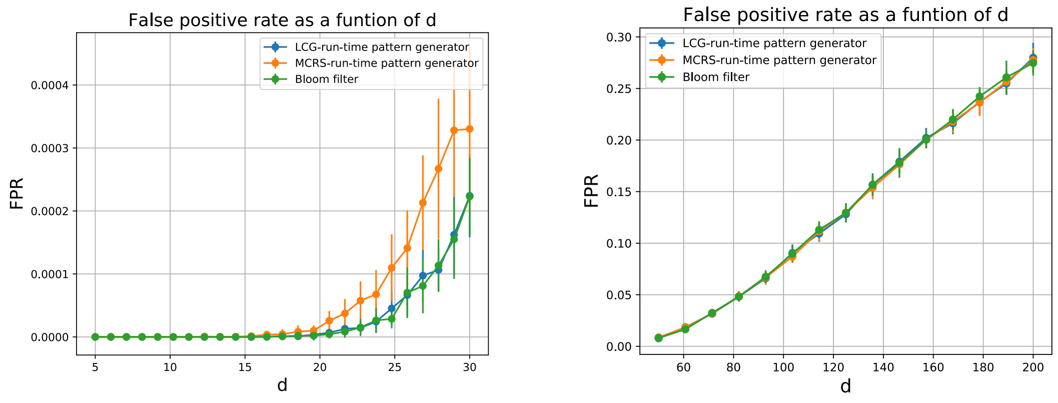


Figure 4.6: Comparison between the theoretical LCG-generated, MCRS-generated and the theoretical Bloom filter as a function of the number of items inserted in the filter. m = 509, b = 1. The values used for the LCG-generator are modulus = 2147483647, multiplier = 48271, increment = 0. The primes are chosen in the same manner as in the MCRS-algorithm. The filters are generated 30 times with 1 000 000 tests each run to get accurate FPR results.

While the LCG-generator is consistently equal in performance to the regular Bloom filter, the MCRS-algorithm is more interesting: It performs equally well for large amounts of insertions, i:e a low bit-per-element ratio, but worse for fewer insertions. This bahvior while be analysed in greater detail in the discussion section. While not shown here, the LCG-generators performance quickly deteriorates if the vector length m is not prime, making the generator unsuitable if correct bitlength is not chosen. However, for realistic application there is often a close enough prime so that this is not an issue. Speed comparisions are not provided here, but will discussed in greater detail in chapter 5.

4.7 Improvement of the CRS-algorithm for non-adaptive group testing

As a side result of our work on linking pattern generation for Bloom filters with group testing we have designed an improvement to the CRS-algorithm for non-adaptive group testing. In this section we introduce a deterministic construction for d-disjunct matrices usable when there is a higher number of defectives, significantly reducing the number of needed tests. The modification goes as follows:

Choose the two smallest primes p_1 and p_2 so that they are both larger than $\sqrt{n}$. Then choose the $d-1$ primes larger than both p_1 and p_2. Use these primes to construct the matrix in the same way as in the Chinese Remainder Sieve algorithm (or the MCRS-algorithm). This modification will produce a $d-$disjunct matrix.

Algorithm 8 new Chinese Remainder Sieve

```
   Choose two unique primes $p_1 > \sqrt{n}$ and $p_2 > \sqrt{n}$.
2: Choose the $d-1$ next primes larger than $p_1$ and $p_2$.
   **for** each prime $p_x$ **do**
4:     $P_x$ := matrix of dimensions $p_i \times n$ with all indices set to 0.
       **for** i = 1 to $n$ **do**
6:         $P_x[i \text{ modulo } p_x][n] := 1$
   Set M to be the vertical concatenation of all matrices.
8: return M
   **End**
```

The number of tests needed will be equal to the sum of the chosen primes, in the same manner as the regular CRS-algorithm. Why this improvement is favorable for higher values of d will be expanded on in the discussion, together with the analysis needed to support the disjunctive claim. [1]

[1]It recently came to our attention that this modification has been proposed previously to guarantee a fixed level of each vector [31]. However, to the best of our knowledge we provide the first analysis of why this algorithm can be favorable in terms of tests and not just in terms of the vectors Hamming weight. We also provide a stronger approximation for the needed number of tests.

5
Discussion

In this chapter the results obtained and presented in the previous chapter are discussed in greater detail. It also contains basic analysis of new algorithms and concludes with short segment regarding open questions posed by the found results.

5.1 Neighbourhood levels

In this section we will discuss our finding regarding the neighbourhood levels and their significance.

As the Figure 4.1 shows, there is no discernible benefit of mixing patterns on different levels. In fact, graphs clearly show that it might be better to choose a suboptimal k level than to use a mix of two k levels. However we note the proof that Damaschke and Schliep produced which shows that the single level k vectors cannot result in optimal FPR [9]. Hence the results in 4.1 are interesting and warrant further investigation.

Firstly, one should note that only using vectors with a fixed level does not break the weak antichain property. For the weak antichain property to break it is required that two vectors are in a $x \leq y$ relation (with $x \neq y$) with the difference being in more than 1 bit. Clearly, this cannot happen for vectors with equal level, and hence the weak antichain property holds. Moreover, while using a mixture of two neighbouring levels also fulfills the weak antichain property, it is not apparent how such a distribution dominates another distribution which also has the support of a weak antichain.

By using a distribution which has the support on two neighbouring levels, what we are mostly doing is increasing the number of unique patterns at the expense of a higher chance of containment error. However, our results indicate that there comes a point where increasing the number of unique patterns does not contribute to lowering the FPR in any significant way. At that point, the main contributor to the FPR is the containment probability. Hence, for a distribution where we do not constrain the amount of patterns, we are unnecessarily adding patterns to decrease the collision probability while in return increasing the containment probability. Since the collision probability is already incredibly small it means that the FPR increase needlessly. We therefore conclude that using a mixture of patterns on two different levels does not seem to improve the FPR when the number of unique vectors is sufficiently large, which it tends to be when it is generated at run-time. However it should be further investigated if there might be improvement in using multiple k levels when the number of patterns, n, is constrained. If the result of such an investigation once again shows that using a single k level is optimal it would be reasonable to conclude that Damaschke's and Schliep's conjecture is not correct.

5.2 Group testing algorithm results

The baseline that each algorithm is compared against is the CHE-algorithm since it is the standard approach of pattern construction. As such, looking at the results in Figure 4.6 there is only one algorithm that has a better FPR than the CHE algorithm: the MCRS-algorithm. For all the other algorithms we have found that the level k of their generated patterns is higher than the optimal k, which has a huge effect on FPR. Because of this we only discuss the MCRS-algorithm further.

For small amounts of patterns such as $n = 1500$, the MCRS-algorithm consistently performs better than the CHE-algorithm. The difference between the two algorithms becomes even greater as the number of inserted items d increases. However, for large amount of patterns, the difference decreases to eventually disappear completely, making the two algorithms equal. This can be seen for designs up to 13000 patterns, which is three times the size of the L2 cache on the simulation hardware. Of course these values are quite dependent on the chosen parameters, but are still indicative of the general performance. We suspect that this is linked to the minimum Hamming distance and in particular to the average Hamming distance of the patterns.

The MCRS-algorithm is only noticeably better than the CHE-algorithm when the bits per element ratio is low and the number of patterns is relatively small, a rather restricted usecase. However apart from the improved FPR, the MCRS-algorithm has other beneficial properties which comes from its determinism.

Firstly, there is a benefit in the construction-phase since no randomness is required. It is well known that true randomness is hard to produce on a computer and it is, more importantly, a limited resource. Hence, if the BBFBP is used it might be hard to guarantee good and uniform pattern generation as well as it being noticeably time consuming. Secondly, the determinism provides a sense of security since the matrix will always be identical for equal parameters. For the CHE-algorithm it is entirely possible to generate a set of patterns that result in a very high FPR compared to the average. When using the MCRS-algorithm the user is given a guarantee that poor performance of the structure is not a result of bad initialization.

Another important point is that it is not necessarily useful to fill the entire L2 cache with patterns. Putze et al. suggests that fetching a pattern from a filled cache-memory might result in a cache-error which is equivalent to a cache-miss in terms of computation slowdown [13]. They proceed to argue that to gain any noticeable speed benefits from using BBFBP the entire L2 cache should not be filled but rather some fewer patterns should be used. In these cases the MCRS-algorithm is a good choice since it, as previously stated, has better performance compared to the CHE-algorithm when n is small.

5.3 Run-time generated patterns

This section is dedicated to a discussion regarding the runtime algorithms and their respective properties. While none of the proposed algorithms achieved better performance than the regular Bloom filter, there are some observations that are of interest.

That the LCG- and MCRS-generators have equal performance is not surprising for large insertions: In some sense they are representative of the CHE- and MCRS-algorithm for infinite amounts of patterns. While the LCG-generator does not guarantee that each vector has a fixed level k, this should be the case for most vectors, thus making it a suitable representation of the CHE-algorithm for comparison against the MCRS-algorithm. As we have already discussed the performance of these two algorithms converges when n grows, and for almost infinite number of patterns the collision probability all but disappears. This leaves only the containment rate for vectors with Hamming weight k and as such the performance should be equivalent to a regular Bloom filter.

While unsurprising, since the LCG-filter did not achieve any better performance it can safely be discarded as a Bloom filter alternative. This comes from the lack of parallelism available in the algorithm, since each new index set to 1 is dependent on the previous being already set. In a classic Bloom filter each index can be computed completely in parallel and as such can be k times faster than the LCG-filter, especially if its hardware-implemented.

The MCRS-generator, however, is of greater interest. Like the classic Bloom filter it can compute each index completely in parallel after the initial hash-function. These indices are also calculated by simple modulo operations. Compared to the EGH-filter it computes less of these operations, and the EGH-filter is already noticed to be faster than Bloom filters that use hashing heavier than simple modulo-hashing [12]. Hence the MCRS-generator will be at least equivalent to a Bloom filter in speed, and with equal practical performance for large values of d it is certainly an interesting candidate for further study. Sadly however it has worse performance for small values of d, which may result in it being unsuitable for certain applications. This behaviour comes from how the 1's are placed in a vector by the MCRS-algorithm. The algorithm constrains each 1 to a small subset of all the available indices. As an example if the first prime chosen is 3 then the first 1 can only be placed somewhere among the three first indices. If we then have a value d above 3, then the probability that each of the three first indices is set to 1 approaches 1 as d increases, making the first 1 virtually unnecessary for querying membership. In practice this would then entail that the level of the pattern is not equal to what would be optimal to use for low FPR.

5.4 Analysis of the MCRS-algorithm

Since the MCRS-algorithm works well both for pregenerated- and run-time generated patterns we dedicate this section to analyzing its properties. For both analyses, we consider a case where we select $k = ||\frac{m}{d} * \ln 2||$ powers of primes $p_1..p_k$.

5.4.1 Run-time generation

To analyze the properties when the patterns are generated at run-time is an easier task and is therefore put first. We calculate the probability that an index in our generated pattern is already set to 1 by some previous insertion.

Given the construction of the algorithm, the first index of a pattern will exist somewhere in the p_1 first indices. Hence after d insertions, the probability that this index is 1 is:

$$1 - \left(1 - \frac{1}{p_1}\right)^d .$$

The second 1 will exist in the range p_1 to $p_2 + p_1$ and so on. The probability that a pattern is contained in the filter already is then:

$$\prod_{i=1}^{k} \left(1 - (1 - \frac{1}{p_i})^d\right) .$$

This equation again reaffirms our aim to have a maximum product of primes for the algorithm, which is trivially obtained by taking k primes (or powers thereof) close to each other. In general, it is not necessary to choose powers of primes for low values of m since the gap between low-number primes is usually small. However, when m grows it can be challenging to find primes sufficiently close to each other since the Prime number theorem states that the average gap between primes increases as the universe, in our case the value m, grows. In these cases it is often beneficial to take some powers of primes instead, resulting in the selected values being closer, giving us a higher product. The equation also illustrates our previous point regarding that some submatrices will be dense. If $d > p_k$ then that index will be less and less useful as d increases to lower the FPR.

5.4.2 Preconstructed patterns

Here we give some properties for the construction for certain instances, together with a intuitive understanding of the properties of the patterns. But first we give a timecomplexity for constructing the patterns.

Proposition 5.0.1. *Given that the primes are selected the construction of the MCRS-matrix runs in $\mathcal{O}(kn)$ time.*

Proof. For each column in the matrix k 1's are inserted by performing k modulo operations. For n columns this leads to a complexity of $\mathcal{O}(kn)$. □

This relatively low complexity indicates that the major time sink when initializing a BBFBP with this algorithm comes from selecting the primes if a good algorithm for that task is not deployed. But it also indicates that if the structure is initialized with the primes already chosen then the initialization will be much faster than the standard approach of sampling which requires random values. This makes it beneficial to use this algorithm for example transferring the filter since only the blocks along with the primes need to be transmitted, and the patterns can quickly be reconstructed on the other end. It is also of note that each column can be constructed independently and as such the matrix can be constructed entirely in parallel.

Lemma 5.1. *Let p_1, p_2 be the two smallest powers of primes selected by the MCRS-algorithm. If $n < p_1 * p_2$, and all the powers of primes are co-prime, then the patterns constructed by the MCRS algorithm form a $(m, n, 2(k-1), k)$-constant-weight binary code $\mathcal{C}$.*

Proof. By virtue of [23], we know that a group testing matrix is equivalent to an ECC $\mathcal{C}$, with each column in the matrix being a codeword in $\mathcal{C}$. A pattern in a BBFBP is also equivalent to a column in a group testing matrix [9], therefore it follows that a pattern in a BBFBP is equivalent to a codeword in some ECC. Since each pattern created by the MCRS-algorithm has constant Hamming weight it also follows that the MCRS-algorithm creates a constant weight binary code with n codewords of length m with weight k. Now, if $n < p_1 * p_2$ then the MCRS-algorithm will construct pattern that have at most one bit set to 1 in common, since the first time two bits will be set in common is at pattern $p_1 * p_2$ due to the fact that $gcd(p_1, p_2) = 1$. Therefore the minimum pairwise Hamming distance between patterns equal $2 * (k - 1)$, and hence the MCRS-algorithm constructs a $(m, n, 2 * (k - 1), k)$-constant-weight binary code $\mathcal{C}$. □

This property of the MCRS-algorithm is important as it gives us the following insight of the disjunctive properties:

Corollary 5.1.1. *Let p_1, p_2 be the two smallest powers of primes selected by the MCRS-algorithm. If $n < p_1 * p_2$, and all the powers of primes are co-prime, then the patterns constructed by the algorithm form a (k-1)-disjunct matrix.*

Proof. This follows immediately from a theorem presented in [23] which states that a (m, n, h, w)-constant-weight binary code gives a $\frac{w-1}{w-h/2}$-disjunct matrix, where h in our case equals $2 * (k - 1)$ and $w = k$ according to Lemma 5.1. □

Both of these properties of course also applies to the regular CRS-algorithm. While this is interesting, it does not help us much since k is typically smaller than d/b, at least for genomic applications. We also previously stated that we are more interested in almost disjunct matrices due to their lower need for tests, so we continue by analyzing the algorithms performance in constructing (d, ε)-disjunct matrices. However, as a sideline, these properties can be used to improve the original construction of the CRS-algorithm when d is high as described in chapter 4.

Sadly, the bounds established by Mazumdar on constant weight codes used as almost disjunct matrices [27] are not usable in our case. These bounds grow too quickly as d increases for it to give any insight into how the matrix behaves when used as patterns. However, they do give insights into what properties a high performing matrix have, which we will use as a basis for further discussion.

Mazumdar notes that a good pattern design has a high minimum Hamming distance and a high average Hamming distance compared to the weight of the codes [27]. A randomized construction, such as the CHE-algorithm, gives no guarantees regarding either of these. Even a RDkSD only guarantees a minimal Hamming distance of 2. The MCRS-algorithm does however give guarantees for a higher minimal Hamming distance. As Lemma 5.1 outlines, the Hamming distance is guaranteed to be $2(k-1)$ until pattern $p_1 * p_2$.. At this point the minimal Hamming distance drops to $2(k-2)$. It will then drop first at pattern number $p_1 * p_2 * p_3$ to $2(k-3)$ and so on. This means that as the patterns increase the Hamming distance drops, however, it will do so for the randomized constructions with a high probability as well. For small amounts of patterns however, the MCRS-algorithm guarantees a high minimal Hamming distance which gives it good performance compared to the randomized approaches. In many practical applications with large amounts of insertions it is also likely that $n < p_1 * p_2$.

For example, the application outlined by Georganas et al. uses Bloom filters with a 5% FPR, translating into roughly 5 bits per element in a standard filter [2]. Let us consider this filter as a BBFBP with $m = 512$ and $n = 4096$ and a arbitrary amount of blocks, filling the entire L2 cache with patterns on a consumer computer. In such a case, the number of primes selected is around $k = 4$ which could result in the primes $\{113, 127, 131, 137\}$. Since the two smallest primes here are 113 and 127 we get $113 * 127 = 14351$ patterns before the minimum Hamming weight drops, which is larger than n by a significant margin. Hence all of the above mentioned properties hold if we chose to design the patterns using the MCRS-algorithm.

5.5 Improved CRS-algorithm

We dedicate this section to analysing our proposed alteration for the CRS-algorithm for higher values of d.

The disjunctive properties of this construction comes from the properties discussed in the previous section. Corollary 5.1.1 states that if $n < p_1 * p_2$, with p_1 and p_2 being the two smallest primes, then a CRS-construction yields a $(k-1)-$disjunct matrix. Remember that the two smallest primes chosen by this modification both are larger than $\sqrt{n}$. Clearly, since p_1 and p_2 are both larger than $\sqrt{n}$ then the property $n < p_1 * p_2$ holds. Since we are in total choosing $d+1$ primes, the Hamming weight of each column will be $k = d+1$ according to the construction of the matrix. Again, according to Corollary 5.1.1 this implies that our matrix will be $((d+1)-1)-$disjunct.

It is not immediately obvious why this is better than the regular algorithm, but the strength of this modification comes from the fact that it grows much slower in terms of tests as d increases. Consider how many extra tests would be needed to turn a $d-$disjunct matrix into a $(d+1)-$disjunct matrix using the two constructions. For the regular algorithm, primes needs to be added so that the total product of primes is now equal or larger to n^{d+1}. Clearly, this is on average more than one prime since otherwise we would take a prime larger than n which would result in more tests than items. For the modified version only one prime needs to be added which results in a much slower growing function. This is illustrated in the graph below for $n = 10^4$.

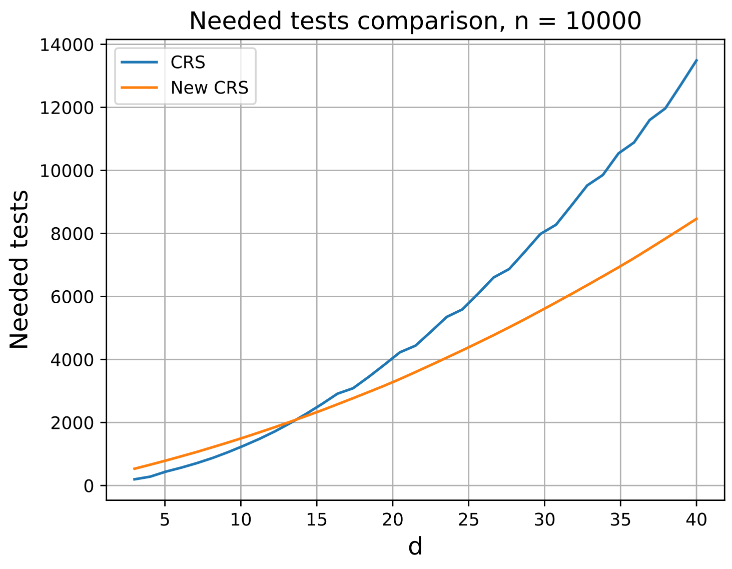


Figure 5.1: A comparasion of needed tests for the old CRS and the New CRS algorithm. $n = 10^4$

However, for small values of d the new algorithm performs worse in the needed amount of tests. To give the user some indication of the needed tests we therefore provide an approximation of the new algorithms performance.

According to the prime number theorem, the average gap between primes below the integer X can be approximated by $\log X$. We use this gap to approximate the first prime to be below $\sqrt{n} + \log X$, since the root of n might be a prime the closest prime above should be approximate $\log X$ apart. The second prime can then be approximated by $\sqrt{n} + 2 * \log X$ and so on. This gives us the function

$$\sum_{i=1}^{d+1}(\sqrt{n} + i * \log X) = \frac{d+1}{2} * ((d+2) * \log X + 2\sqrt{n})$$

which just needs a value for X to be complete. By experimental evaluation we have found that $\frac{n}{d}$ gives a good value for X, which can be motivated by the fact that if the primes selected are above this value then the total number of tests would be more than the number of items, making the construction useless. By setting X to this value we then get the following approximation:

$$\frac{d+1}{2} * ((d+2) * \log(n/d) + 2\sqrt{n}).$$

While we do not provide a bound here on the needed number of tests, this equation provides a solid approximation on the needed number of tests used by the algorithm. One should note, however, that this equation is not perfect: it tends to result in lower amounts of tests than the actual construction if the needed tests exceed the number of items. While this is clearly a weakness, it is not a problem in practice since we will never use the algorithm when the approximated number of needed tests exceed the available items. We leave calculating the exact point d for which the new construction is superior as future work.

5.6 Future work and unsolved problems

In this section we outline some aspects that we have found interesting but not had time to pursue. This includes a rundown of some attributes of the MCRS-algorithm, and also a discussion regrading our evaluation of run-time patterns.

5.6.1 Unsolved problems regarding the MCRS-algorithm

While the MCRS-algorithm works better than the CHE-algorithm for smaller amounts of patterns (and is equal in performance for larger pattern-sets), the deterministic construction raises some question whether it could be optimised further. As outlined in the previous section, the minimal Hamming distance in an MCRS-set decrease (together with the average Hamming distance) when the number of patterns grows. Guided by the bounds established by Mazumdar on constant weight codes [28], we know that an efficient design has a high *average* pairwise Hamming distance between patterns, together with a high minimum Hamming distance. An approach to get a higher distance could be to use some mixture of patterns on two neighbourhood levels, as outlined in the conjecture found in [9]. However, this must also be done so that the ratio between 0's and 1's in the filter after d insertions is still approximately 50/50 and without lowering the minimal Hamming distance. While our results did not give any support for the conjecture for a random design (see chapter 4), we have certain indicators that it might be usable in conjunction with the deterministic design of the MCRS-algorithm.

The primary indicator for this is the fact that the k 1's in a pattern generated by the MCRS-algorithm are not uniformly distributed. In fact, the submatrix constructed by the smallest prime while have significantly higher density than the other submatrices, as seen by example in Figure 5.2. While this is generally not a problem for precomputed patterns, since the collision probability dwarfs the containment probability for values of d where this difference is apparent, this is an issue at run-time construction as shown in the results. We can remedy this somewhat by modifying the run-time construction. If the level k results in a smallest prime p_i so that $d > p_i$,

1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0
0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1
0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0
1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0
0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0
0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0
0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1
1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0
0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0
0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0
0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0
0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1
0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0

Figure 5.2: The MCRS-algorithm for m=15, n=20 with k=3. The primes selected are 3, 5 and 7. From this example one can clearly see that the first 3 rows have a significantly higher density than the 7 last. The maximum number of unique patterns that can be generated by these primes is 3*5*7 = 105. We also see that both the average and minimal Hamming distance decreases as we add pattern 16, which is equal to pattern 1 in two bits.

we can decrease the level k by 1 and chose new primes. Motivating this is the fact that this submatrix will be essentially useless to determine membership since its density is too high, and will as such only constrain the size of the other primes, lowering the overall FPR. By instead constructing the vectors on a lower level we essentially check the same number of indices, but they have a larger space in which they can exist. While this solution does lower the FPR for small d, it is still a factor above the regular Bloom filter in terms of performance. Hence, there might be a better solution to this problem without lowering the level of the vectors.

Another indicator of a possible performance weakness is the constrained amount of patterns that can be created. Our results indicate that the FPR decreases as n grows for the MCRS-design, albeit slowly. However, the number of unique vectors that can be generated by the MCRS-algorithm is significantly smaller than what can be generated by the CHE-algorithm. We illustrate this by example in Figure 5.2: Since the primes chosen are 3, 5 and 7 the maximum amount of patterns that can be constructed are 3*5*7=105. However, with random sampling, it would be possible to construct $\binom{m}{k}$ different vectors, in our case $\binom{15}{3} = 455$. It might be possible to increase the number of available patterns by adding patterns on a neighbourhood level, yet again motivated by the fact that the optimal support can be found on a weak antichain [9], and that vectors with constant weight cannot be optimal. While this is often not needed when using precomputed patterns since the cache size constrains the number of patterns that can be used, this might give improvements when computing vectors at run-time.

A motivating factor behind further research into this is the fact that the run-time construction of these patterns achieved equal performance to a regular Bloom filter in practice for small bits-per-element ratios. Any improvements to this algorithm that can also be carried over to a run-time scenario might in turn help construct a new data structure with better performance than the regular Bloom filter.

5.6.2 Run-time construction evaluation

In our comparison between run-time constructed patterns we only considered the FPR between generators and gave just small indications of the actual performance in rejection time. This is mainly due to that real-world implementation of filters can be hardware implemented, or at least fully parallel at a filter level. Since we decided on the FPR as the focal point our parallelism lay much higher than the individual filter and could thus give no good values for the actual time it took to query the filter. Hence we have no conclusive results on how efficient the pattern generators really are in terms of computational speed. It would be interesting to see how these fare in "real" applications, especially for the MCRS-generator.

5.6.3 The relation between patterns and blocks

While this thesis did in general only concern itself with finding a good pattern design for a prescribed value n, there is another aspect to FPR-minimisation for BBFBPs that we did not consider. As outlined previously, both the patterns and the blocks need to fit on the cache for efficiency in execution for the filter. While increasing the number of patterns reduces the collision probability, it also constrains the number of blocks that can be hosted on the cache. Hence it may not always be beneficial to increase n, especially if it means that we have to reduce our number of blocks, giving in practice a smaller bit per element ratio. An open problem thus remains what the optimal ratio of blocks to patterns are for a fixed number of insertions. We note however that the MCRS-algorithm works comparatively better than the CHE-algorithm for small blocks and will as such be a better choice for pattern construction should the optimal number of patterns be small.

6 Conclusion

In the thesis we have analysed how different group testing algorithms used to generate patterns in BBFBP in terms of FPR, together with investigations on how the FPR behaves for neighbourhood levels k and $k + 1$ for patterns. We also provide an improvement to the regular CRS-algorithm for larger number of defectives. Our main result, however, comes in the form of a modification to the Chinese Remainder Sieve algorithm. We show that when using the resulting matrix as patterns in a BBFBP the resulting FPR is lower for all problem instances compared to the regular pattern design, and especially for low amounts of patterns. The design is deterministic and can be reconstructed from just a small amount of auxiliary memory which, apart from superior FPR, gives certainty for the user concerning performance. It also enables the filter to quickly be transmitted by just sending the blocks together with the chosen primes. The design works particularly well when the amount of bits per element is low which makes it a viable alternative for genomic applications where the number of insertions is high. While it is not yet strong enough to rival regular Bloom filters when generating patterns at run-time, we have identified several key features of the algorithm which could be improved upon which could result in better performance than a regular Bloom filter.

While our results do improve the FPR for BBFBPs we observe that the new algorithm does not construct patterns with optimal FPR. Hence this area is still in need of continuous work to improve both the group testing algorithms and, in turn, improving Bloom filters with bit patterns. Any improvements in this area will lead to faster and safer data structures, and their recent prominence in the field of bioinformatics gives ample motivation for further studies in optimising these structures.

Bibliography

[1] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970.

[2] E. Georganas, S. Hofmeyr, L. Oliker, R. Egan, D. Rokhsar, A. Buluc, and K. Yelick, "Extreme-scale de novo genome assembly," *Exascale Scientific Applications: Scalability and Performance Portability*, p. 409, 2017.

[3] A. Broder and M. Mitzenmacher, "Network applications of bloom filters: A survey," *Internet Mathematics*, vol. 1, no. 4, pp. 485–509, 2004.

[4] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, pp. 4:1–4:26, Jun. 2008.

[5] J. K. Mullin, "Optimal semijoins for distributed database systems," *IEEE Transactions on Software Engineering*, vol. 16, no. 5, pp. 558–560, 1990.

[6] R. S. Roy, D. Bhattacharya, and A. Schliep, "Turtle: Identifying frequent k -mers with cache-efficient algorithms," *Bioinformatics*, vol. 30, no. 14, pp. 1950–1957, 2014. [Online]. Available: +http://dx.doi.org/10.1093/bioinformatics/btu132

[7] P. Pandey, M. A. Bender, R. Johnson, R. Patro, and B. Berger, "Squeakr: an exact and approximate k-mer counting system," *Bioinformatics*, vol. 34, no. 4, pp. 568–575, 2018.

[8] P. Melsted and J. K. Pritchard, "Efficient counting of k-mers in dna sequences using a bloom filter," *BMC bioinformatics*, vol. 12, no. 1, p. 333, 2011.

[9] P. Damaschke and A. Schliep, "An optimization problem related to bloom filters with bit patterns," in *SOFSEM 2018: Theory and Practice of Computer Science*, A. M. Tjoa, L. Bellatreche, S. Biffl, J. van Leeuwen, and J. Wiedermann, Eds. Lecture Notes in Computer Science, vol. 10706, Springer, 2018, pp. 525–538.

[10] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo filter: Practically better than bloom," in *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '14. New York, NY, USA: ACM, 2014, pp. 75–88. [Online]. Available: http://doi.acm.org/10.1145/2674005.2674994

[11] A. Pagh, R. Pagh, and S. S. Rao, "An optimal bloom filter replacement," in *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '05. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2005, pp. 823–829. [Online]. Available: http://dl.acm.org/citation.cfm?id=1070432.1070548

[12] S. Z. Kiss, E. Hosszu, J. Tapolcai, L. Rónyai, and O. Rottenstreich, “Bloom filter with a false positive free zone,” 2018.

[13] F. Putze, P. Sanders, and J. Singler, “Cache-, hash-, and space-efficient bloom filters,” *J. Exp. Algorithmics*, vol. 14, pp. 4:4.4–4:4.18, Jan. 2010.

[14] R. Dorfman, “The detection of defective members of large populations.” *The Annals of Mathematical Statistics*, vol. 14, no. 4, p. 436–440, 1943.

[15] E. Barillot, B. Lacroix, and D. Cohen, “Theoretical analysis of library screening using a n-dimensional pooling strategy,” *Nucleic acids research*, vol. 19, no. 22, pp. 6241–6247, 1991.

[16] E. S. Hong and R. E. Ladner, “Group testing for image compression,” *IEEE Transactions On image processing*, vol. 11, no. 8, pp. 901–911, 2002.

[17] D. Du, F. Hwang, and E. C. (e-book collection), *Pooling designs and nonadaptive group testing: important tools for DNA sequencing.* New Jersey: World Scientific, 2006, vol. 18.

[18] D. Eppstein, M. T. Goodrich, and D. S. Hirschberg, “Improved combinatorial group testing algorithms for real-world problem sizes,” *SIAM Journal on Computing*, vol. 36, no. 5, pp. 1360–1375, 2007. [Online]. Available: https://doi.org/10.1137/050631847

[19] C. L. Chan, P. H. Che, S. Jaggi, and V. Saligrama, “Non-adaptive probabilistic group testing with noisy measurements: Near-optimal bounds with efficient algorithms,” in *2011 49th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, Sept 2011, pp. 1832–1839.

[20] M. Aldridge, L. Baldassini, and O. Johnson, “Group testing algorithms: Bounds and simulations,” *IEEE Transactions on Information Theory*, vol. 60, no. 6, pp. 3671–3687, June 2014.

[21] A. J. Macula, “A simple construction of d-disjunct matrices with certain constant weights,” *Discrete Mathematics*, vol. 162, no. 1-3, pp. 311–312, 1996.

[22] W. J. Bruno, E. Knill, D. J. Balding, D. Bruce, N. Doggett, W. Sawhill, R. Stallings, C. C. Whittaker, and D. C. Torney, “Efficient pooling designs for library screening,” *Genomics*, vol. 26, no. 1, pp. 21–30, 1995.

[23] W. Kautz and R. Singleton, “Nonrandom binary superimposed codes,” *IEEE Transactions on Information Theory*, vol. 10, no. 4, pp. 363–377, 1964.

[24] L. Gyorfi, J. Massey *et al.*, “Constructions of binary constant-weight cyclic codes and cyclically permutable codes,” *IEEE Transactions on Information Theory*, vol. 38, no. 3, pp. 940–949, 1992.

[25] F.-W. Fu, A. H. Vinck, and S.-Y. Shen, “On the constructions of constant-weight codes,” *IEEE Transactions on Information Theory*, vol. 44, no. 1, pp. 328–333, 1998.

[26] E. Porat and A. Rothschild, “Explicit non-adaptive combinatorial group testing schemes,” in *International Colloquium on Automata, Languages, and Programming.* Springer, 2008, pp. 748–759.

[27] A. Mazumdar, “Nonadaptive group testing with random set of defectives,” *IEEE Transactions on Information Theory*, vol. 62, no. 12, pp. 7522–7531, 2016.

[28] A. Barg and A. Mazumdar, “Group testing schemes from codes and designs,” *IEEE Transactions on Information Theory*, vol. 63, no. 11, pp. 7131–7141, 2017.

[29] M. Matsumoto and T. Nishimura, “Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator,” *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 8, no. 1, pp. 3–30, 1998.

[30] S. K. Park and K. W. Miller, “Random number generators: good ones are hard to find,” *Communications of the ACM*, vol. 31, no. 10, pp. 1192–1201, 1988.

[31] Y. Erlich, K. Chang, A. Gordon, R. Ronen, O. Navon, M. Rooks, and G. J. Hannon, “Dna sudoku—harnessing high-throughput sequencing for multiplexed specimen analysis,” *Genome research*, vol. 19, no. 7, pp. 1243–1253, 2009.

A
Framework usage tutorial

This chapter outlines a tutorial for using the framework for testing pattern designs. The framework allows users to send custom pattern designs to a BBFBP and compare its performance in terms of false positives to other designs. The complete code can be found at *https://github.com/Cannonbait/master-thesis*. The README for this project is at the time of writing identical in purpose to the one found in this appendix. The framework is built, and tested, for Ubuntu 16.04 with experimental support for Windows OS. We do not have any guarantees for other setups. The BBFBPs provided here are not written, or optimized, for commercial use, as they lack support for SIMD-instructions which is necessary for efficient filters of this type. We note however that many modern processors have built-in support for SIMD-instructions which allows for usage without hand-written assembly code. The Ubuntu installation should work for most recent nix-systems but is only tested for Ubuntu systems.

A.1 Installation

The analysis tools are built in Python while the inner loop is written in C++. Bindings between these are provided by Cython and as such Python3 and the g++ compiler must be installed (at least version 14), The instructions below assume that both Python3 and g++ exist on the users current machine.

A.1.1 Windows

Install GMP (high precision library), Cython and Boost

Note: The windows build is highly experimental. Should you manage to successfully install GMP, Cython and boost the setup in the following section should work. However we will not guide you through that process.

A.1.2 Ubuntu 16.04

```
\$ apt−get install python3−numpy python3−tk
\$ apt−get install libboost−dev libgmp3−dev
\$ pip3 install cython matplotlib pandas
```

A.2 Setup

After cloning the repository, move into the master-thesis/code-framework/ folder and execute the following commands. Given that all dependencies are installed as specified above, it will create a .so file which can be imported into python separately or used with the pre-built analysis program.

A.2.1 Windows

```
\$ cd <your path>/master-thesis/code-framework/cython
\$ python3 setup_serial_framework_win.py
```

Should the build fail it might be because of invalid pathings to the libraries (GMP etc). Check the setup_serial_framework_win.py file as well as the serial_framework.pxd and serial_framework.pyx files to correct the paths as installed on your system.

A.2.2 Ubuntu 16.04

```
\$ cd <your path>/master-thesis/code-framework/cython
\$ python3 setup_serial_framework_nix.py build_ext --inplace
```

A.3 Running the analysis tool

The analysis tool can be executed either from the command line for existing designs or through Python code with a few lines. Below are instructions on both of these cases.

A.3.1 Command line

From the command line the analysis tool can run for a number of parameters and existing designs. While this is not immediately useful since you cannot pass new designs into the program this way, it helps the user get familiar with the parameters and how they can vary.
To run a quick demo of the program, use the following commands which should yield the following output:

```
\$ cd <your path>/master-thesis/code-framework/python
\$ python3 analysis.py -che -comp
Found no "source" argument, trials will be run with random
input
Initializing filters...
```

This will run a quick demo comparing the CHE (Cache- Hash- efficent) and COMP (Combinatorial Orthogonal Matching Pursuit) algorithms as pattern designs for some default parameters. The execution time can vary from machine to machine depending on the cachesize. This should in the end display a graph looking like this:

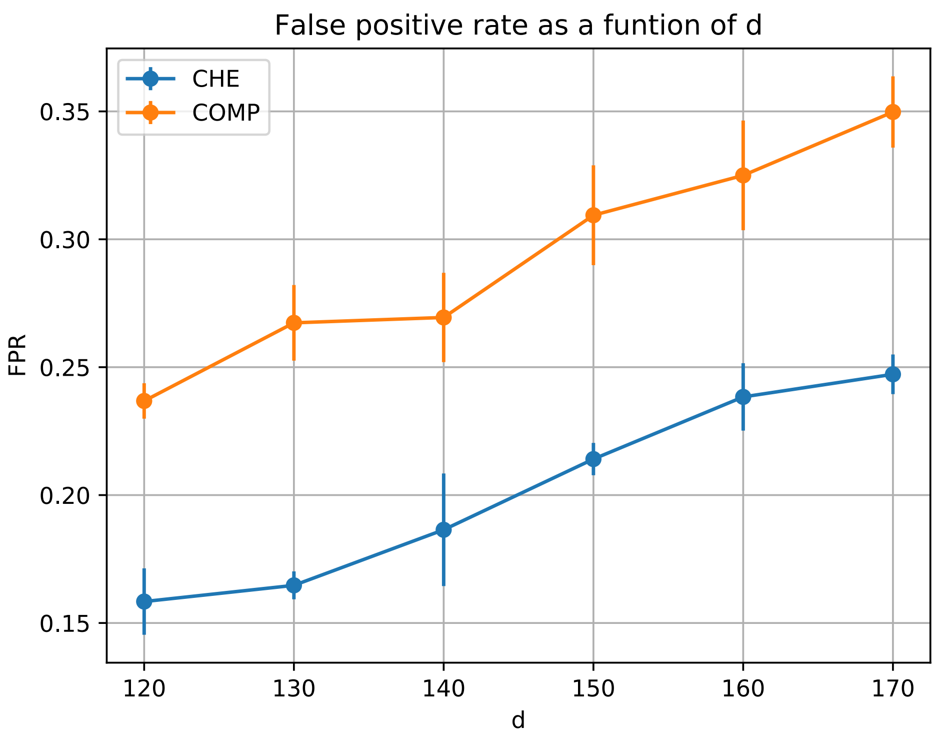


Figure A.1

Where the Y-axis represents the false positive rate (FPR) with standard deviation and the X-axis the number of insertions into the filter (d). The exact visual presentation will vary since the FPR is not guaranteed to be the same for each run of the program. The supplied parameters can be customized be various flags at application start. The customizable parameters can be seen on the following page.

- **-m=x** where x is the number of bits in the filter (defaults to 512)
- **-n=x** where x is the number of patterns in the filter (defaults to 4096)
- **-d=x** where x is the number of insertions in the filter (defaults to 120)
- **-b=x** where x is the number of blocks in the filter (defaults to 1)
- **-tests=x** where x is the number of tests on datapoints towards the filter. (defaults to 10000)
- **-y_step=x** where x is the step size on the varying parameter y as explained below (defaults to 1)
- **-pattern_trials=x** where x is the number of times the patterns should be regenerated according to the specified designs and tested. This parameter affects the standard deviation in the graph (defaults to 5)
- **-interpret** Interprets the labels for ease of reading the plots, for example turns "d" to "inserted items" when plotting the data.
- **-comp** the COMP-algorithm as a pattern design (used for comparative purposes)
- **-che** the CHE-algorithm as a pattern design (used for comparative purposes)
- **-mcrs** the MCRS-algorithm as a pattern design (used for comparative purposes)
- **-source="<file>"** a source file with precomputed data in number format, separated by line break. The framework allows for high precision and can parse numbers up to 120 characters long. These numbers are then hashed using modulo hashing for a more uniform distribution. It is possible to declare multiple sources, in which case all the declared sources will be used in separate trials. In order to declare the explicit use of a mersenne twister as a source the command is made as **-source=Random**. If no source is declared then a mersenne twister is automatically used.

To be able to get a visualization, one or two parameters must vary. This is done by adding a new flag on the varying parameter(s) with "_end" appended. For example, if one wants to experiment for varying values of d as in the example, one can choose the flags -d=120 and -d_end=160. The parameters that can vary are m, n, d and b. If two parameters vary, the displayed graph will be in three dimensions, for example:

```
\$ python3 analysis.py -mcrs -comp -d=1200 -d_end=1600
-n=1500 -n_end=2000 -b=10 -d_step=100 -n_step=100
Found no "source" argument, trials will be run with random
input
Initializing filters...
```

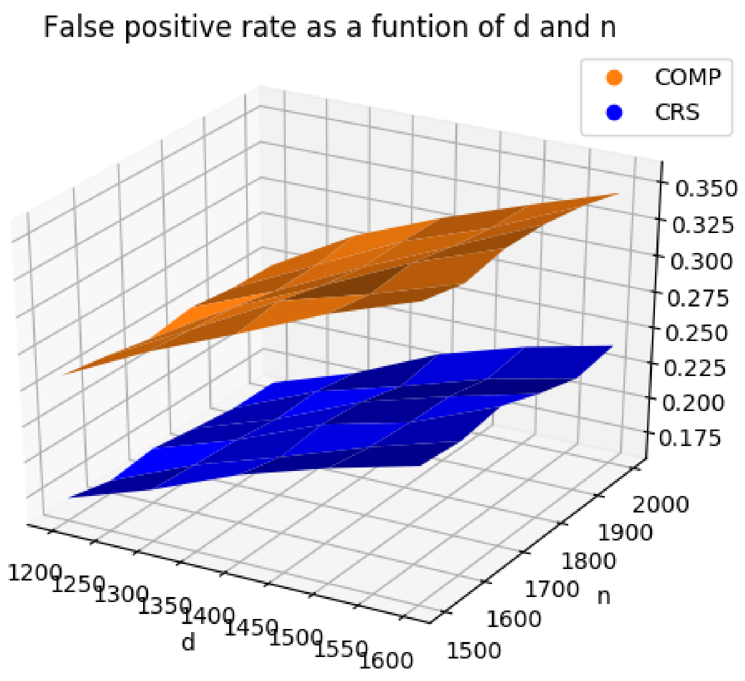


Figure A.2

For reasons of readability, standard deviation will not be displayed when plotting the data in 3D. One should also note that generating 3D-plots take significantly more time than a standard 2D-plot.

A.3.2 Python code

To run the program from Python, three steps need to be completed. Setting the various parameters for the experiments as outlined above, generating data and displaying it. This is done in the following fashion:

```python
import sys

from analysis_settings import AnalysisSettings
from display import display_data
from data_generator import generate_data

settings = AnalysisSettings(sys.argv)
(result, deviation) = generate_data(settings)
display_data(result, deviation, settings)
```

Data can of course be generated without displaying it. If you do not want to set the flags at upstart, the argumentline can be changed to any list of strings which contains flags. For example, this is also valid code:

```python
import sys

from analysis_settings import AnalysisSettings
from display import display_data
from data_generator import generate_data

sys.argv[1:] = ["-d_end=151", "-d_step=10", "-b_step=2",...
"-b_end=21", "-che", "-comp"]
settings = AnalysisSettings(sys.argv)
(result, deviation) = generate_data(settings)
display_data(result, deviation, settings)
```

A.4 Writing new patter-design generators

Writing a new generator is a fairly easy task. Provided for guidance is an interface called **IPatternGenerator** located in the

master-thesis/code-framework/python/pattern_design

folder. This interface is required of the generators to implement but only demands two definitions: **get_name** and **generate_patterns(m,n,d,b)**, where **generate_patterns** should return a matrix of dimensions **mxn** as patterns in the BBFBP. The **get_name** method is only used in the visualization for easy of identifying the new algorithm. Below is an example of a (stupid) pattern generator:

```python
from pattern_design.pattern_interface import IPatternGenerator
import numpy as np

class ExampleGenerator(IPatternGenerator):

    def generate_patterns(m,n,d,b):
        patterns = np.zeros((m,n), dtype='bool')
          for i in range(n):
            patterns[i % m][i] = 1
        return patterns

    def get_name():
        return "Example Generator"
```

Any generator created in this way can be added to the settings as displayed above (using the add_designs method). This can be done in the following fashion:

```python
import sys

from analysis_settings import AnalysisSettings
from pattern_design.example import ExampleGenerator
from display import display_data
from data_generator import generate_data

settings = AnalysisSettings(sys.argv)
settings.add_designs([ExampleGenerator])
(result, deviation) = generate_data(settings)
display_data(result, deviation, settings)
```

The *add_designs* method can take multiple generators in its parameter list.

A.5 Generating data

To use a source-file in the framework, i:e some real data, the file must be formatted in a special way. Each line in the file must contain only numbers which are then parsed into the program. These numbers may be of any length, but be warned that the string representation will be used in the program to make sure that we do not count true negatives. Hence if the string representation becomes to long, and to many items are stored in the filter, then we can still fail while allocating memory. The need for high precision comes from the framework being tested on biomedical data, which can be generated from .fastq files in the following fashion:

```
\$ cd <your path>/master-thesis/code-framework/data-generation
\$ python3 data\_preparation.py --source="<your file>"
Reading from file: <your file>
Translating characters
Writing output to: <your file>.prep
```

What this does is taking a file consisting of characters **G, A, T, C** and **N**, converting these to an integer based on their binary mapping and then finally writing the data to a .prep file, which can be used as a source file for the framework. All other characters will be ignored by this. Separate from this framework, the data can also automatically be constrained through modulo-operations by supplying the following flags (for example for usage in Turtle or other *k*-mer counting software).

- **threads=x** where x is the number of threads in the application
- **patterns=x** where x is the number of patterns in the filter
- **blocks=x** where x is the number of blocks in the filter
- **-h or –help** Displays example usage of the program
- **–output=<filename>** Specifies another filename for the output.

B

Primeselection in the MCRS-algorithm

Below is a simple, greedy, algorithm for selecting k primes for use in the MCRS-algorithm. Although far from perfect, this can give an initial approach for future implementations. For efficiency this algorithm does not calculate which numbers that actually are prime, but assumes that all primes below some fixed, realistic number exists as a header definition.

The approach is as follows: Take k unique primes so each is as large as possible, but below $\frac{m}{k}$. This ensures that their sum does not exceed m. Then do a second pass through the selected primes, starting with the lowest prime. See if this prime can be replaced by a larger, as of not yet selected prime so that the sum of primes is still below m. The replacement prime does not necessarily have to conform to the constraint to be below $\frac{m}{k}$. Return the selected primes as primes for the MCRS-algorithm.

Algorithm 9 Greedy Prime selection algorithm

```
   selected := {}
2: for i = 1 to k do
       selected[i] := the largest prime $p \leq \frac{m}{k}, p \notin$ selected.
4: for i = k to 1, i– do
       check if selected[i] can be replaced with a larger prime $p$ such that
   $\sum selected < m, p \notin$ selected. If so, replace selected[i] with $p$.
6: return selected
   End
```

C

MCRS-primes

Since the primes used by the MCRS-algorithm were predetermined for experimental purposes, the chosen primes are listed below for possible experimental reevaluation. 18 is the maximum number of primes that can be chosen for $m = 512$

k	**Chosen primes**
1	509
2	241,269
3	151,179,181
4	127, 131, 137, 113
5	101, 103, 107, 73, 127
6	61, 79, 83, 89, 97, 101
7	83, 59, 61, 67, 71, 73, 97
8	47, 53, 59, 61, 67, 71, 73, 79
9	43, 47, 53, 59, 61, 67, 71, 73, 37
10	31, 37, 41, 43, 47, 53, 59, 61, 67, 71
11	23, 29, 31, 37, 41, 43, 47, 53, 61, 67, 79
12	19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67
13	13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 97
14	11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 67, 79
15	7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 7
16	5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 73
17	3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 73
18	2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61