Journal on Satisfiability, Boolean Modeling and Computation 8 (2014) 129-148

Satisfiability-based Set Membership Filters

Sean A. Weaver saweave@tycho.ncsc.mil
Katrina J. Ray dr.katieray@gmail.com
Victor W. Marek marek@cs.uky.edu
Andrew J. Mayer mayer@ccrwest.org
Alden K. Walker akwalker@math.uchicago.edu

Abstract

Introduced here is a novel application of Satisfiability (SAT) to the *set membership problem* with specific focus on efficiently testing whether large sets contain a given element. Such tests can be greatly enhanced via the use of *filters*, probabilistic algorithms that can quickly decide whether or not a given element is in a given set. This article proposes *SAT filters* (i.e., filters based on SAT) and their use in the set membership problem. Both the theoretical advantages of SAT filters and experimental results show that this technique yields significant performance improvements over previous techniques. Specifically, a SAT filter is a filter construction that is simple yet efficient in terms of both query time and filter size; i.e., SAT filters asymptotically achieve the information-theoretic limit while providing fast querying. As well, this is the first application that makes use of the random k-SAT phase transition results and may drive research into efficient solvers for this and similar applications.

KEYWORDS: *Satisfiability, Set Membership, Filters, Random k-SAT*

Submitted August 2013; revised December 2013; published January 2014

1. Introduction

This paper describes a novel and simple technique for building efficient set membership filters (in terms of both the amount of long-term storage and query time) by translating data sets to instances of *Satisfiability* (SAT) [8] and solving them with a SAT solver.

1.1 Set Membership

Set membership testing is encountered in many computer applications. However, some terms must first be defined before being able to accurately discuss the topic. Let D be a particular domain. The element being tested for and the set being tested against will be drawn from D. Some examples of such a domain may be the collection of all strings in the English alphabet, the collection of all valid HTML documents, the collection of all digital JPEG encoded images, or the collection of all binary strings of length 1024. These examples should make it clear that the domain may be very large, potentially infinite.

The *set membership problem* is then the following: given an element $x \in D$ and a set of interest $Y \subseteq D$, determine if $x \in Y$. For the applications presented here, Y will be finite, but potentially very large. To continue with the example given previously where D is the set of all strings in the English alphabet, let Y be the set of all words defined in a given

dictionary. The set membership problem is now akin to spell checking, i.e., testing if a string of characters is a word defined in the dictionary.

1.2 Filters

If the elements of D have a simple representation and if $|Y|$ is small, a *naïve approach* to set membership testing can be taken. Namely, list all representations of elements of Y in an array A of length $|Y|$ and then, given an element $x \in D$, compare the representation of x against every entry of A. Unfortunately, testing set membership in this way is inefficient in a variety of situations, particularly when $|Y|$ is very large. For this reason, the set membership problem is often solved by first querying a *filter*. A filter is a mathematical object that can be *queried* with an element, returning either *Maybe* or *No*. In both cases discussed in later sections (Bloom filters and SAT filters), *Maybe* is interpreted by the user as *possible presence* and *No* is interpreted as *definite absence*. Observe that, by contrast, in the naïve approach a positive response is interpreted as *definite presence* and *No* is interpreted as *definite absence*. This means that a filter admits *false positives*, a phenomenon that does not occur when using the naïve approach. Because of this, a secondary test (such as the naïve approach) is often used as a fallback when a filter returns *Maybe*.

Hence, the point of a filter is to provide an efficient *primary* test for set membership. The space used to store a filter for Y is ideally far less than the space necessary to store Y, and the time required to query a filter is ideally far less than the time required to query Y, even in the case where Y has some natural order and intelligent search can be used. The trade-off for the decrease in time and space is that, as mentioned above, the answers returned by the filter are imperfect. An element which passes the filter may require a costly secondary test, but this test needs to be performed rarely.

1.3 Examples

Set membership filters are truly pervasive in modern computing. For example, [33] states that Google's Bigtable makes heavy use of filters and directly supports many popular Google services such as Google Maps, Google Earth, and Web indexing. In terms of security, Google Chrome uses filters to detect malicious websites and virus scanners make use of filters to speed up virus detection [30]. Filters are also used to support network routing, packet monitoring, and P2P networks, just to name a few. Two good surveys of different filter constructions and their uses in real-world applications are provided in [12, 33]. These surveys provide much more detail on the preceding applications as well as many others.

To give one simple concrete example, consider the set D of all strings of 8 digits. Suppose some subset $Y \subseteq D$ of the strings are to be stored in a filter. The filter needs to compress Y in a lossy way such that membership testing is efficient. One possibility is to create a list F_Y containing the first digit of all elements in Y. It is easy to see that $0 \leq |F_Y| \leq 10$, which is potentially much smaller than Y. Given a new string $x \in D$, if the first digit does not appear in F_Y (a quick test), x cannot be an element of Y and is rejected. If the first digit is in F_Y (also a quick test), then maybe $x \in Y$, but further testing is necessary.

1.4 Outline of the Article

This article is structured as follows. § 2 describes a well-known filter construction, namely, the Bloom filter. § 3 describes a new filter construction, called a SAT filter, of which instances are built by finding solutions to random k-SAT instances. The SAT filter asymptotically achieves the information-theoretic limit; in particular, it is more memory efficient than the popular Bloom filter while retaining many of its good qualities. § 4 describes formal methods for comparing filter constructions as well as provides a formal analysis of both Bloom filters and SAT filters. § 5 shows how to select parameters for SAT filters in real-world situations. § 6 presents results of an experimental implementation of SAT filters and also compares Bloom filters against what is currently achievable by SAT filters.[1] § 7 provides some concluding remarks.

2. Bloom Filter Construction

In [9], Bloom proposes the following filter construction, although a simplified version is presented here first. Let D be any set (the domain), let $Y \subseteq D$ with $m = |Y|$, and let the memory available for the filter B_Y be n bits. Choose a hash function $h : D \to \mathbb{Z}$ that maps the elements of D uniformly at random into the range $[0, n)$. Initialize all the bits of B_Y to 0. To *store* an element $y \in Y$ into B_Y, set the bit at index $h(y)$ to 1, i.e., $B_Y[h(y)] \coloneqq 1$. To store all of Y, simply store all the elements of Y in turn.

To query the filter B_Y with an element $x \in D$, check if the bit at index $h(x)$ is set to 1. If so, the filter produces *Maybe*. If the bit is set to 0, then $x \notin Y$ and the filter produces *No*.

In the full version of the construction, there are several hash functions $h_1, \ldots, h_k$. To store $y \in Y$, compute $h_1(y), \ldots, h_k(y)$, and set the bits of B_Y to 1 at all of those locations. Algorithm 1 provides pseudo-code for the full Bloom filter construction. To query B_Y with $x \in D$, compute $h_1(x), \ldots, h_k(x)$ and check the bits of B_Y at all of those locations. If any of the bits are set to 0, the filter produces *No* and x is rejected; otherwise, the filter produces *Maybe*. Algorithm 2 provides pseudo-code for querying a Bloom filter.

Algorithm 1 BuildBloomFilter($Y \subseteq D, n, k, h_1, \ldots, h_k$)
n is the amount of memory available to store the Bloom filter B_Y, namely $|B_Y|$
$h_1, \ldots, h_k$ are functions that map elements of D to $[0, n)$

```
Initialize all n entries of Bloom filter B_Y to 0
for each element y ∈ Y do
    for i := 1 to k do
        index := h_i(y)
        B_Y[index] := 1
    end for
end for
return B_Y
```

1. The efficiency achievable in practice is limited by the ability of current SAT solvers to find solutions to random k-SAT instances near the threshold for random k-SAT.

Algorithm 2 QueryBloomFilter($B_Y, x \in D, n, k, h_1, \ldots, h_k$)
n is the amount of memory available to store the Bloom filter B_Y, namely $|B_Y|$
$h_1, \ldots, h_k$ are functions that map elements of D to $[0, n)$

```
for i := 1 to k do
    index := h_i(x)
    if B_Y[index] = 0 then
        return No
    end if
end for
return Maybe
```

Of course, n and k must be chosen appropriately for a given m. If n is too small or k too large, all the bits of B_Y could essentially become set to 1. This means the filter will never reject when queried (i.e., the filter will have a high false positive rate), rendering B_Y useless.

3. Satisfiability Filters

This section introduces Satisfiability and SAT filters and provides algorithms for their implementation.

3.1 Satisfiability

Before introducing SAT filters, both the generic SAT instance and a specialization called a random k-SAT instance will be presented. SAT is a Knowledge Representation mechanism into which finite-domain Constraint Satisfaction Problems [6, 17] are encoded as Boolean functions, usually (but not always) expressed as Conjunctive Normal Form (CNF) formulas, or equivalently, sets of clauses.

A CNF formula is an expression of the form

$$C_1 \wedge \ldots \wedge C_m$$

where the symbol $\wedge$ represents logical conjunction (`AND`) and each C_i, $1 \leq i \leq m$, is described by a *clause*, i.e., an expression of the form

$$l_{i,1} \vee \ldots \vee l_{i,k_i},$$

where the symbol $\vee$ represents logical disjunction (`OR`) and each $l_{r,s}$ is a literal, i.e., a Boolean variable or its negation. A pair of literals is said to be *complementary* if both are the same variable but have different signs, i.e., x_i and $\overline{x_i}$ are complementary literals. Specifically,

$$\bar{l} = \begin{cases} \overline{x_i} \text{ if } l = x_i \\ x_i \text{ if } l = \overline{x_i}. \end{cases}$$

An assignment v is a function from the set of variables *Vars* into the set *Bool*, i.e., $\{0, 1\}$. An assignment v *satisfies* x_i if $v(x_i) = 1$ and v satisfies $\overline{x_i}$ if $v(x_i) = 0$. An assignment v satisfies a clause C_i if for some j, $1 \leq j \leq k_i$, $v(l_{i,j}) = 1$ and satisfies a CNF $\mathrm{C}_1 \wedge \ldots \wedge C_m$ if v satisfies all C_i, $1 \leq i \leq m$.

3.2 Random k-SAT

A clause C has width k if it has exactly k distinct literals and no pair is complementary. A random k-SAT instance is a set of clauses drawn uniformly, independently, and with replacement from the set of all width k clauses [19]. Random k-SAT instances exhibit quite regular behavior in terms of the clauses-to-variables ratio. Specifically, this ratio determines almost certainly (i.e., with high probability) the satisfiability of the set of clauses drawn [1].

A clause can be thought of as a *constraint* on a putative satisfying assignment. Therefore, a collection of clauses can be thought of as a *set* of constraints on a putative satisfying assignment. Given a random k-SAT instance $\mathcal{X}$, the strength of $\mathcal{X}$ (as a set of constraints) can be measured in terms of the ratio $\alpha_{\mathcal{X}} = \frac{m}{n}$ where $m = |\mathcal{X}|$ and $n = |\mathit{Vars}|$.[2.] Intuitively, each constraint represents the same "strength" (each only depends on its length, k). Yet, the ratio $\alpha_{\mathcal{X}}$ determines with high probability the satisfiability of $\mathcal{X}$. Specifically, given a fixed k there exists a number α_k such that whenever $\alpha_{\mathcal{X}} < \alpha_k$ then $\mathcal{X}$ is almost certainly satisfiable, and whenever $\alpha_{\mathcal{X}} > \alpha_k$ then $\mathcal{X}$ is almost certainly unsatisfiable. In [4] it was found that $\alpha_k = 2^k \ln 2 - O(k)$.

In addition to these theoretical results that prove the bound on the growth of α_k but do not provide its closed form, experimental results have established approximate values of α_k for small values of k. These values are given next and are reproduced from [23].

k	1	2	3	4	5	6	7	8	9	10
α_k	0	1	4.26	9.93	21.11	43.37	87.79	176.54	354.01	708.92

Polynomial time algorithms exist for solving random-k SAT instances with $k < 3$. For $k \geq 3$, k-SAT is NP-complete, and no polynomial time algorithms are known for random instances near the satisfiability threshold, α_k. As well, there is evidence of an algorithmic barrier, i.e., efficient algorithms for solving random k-SAT instances with $m/n \geq \frac{2^k \ln k}{k}$, for large k, are unlikely to exist [2]. But, for small k, there are algorithms (such as Survey Propagation [11]) that can solve random k-SAT instances near the satisfiability threshold. [16] provides a good survey of different algorithms for solving random k-SAT instances and the ratios up to which they succeed.

3.3 SAT Filter Construction

The SAT filter, introduced here, has two stages, building and querying. Both stages, while analogous to the Bloom filter, operate differently than the Bloom filter.

3.3.1 Building a SAT Filter

The construction of a SAT filter is one-time work that must be performed for a particular data set $Y \subseteq D$. This construction consists of three phases that involve converting each element in Y into a width k clause (where k is small), finding a solution to the set of clauses via a SAT solver, and encoding the solution in a form amenable for querying. Due to the use of a SAT solver, one disadvantage of the SAT filter over Bloom filters is that

2. The m and n used here are related to the definitions given in § 4 namely, $m = |Y|$ and n is the number of bits of long-term storage of a filter.

SAT filters do not allow insertions after they have been built, i.e., SAT filters are static.[3.] Formally, SAT filters are *offline filters*, whereas Bloom filters are *online filters* [34]. The major disadvantage of online filter constructions is that, unlike offline filter constructions, they always use more long-term storage than optimal [22].

Phase One: Building $\mathcal{X}_Y$ During this first phase, each element $y \in Y$ is used to produce a propositional clause via a set of hash functions. Specifically, the hash functions map each element of D uniformly at random into a set of literals that, when combined using disjunction, constitute a *random* width k clause. In this, the SAT filter is different from a Bloom filter (which creates *positions* where entries in an array that have value 0 are converted to a 1).

Remember that a random width k clause has exactly k distinct literals where no pair is complementary. One way to ensure that this property holds is to add a *nonce* as input to each hash function that can modify the hash functions' outputs to ensure that only width k clauses are generated. Algorithm 3 demonstrates one way to guarantee width k clauses are generated correctly.

Phase one is then: use a set of hash functions $h_1, \ldots, h_k$ to create a random width k clause C_y for each $y \in Y$. All these clauses are conjoined together, creating a CNF $\mathcal{X}_Y$, i.e., a random k-SAT instance. § 5 introduces techniques for determining values for various parameters, such as the number of variables n and number of literals per clause k, to guarantee $\mathcal{X}_Y$ will be satisfiable.

Phase Two: Finding a Satisfying Assignment for $\mathcal{X}_Y$ In this phase, a SAT solver is used to find a solution to the random k-SAT instance $\mathcal{X}_Y$ created during Phase One. The particular SAT solver used is agnostic to SAT filters, as long as the SAT solver is capable of solving moderate-to-large satisfiable random k-SAT instances.

The Final Phase: Defining the Filter Finally, the solution to $\mathcal{X}_Y$ discovered by the SAT solver is stored as an array of bits. *This array is a representation of the SAT filter.* The array of bits is stored so that index i of the array corresponds to the value (either 1 or 0) of variable x_i in the satisfying solution for $\mathcal{X}_Y$.

The clauses $\mathcal{X}_Y$ can be discarded once a solution has been found, hence the amount of long-term storage required is the number of bits needed to represent the solution, i.e., n. In general, this amount of memory is small compared to the memory required to fully specify the original data set. Algorithm 4 provides pseudo-code for building a SAT filter.

3.4 Querying a SAT Filter

Similar to querying a Bloom filter, querying a SAT filter is work that must be done every time an element is tested for membership in the original data set. Unlike the first phase (one-time work), which may be quite time consuming, the querying phase (every-time work) should run very quickly.

To determine if an element $x \in D$ is in the original data set Y, first produce a width k clause C_x from x using the same hash functions used in the construction of the SAT filter being queried. Next, if for any literal l_i in the clause, the SAT filter at index i satisfies l_i

3. Several other filter constructions, such as those described in [13, 14, 15, 24, 26, 29], are also offline filters.

Algorithm 3 ElementToClause($e \in D, n, k, h_1, \ldots, h_k$)
k is the number of literals per clause
$h_1, \ldots, h_k$ are functions that map elements of D to $[-n, n] \setminus \{0\}$

```
nonce ≔ 0
repeat
    C ≔ {}, the empty clause
    for i ≔ 1 to k do
        literal l ≔ h_i(e, nonce)
        C ≔ C ∪ {l}
    end for
    nonce ≔ nonce + 1
until all literals of C are distinct and no pair is complementary
return C
```

Algorithm 4 BuildSatFilter($Y \subseteq D, n, k, h_1, \ldots, h_k$)
n is the amount of memory available to store the SAT filter S_Y, namely $|S_Y|$
k is the number of literals per clause
$h_1, \ldots, h_k$ are functions that map elements of D to $[-n, n] \setminus \{0\}$

```
m ≔ |Y|
X_Y ≔ {}, the empty formula
for each element y ∈ Y do
    C_y ≔ ElementToClause(y, n, k, h_1, ..., h_k)
    X_Y ≔ X_Y ∪ {C_y}
end for
if the random k-SAT instance X_Y is unsatisfiable then
    return failure
else
    Let S_Y be a solution to X_Y
    return S_Y
end if
```

Algorithm 5 QuerySatFilter($S_Y, x, n, k, h_1, \ldots, h_k$)
n is the amount of memory available to store the SAT filter S_Y, namely $|S_Y|$
k is the number of literals per clause
$h_1, \ldots, h_k$ are functions that map elements of D to $[-n, n] \setminus \{0\}$

```
C_x ≔ ElementToClause(x, n, k, h_1, ..., h_k)
for each literal l ∈ C_x do
    if S_Y(l) = 1 then
        return Maybe
    end if
end for
return No
```

(i.e., either literal l_i is positive and the SAT filter at index i contains value 1, or literal l_i is negative and the SAT filter at index i contains value 0), then this phase produces *Maybe*. If no literal is satisfied then this phase produces *No*.

The rationale here is that if the clause C_x is falsified by the SAT filter associated with Y, then by construction, x is definitely not in Y. If, on the other hand, C_x is satisfied by the SAT filter, then x may or may not be in Y (a potential false positive). Algorithm 5 provides pseudo-code for querying a SAT filter.

3.5 False Positive Rate, Query Time, and Storing Multiple Solutions

It is important to consider how often a SAT filter will say *Maybe*, i.e., the false positive rate. The false positive rate of a SAT filter is the probability that the clause generated by the query is satisfied by the stored solution. This is equivalent to the probability that a random width k clause is satisfied by a random solution, i.e., $1 - 2^{-k}$. For each literal in such a clause, there is a $\frac{1}{2}$ chance that a random solution contains that literal set to 1. The false positive rate can be improved as desired by either storing multiple solutions to multiple SAT instances or storing multiple *disparate* solutions to a single SAT instance.

By generating s different SAT instances and storing one solution to each, the false positive rate is improved to $(1 - 2^{-k})^s$ because, during SAT filter query, s different clauses are generated from each element and each have to be satisfied by a different solution. It is important to note that by storing s solutions, the amount of long-term storage will be multiplied by s as well. Also, query time is increased because sk hashes are computed, although all but the first two hashes can be simple (see [21]).

Likewise, storing multiple solutions to a single SAT instance can cause the false positive rate to come out higher than expected because the solutions may not be independent, i.e., they are all solutions to the same SAT instance. The amount of increase in the false positive rate depends on how different the solutions are from one another, i.e., the greater the correlation between pairs of solutions, the higher the false positive rate will be. If this approach is to be used, the specific SAT solver used must be capable of finding many disparate solutions to the same instance.

The benefit of this second approach is that it does *not* negatively impact query time since only one clause is generated per query and that clause can be checked against all s solutions in parallel using bit-packing and word-level operations. For this to be the case, the solutions need to be stored such that all s solution bits corresponding to a variable are stored together.

To give an example set of parameters, a SAT filter with a false positive rate of $p \approx \frac{1}{4}$ and $k = 5$ needs $s = 44$ solutions to be stored. According to the results given in § 6, if $m = 2^{16}$, a SAT filter can be built that will use $sn = 145112$ bits of long-term storage, a 22% reduction over an optimal Bloom filter's long-term storage. See Tables 2, 3, and 4 for metrics on different sized data sets, efficiencies, and query times.

The next section provides the mathematics needed to choose appropriate parameters for any filter construction and to compare different filter constructions against one another.

4. Filter Efficiency

There are many different filter constructions and also several important factors to consider when choosing a filter construction. Filters may need to be constructed in a reasonable amount of time (one-time work) and querying the filter may need to be fast (every-time work). Also, the particular application making use of the filter may demand the use of an online filter. This section, however, addresses another aspect, namely, how well a filter uses the memory available to it.

There is a distinction between the filter construction algorithm and a particular filter instance output by the algorithm. Given a filter instance F, the *false positive rate* is

$$p(F) = \mathrm{P}\left[F(x) = Maybe \mid x \in D \setminus Y\right].$$

In other words, the false positive rate $p(F)$ is the probability that F passes an element erroneously. For certain inputs, a filter construction algorithm might output a specific filter instance F with a higher or lower false positive rate. To measure the quality of a filter construction, it is necessary to compute an appropriate average of the false positive rate of filter instances. For a given input set Y, let $F(Y)$ be the filter instance, and define the false positive rate of the *filter construction* under load m to be

$$p = \mathrm{P}\left[p(F(Y)) \mid Y \subseteq D, |Y| = m\right].$$

That is, the false positive rate of a filter construction is the probability that a filter instance $F(Y)$, built from a uniform random input of size m from the domain D, erroneously accepts an element from D chosen uniformly at random.

When comparing filter constructions, it is standard to assume that $|Y| \ll |D|$; that is, that the number of input elements $|Y|$ is insignificant compared to the total number of elements $|D|$. Further, it is standard to assume that elements are queried from D uniformly at random. This is the case in essentially all applications, and it simplifies the computation of the false positive rate, since the false positive rate becomes simply the *positive* rate. In addition, the memory available for each element is far less than the memory needed to represent them perfectly, which is typically assumed to be infinite. This avoids spurious degenerate situations that complicate an analysis, such as having enough memory to simply store Y, resulting in a false positive rate of 0.

If the filter is given a lot of memory, it should have a low false positive rate. So, it is necessary to measure not just the false positive rate of a filter construction, but the *efficiency*, i.e., how well a filter uses the memory available to it. As introduced in [34] (also see [26]), given a filter with false positive rate p, n bits of memory, and $m = |Y|$, the efficiency of the filter is

$$\mathcal{E} = \frac{-\log_2 p}{n/m}.$$

The numerator of $\mathcal{E}$ measures the *bits of cut-down*. For example, if the filter has a false positive rate of $1/8$, then it has 3 bits of cut-down. The denominator is the number of bits of memory available to represent each item in the filter. Intuitively, some number of bits are used to specify each $y \in Y$. For example, if there are $n = 3$ bits available to the filter and there are $m = 6$ elements of Y, the filter has half a bit of information available to store

each element y. From an information-theoretic perspective, it is reasonable to conjecture that the maximum possible cut-down of such a filter is half a bit for a false positive rate of $2^{-1/2} \approx 0.71$. In fact, this is true. In [34], the authors prove

Theorem 4.1. *For any filter,* $\mathcal{E} \leq 1$.

Proof. The proof from [34] is repeated here. For this proof, the definition of a filter needs to be abstracted somewhat because the following statements are being made about all filters. First, after applying a hash function, assume that D is the real interval $[0, 1]$, and Y is a collection of m samples from the uniform distribution. Furthermore, queries from D will come uniformly at random. After inserting all the elements of Y in F, the filter produces n bits, which could be in any one of 2^n different configurations. A subset of $[0, 1]$ is associated with each memory configuration on which the filter responds *Maybe*. Note that the subset that is produced must contain all the input elements. So, a filter construction can be thought of as a collection of subsets of $[0, 1]$, together with some algorithm that takes Y as input and produces a subset of $[0, 1]$ that contains Y. For a filter F, $A \in F$ is used to mean that A is one of the possible output subsets. If the filter gives the set A as output, then the false positive rate is $\mu(A)$ where μ denotes Lebesgue measure. Since the input Y is random, the output set A is also random. The probability of selecting A as output is denoted by $\mathrm{P}(A)$, and the filter's false positive rate is

$$p = \mathbb{E}(\mu(A)) = \sum_{A \in F} \mathrm{P}(A)\mu(A).$$

Recall that the output set A must contain Y, so the probability of the filter producing A is at most the probability that all the input elements are in A. Because the input is uniform random, $\mathrm{P}(A) \leq \mu(A)^m$. This yields

$$p \geq \sum_{A \in F} \mathrm{P}(A)^{1+1/m}.$$

There is also the constraint that $\sum_{A \in F} \mathrm{P}(A) = 1$, simply because *some* output must be selected. An easy application of the Hessian test from calculus shows that this constrained minimization problem has a minimum when the $\mathrm{P}(A)$ are all equal. Let N be the total number of sets in the filter.[4.] Then, the minimum occurs at $\mathrm{P}(A) = 1/N$ for all A. So,

$$p \geq N \left[\frac{1}{N}\right]^{1+1/m} = \left[\frac{1}{N}\right]^{1/m}.$$

This gives an upper-bound, i.e., the *information-theoretic limit* on efficiency, namely

$$\mathcal{E} = \frac{-\log_2 p}{\log_2 N/m} \leq \frac{\frac{1}{m}\log_2 N}{\log_2 N/m} = 1.$$

□

4. This is a better measure than 2^n because it allows for filter sizes that are not powers of 2.

4.1 Bloom Filter Efficiency

For a specific filter construction, it is usually straightforward to analyze efficiency. What is really being compared between filtering algorithms is the *asymptotic* efficiency, i.e., the efficiency that filters can achieve as the memory available to them becomes large.

Presented here is the efficiency of the Bloom filter. Recall the description of the algorithm from § 2; the same notation will be used here. First, compute the probability that a given bit is set. It is:

$$\mathrm{P}(\text{bit } i \text{ is set}) = 1 - \left[1 - \frac{1}{n}\right]^{km}.$$

When querying, k bits are checked; to pass, all the bits must be set[5], so

$$p = \left[1 - \left[1 - \frac{1}{n}\right]^{km}\right]^{k}$$

which gives an approximation

$$p \approx \left[1 - e^{-km/n}\right]^{k}.$$

As n gets large, the false positive rate converges to the approximation. Critically, the false positive rate converges *from above*, so an asymptotic upper bound on the efficiency can be achieved.

Plugging this into the efficiency formula,

$$\mathcal{E} = \frac{-\log_2 p}{n/m} \leq \frac{-k \log_2 \left(1 - e^{-km/n}\right)}{n/m}.$$

Using the first and second derivative, it is simple to see that the expression on the right is maximized when $m/n = \ln 2/k$, and the maximum is $\ln 2$. Therefore, Bloom filters can achieve an efficiency of at most $\ln 2$.

The mathematics of the construction of Bloom filters are well-known, and there are guidelines that allow for the selection of suitable n and k and the selection of a uniform and independent choice of hash functions $h_1, \ldots, h_k$ (see [25][Sec. 5.5]).

4.2 SAT Filter Efficiency

An asymptotic analysis of the SAT filter is performed here. For a given element $x \in D$, to pass a SAT filter comprised of the solutions to s random-k SAT instances means that at least one of the k literals in each of x's corresponding s clauses[6] is set to 1. For a random k-SAT solution, each bit is set with probability $1/2$, so the false positive rate, i.e., the probability of passing x is

$$p = (1 - 2^{-k})^{s}.$$

5. As shown in [10], the common assumption that all the bits are set independently is incorrect. This error causes the derivation below to result in a slightly lower false positive rate than achievable in practice.
6. Depending on the implementation choice in § 3.5, these may all be the same clause.

Also, recall that a SAT filter uses sn bits of long-term memory. Therefore, the efficiency of a SAT filter is

$$\begin{aligned}\mathcal{E} =& \frac{-\log_2((1-2^{-k})^s)}{sn/m} \\ =& \frac{-\log_2(1-2^{-k})}{n/m}.\end{aligned}$$

As previously described, the ratio m/n (i.e., $\alpha_{\mathcal{X}_Y}$) must be selected carefully so that the problem is solvable in practice; this is SAT-solver-dependent. However, there are theoretical bounds that determine, with high probability, whether or not a random k-SAT instance is satisfiable, or rather, that a random k-SAT filter can be built. In [4], the authors prove

Theorem 4.2. *if $m/n < 2^k \ln 2 - O(k)$, then as $k, n, m \to \infty$, a random k-SAT instance with n variables and m clauses is almost surely satisfiable, and unsatisfiable otherwise.*

That is, as k gets large, a large random k-SAT instance with $m/n < 2^k \ln 2 - O(k)$ is highly likely to be satisfiable. In order to maximize the efficiency of a SAT filter, m/n should be as large as possible while keeping the random k-SAT instance satisfiable. Theorem 4.2 gives a lower bound; hence, setting $m/n = 2^k \ln 2 - k$ gives a satisfiable problem with high probability as k grows. Plugging this into the efficiency formula produces the following estimate:

$$\begin{aligned}\mathcal{E} =& \frac{-\log_2(1-2^{-k})}{n/m} \\ \geq& -(2^k \ln 2 - k)\log_2(1-2^{-k}).\end{aligned}$$

A simple calculus argument shows that the expression on the right tends to 1 as k tends to infinity. Therefore, SAT filters can theoretically achieve the information-theoretic limit in terms of efficiency.[7.]

Since the first derivative of the SAT filter efficiency function is always positive, SAT filter efficiency is a monotonically increasing function. Thus, efficiency increases as k increases, i.e., there is not a specific value of k that will maximize SAT filter efficiency. On the positive side, there are diminishing returns as k grows, so small k (say five or six) can give near optimal efficiency (see Figure 1).

Note that the every-time work for a SAT filter is similar to that of a Bloom filter. Therefore, SAT filters can be queried quickly while achieving near perfect efficiency. Although building a SAT filter requires more work than a Bloom filter, this is typically not critical because many common applications spend far more time querying than building.

Bloom filters are very fast to construct and very fast to query, but their achieved efficiency is far from optimal. Other filter constructions, such as Compressed Bloom filters [24], achieve higher efficiency at the expense of both more one-time work and more every-time work, whereas the SAT filter retains the fast every-time work of the Bloom filter with near optimal efficiency.

7. § 6 provides results indicating currently achievable SAT filter efficiency.

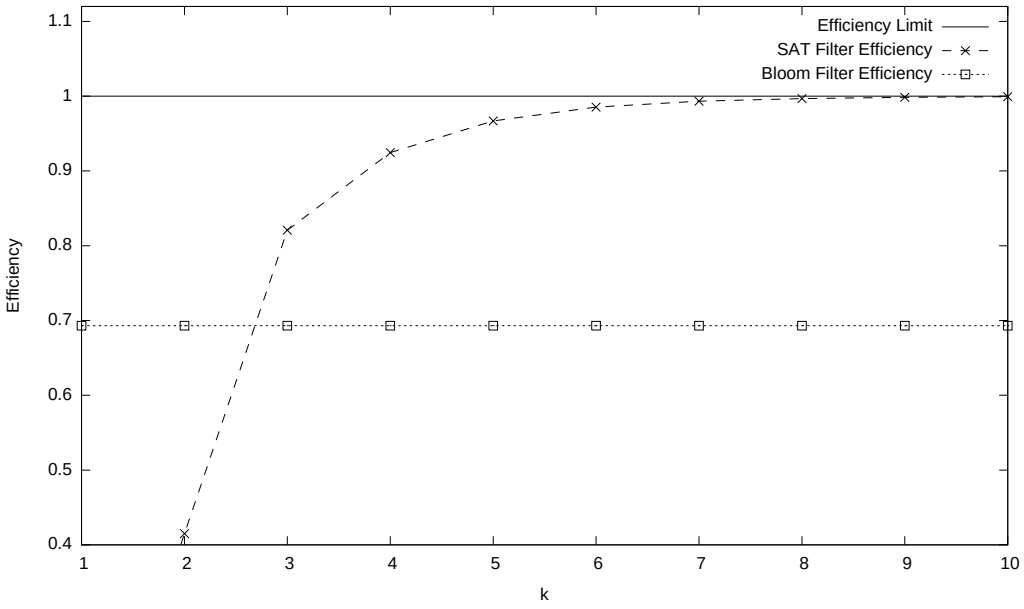

Figure 1. Theoretically Achievable SAT Filter Efficiency for Various k.

5. SAT Filter Parameters

This section discusses the selection of parameters for SAT filters (see Table 1). Definitions for each of the parameters are given first. A SAT filter uses the following parameters:

Table 1. SAT Filter Parameters.

p	the false positive rate of a SAT filter
s	the number of SAT instances per SAT filter
n	the number of variables per SAT instance
m	the number of clauses per SAT instance
k	the number of literals per clause

A value for k should be selected first. As stated earlier, a small k (say five or six) is sufficient to achieve near perfect efficiency. This is good because the time it takes to query a SAT filter increases with k.

A value for n should be selected next. The phenomenon of the random k-SAT phase transition (see § 3.2) allows n to be appropriately selected so that there is a high probability of the SAT instance being satisfiable, while keeping the size of the filter (long-term storage) as small as possible. Hence, being able to solve random satisfiable k-SAT instances right at the phase transition allows filters to be built that use a small amount of long-term storage, thus achieving high efficiency. However, the closer $\alpha_{\mathcal{X}_Y}$ is to α_k, the harder the

SAT instance is for SAT solvers to solve. As shown in § 4.2, the information-theoretic limit acts as a barrier, denying access to satisfiable random k-SAT instances that otherwise could be used to build filters with an efficiency greater than theoretically possible.

The size of the data set $m = |Y|$ is assumed to be given. Letting $\alpha_{\mathcal{X}_Y} = e\alpha_k$, where $0 < e \leq 1$ corresponds to the power of the SAT solver being used, n can be determined as follows,

$$n = \frac{m}{e\alpha_k}.$$

Finally, values for either s or p should be selected. These parameters bring about a trade-off between the amount of long-term storage (sn) and the false positive rate. The lower the false positive rate, the higher the amount of long-term storage. The lower the amount of long-term storage, the higher the false positive rate.

If the amount of long-term storage is known, p can be determined as follows,

$$p = (1 - 2^{-k})^s.$$

If, instead, the false positive rate is known, s can be determined as follows,

$$s = \left\lceil \frac{\log_2 p}{\log_2(1 - 2^{-k})} \right\rceil.$$

The amount of one-time memory needed (in bits) to build all SAT instances simultaneously for a given filter is (assuming 64-bit integers are used to represent literals)

$$64kms.$$

If enough memory is available to hold all the SAT instances, they can be solved in parallel; otherwise, they can be solved sequentially. Although the previous equation shows the memory necessary to store the clauses for all SAT instances, there is typically additional memory needed for other SAT solver data structures (such as an occurrence list or incidence graph) that can cause this number to grow by an order of magnitude while a SAT solver is at work.

6. Experimental Results

As a proof of concept, four dictionaries were built consisting of 2^{14}, 2^{15}, 2^{16}, and 2^{17} random strings. To ensure that random k-SAT instances were generated, the strings were transformed into clauses using the MurmurHash3 hash function [5] combined with a technique described in [21], namely, the set of hashes $h_i(x) = h_1(x) + ih_2(x)$ for $3 \leq i \leq k$ can be used without increasing the false positive rate of the filter.[8] Finally, SAT filters were generated for $k = 3 \ldots 6$ with $p \approx \frac{1}{4}$ and $\alpha_{\mathcal{X}_Y}$ (i.e., m/n) set according to a range of values.

Most SAT solvers have not been designed to efficiently solve random k-SAT instances because there has not been a driving application. Nevertheless, the annual SAT competition [7] has encouraged the implementation of such solvers by continuing to offer a track for random instances. One solver that performs well in the random track is Dimetheus [20],

8. Though, for single-instance SAT filters, this relation does not provide sufficient entropy to achieve small false positive rates.

a highly parameterized solver that implements Survey Propagation [11], a message passing heuristic shown to perform well on random k-SAT instances with m/n near the threshold. Unfortunately, Dimetheus does not work well (without tuning) when m/n is significantly less than the threshold. However, another SAT solver, WalkSAT [32] is good at solving random k-SAT instances when m/n is significantly less than the threshold.

Table 2 presents the time taken to build *multi-instance* SAT filters, that is, s SAT instances were generated per filter and one solution was found for each (as discussed in § 3.5). The SAT instances were solved sequentially using a dovetailing of Dimetheus and WalkSAT. Run-times of Dimetheus are in **bold** and those of WalkSAT are not.

SAT filters built in this way achieve the desired false positive rate. This was verified experimentally by querying each SAT filter with 2^{27} random elements (as in Table 4), calculating the observed false positive rate and comparing it to the desired false positive rate.

In Table 2, the symbol "-" is used to denote that the filter could not be built within 2000 seconds. The vertical bar represents the location of the random k-SAT threshold for each k. These results demonstrate that SAT filters can be built near the information-theoretic limit. Although SAT filter build times seem large (as compared to Bloom filters), achieving higher efficiency (less long-term storage) at the expense of more one-time work is often acceptable.

All results were collected using an early 2009 MacBook Pro with a 3.06-GHz Intel Core 2 Duo processor and 4 GB of RAM. All times are reported in seconds. All sizes are reported in bits.

Tables 3 and 4 provide a comparison between the sizes and query times of the smallest SAT filters built versus optimal Bloom filters built using the same data set and false positive rate and the optimal $k = 2$ hash functions. These tables show that it is possible to build SAT filters that if, for example, $p \approx \frac{1}{4}$ and $k = 5$, use 22% less long-term storage than Bloom filters and accept more than $300,000$ queries a second (although roughly 14 times slower than Bloom filters).

SAT filter query time can be decreased by either decreasing k (which may increase long-term storage, i.e., decrease efficiency) or storing multiple solutions to a *single-instance* (as discussed in § 3.5) and making use of optimization techniques such as bit-packing and word-level instructions. This second option produces SAT filters that can be queried within an order of magnitude of Bloom filters, although, currently, at the expense of more one-time work in finding disparate solutions to a single random k-SAT instance (per SAT filter). The discovery of such solutions has been achieved here by calling a SAT solver *many* times on the same SAT instance. After each call, the average Hamming distance was calculated between all pairs of solutions found up to that point. If accepting the latest solution caused the average Hamming distance to fall below some predetermined limit, a handful of k-clauses were added to the instance to block the parts (the backbone) of the non-accepted solution that were most similar to already accepted solutions.[9] This process was repeated until s disparate solutions were found.

The results of building and querying a few *single-instance* SAT filters are presented in Table 5. The table shows that SAT filters can be built that use less long-term storage than

9. The method used here is ad hoc. One avenue of future research is to discover more rigorous methods for finding many disparate solutions to a given SAT instance.

Table 2. Seconds Taken to Build *Multi-Instance* SAT Filters for $p \approx \frac{1}{4}$

$\mathcal{E}$	0.75	0.77	0.79	0.81	0.83	0.85	0.87	0.89	0.91	0.93	0.95	0.97
$k = 3,\ s = 11 \rightarrow p \approx 23.02\%$												
$m = 2^{14}$	1	1	**2**	**137**	-	-	-	-	-	-	-	-
$m = 2^{15}$	1	3	**15**	**101**	-	-	-	-	-	-	-	-
$m = 2^{16}$	2	6	**67**	**101**	-	-	-	-	-	-	-	-
$m = 2^{17}$	7	17	**116**	**120**	-	-	-	-	-	-	-	-
$k = 4,\ s = 22 \rightarrow p \approx 24.18\%$												
$m = 2^{14}$	1	1	1	1	1	2	6	**39**	-	-	-	-
$m = 2^{15}$	1	1	1	2	2	9	**19**	**62**	-	-	-	-
$m = 2^{16}$	1	2	2	4	10	**26**	**36**	**71**	-	-	-	-
$m = 2^{17}$	4	5	7	11	25	**62**	**74**	**144**	-	-	-	-
$k = 5,\ s = 44 \rightarrow p \approx 24.74\%$												
$m = 2^{14}$	1	1	1	1	4	33	**183**	**285**	**1436**	-	-	-
$m = 2^{15}$	1	1	2	4	21	**96**	**97**	**117**	**349**	-	-	-
$m = 2^{16}$	3	3	5	8	71	**211**	**219**	**222**	**548**	-	-	-
$m = 2^{17}$	7	9	14	23	**471**	**484**	**499**	**517**	**626**	-	-	-
$k = 6,\ s = 89 \rightarrow p \approx 24.62\%$												
$m = 2^{14}$	1	2	6	29	**54**	**68**	**115**	**235**	-	-	-	-
$m = 2^{15}$	3	11	105	**105**	**110**	**123**	**167**	**389**	-	-	-	-
$m = 2^{16}$	7	79	**250**	**251**	**258**	**268**	**324**	**671**	-	-	-	-
$m = 2^{17}$	21	**555**	**567**	**569**	**579**	**604**	**640**	**1283**	-	-	-	-

Table 3. Bloom Filter Size Versus SAT Filter Size (in bits). [Each pair of filters (Bloom and SAT) were built with optimal parameters given the same dictionaries and false positive rate. The parameters used to build the SAT filters are those with highest efficiency from Table 2.]

	$m = 2^{14}$		$m = 2^{15}$		$m = 2^{16}$		$m = 2^{17}$	
	Bloom	SAT	Bloom	SAT	Bloom	SAT	Bloom	SAT
$k = 3,\ s = 11 \rightarrow p \approx 23.02\%$	50090	42856	100179	85734	200356	171446	400715	342914
$k = 4,\ s = 22 \rightarrow p \approx 24.18\%$	48419	37708	96837	75416	193674	150832	387348	301664
$k = 5,\ s = 44 \rightarrow p \approx 24.74\%$	47638	36256	95275	72556	190550	145112	381099	290268
$k = 6,\ s = 89 \rightarrow p \approx 24.62\%$	47797	37202	95593	74404	191186	148897	382371	297794

Table 4. Seconds Taken to Query the Bloom and SAT Filters from Table 3 with 2^{27} Random Elements.

	$m = 2^{14}$		$m = 2^{15}$		$m = 2^{16}$		$m = 2^{17}$	
	Bloom	SAT	Bloom	SAT	Bloom	SAT	Bloom	SAT
$k = 3$, $s = 11 \rightarrow p \approx 23.02\%$	30	84	30	84	30	84	30	85
$k = 4$, $s = 22 \rightarrow p \approx 24.18\%$	31	186	31	186	31	185	31	185
$k = 5$, $s = 44 \rightarrow p \approx 24.74\%$	31	415	31	410	31	401	31	421
$k = 6$, $s = 89 \rightarrow p \approx 24.62\%$	31	1034	31	1031	31	1028	31	1011

Bloom filters and also have comparable query time and false positive rates. These results also provide evidence that to achieve the desired false positive rate, the average Hamming distance of the set of solutions found needs to be at least 50% of the total number of variables.

Table 5. Build Time, Filter Size, Query Time, and Achieved False Positive Rate for *Single-Instance* SAT Filters Built with $\mathcal{E} = 0.75$ and $m = 2^{14}$. [Two sets of filters were built, one where the average Hamming distance of the set of s solutions (taken pairwise) was at least 50% of the total number of variables, and one with at least 49%. Each filter was queried with 2^{27} random elements (as in Table 4).]

	Build Time	Size	Query Time	Achieved p
Hamming Distance = 50%				
$k = 4$, $s = 22 \rightarrow p \approx 24.18\%$	20802	44748	47	24.20%
$k = 5$, $s = 44 \rightarrow p \approx 24.74\%$	610	44000	51	24.86%
$k = 6$, $s = 89 \rightarrow p \approx 24.62\%$	643	44144	61	25.09%
Hamming Distance = 49%				
$k = 4$, $s = 22 \rightarrow p \approx 24.18\%$	18	44748	47	26.14%
$k = 5$, $s = 44 \rightarrow p \approx 24.74\%$	14	44000	51	27.33%
$k = 6$, $s = 89 \rightarrow p \approx 24.62\%$	10	44144	61	28.16%

7. Conclusions and Future Work

A SAT filter is a simple offline filter construction that is efficient in terms of both the amount of long-term storage and query time, i.e., SAT filters asymptotically achieve the information-theoretic limit and support fast queries. This filter can be used effectively for testing set membership in large families of sets, providing significant improvements over current techniques such as the standard Bloom filter, as well as its compressed version [24].

The SAT filters described here are more appropriately called random k-SAT filters. They provide a novel application of random k-SAT along with new insights into the random k-SAT phase transition. To the best of the authors' knowledge, SAT filters are the first

instance of using random k-SAT for a real-world application, and hence a good instance of theory supporting practice. These investigations show that the study of random k-SAT is no longer "purely academic." A renewed research effort into random SAT could help create SAT solvers capable of solving instances much nearer to the k-SAT threshold than currently achievable. This would enable more efficient SAT filters to be built, saving both time and memory for the many applications making use of filters.

There is potential for this work to be generalized and applied to other domains where random phase transitions exist. More SAT filter constructions could be developed that use different ways of generating random clauses [28] or that use other kinds of constraints. For example, pairs of randomly chosen solutions to random Not-All-Equal (NAE) k-SAT instances are uncorrelated [3] (not true for random k-SAT [27, 4]), and so single-instance SAT filters may be easier to build using width k *NAE* constraints. Also, random k-XORSAT instances are solvable in polynomial time [31], and so SAT filters could be built without solving an instance of an NP-complete problem.[10.]

Acknowledgments

The authors would like to thank Ian Blumenfeld, John Franco, Marijn Heule, Eric I. Hsu, Donna Smith, and Christian Svetnicka for many engaging discussions on the subject of random k-SAT and set membership filters.

References

[1] Dimitris Achlioptas. Random satisfiability. In Biere et al. [8], page 245.

[2] Dimitris Achlioptas and Amin Coja-Oghlan. Algorithmic barriers from phase transitions. In *Foundations of Computer Science, 2008. FOCS'08. IEEE 49th Annual IEEE Symposium on*, pages 793–802. IEEE, 2008.

[3] Dimitris Achlioptas and Cristopher Moore. The asymptotic order of the random k-sat threshold. In *Foundations of Computer Science, 2002. Proceedings. The 43rd Annual IEEE Symposium on*, pages 779–788. IEEE, 2002.

[4] Dimitris Achlioptas and Yuval Peres. The threshold for random k-SAT is $2^k log 2 - O(k)$. *Journal of the American Mathematical Society*, **17**(4):947–973, 2004.

[5] Austin Appleby. Murmurhash3. http://code.google.com/p/smhasher/wiki/MurmurHash3, 2011.

[6] Krzysztof R. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.

[7] Daniel Le Berre, Olivier Roussel, and Laurent Simon. SAT competition. http://www.satcompetition.org/, 2013.

10. Connections between Bloom filters and k-XORSAT have been made before, specifically, Cuckoo Hashing (a technique related to Bloom filters), shares the same thresholds as k-XORSAT [18] for all values of $k > 2$.

[8] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, **185** of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[9] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, **13**(7):422–426, 1970.

[10] Prosenjit Bose, Hua Guo, Evangelos Kranakis, Anil Maheshwari, Pat Morin, Jason Morrison, Michiel Smid, and Yihui Tang. On the false-positive rate of Bloom filters. *Information Processing Letters*, **108**(4):210–213, 2008.

[11] Alfredo Braunstein, Marc Mézard, and Riccardo Zecchina. Survey propagation: An algorithm for satisfiability. *Random Structures & Algorithms*, **27**(2):201–226, 2005.

[12] Andrei Broder and Michael Mitzenmacher. Network applications of Bloom filters: A survey. *Internet Mathematics*, **1**(4):485–509, 2004.

[13] Andrej Brodnik and J. Ian Munro. Membership in constant time and almost-minimum space. *SIAM Journal on Computing*, **28**(5):1627–1640, 1999.

[14] Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and Ayellet Tal. The bloomier filter: an efficient data structure for static support lookup tables. In *Proceedings of the fifteenth annual ACM-SIAM Symposium on Discrete Algorithms*, pages 30–39. Society for Industrial and Applied Mathematics, 2004.

[15] Saar Cohen and Yossi Matias. Spectral Bloom filters. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 241–252. ACM, 2003.

[16] Amin Coja-Oghlan. A better algorithm for random k-sat. *SIAM Journal on Computing*, **39**(7):2823–2864, 2010.

[17] Rina Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

[18] Martin Dietzfelbinger, Andreas Goerdt, Michael Mitzenmacher, Andrea Montanari, Rasmus Pagh, and Michael Rink. Tight thresholds for cuckoo hashing via XORSAT. In *Automata, Languages and Programming*, pages 213–225. Springer, 2010.

[19] John Franco and Marvin Paull. Probabilistic analysis of the Davis Putnam procedure for solving the satisfiability problem. *Discrete Applied Mathematics*, **5**(1):77–87, 1983.

[20] Oliver Gableske. Solver description of dimetheus v. 1.700 for the SAT competition 2013. *Proceedings of SAT Competition 2013*, page 30, 2013.

[21] Adam Kirsch and Michael Mitzenmacher. Less hashing, same performance: building a better Bloom filter. In *Algorithms–ESA 2006*, pages 456–467. Springer, 2006.

[22] Shachar Lovett and Ely Porat. A lower bound for dynamic approximate membership data structures. In *Foundations of Computer Science (FOCS), 2010 51st Annual IEEE Symposium on*, pages 797–804. IEEE, 2010.

[23] Stephan Mertens, Marc Mézard, and Riccardo Zecchina. Threshold values of random k-SAT from the cavity method. *Random Structures & Algorithms*, **28**(3):340–373, 2006.

[24] Michael Mitzenmacher. Compressed Bloom filters. *IEEE/ACM Transactions on Networking (TON)*, **10**(5):604–612, 2002.

[25] Michael Mitzenmacher and Eli Upfal. *Probability and computing: Randomized algorithms and probabilistic analysis*. Cambridge University Press, 2005.

[26] Anna Pagh, Rasmus Pagh, and S. Srinivasa Rao. An optimal Bloom filter replacement. In *Proceedings of the sixteenth annual ACM-SIAM Symposium on Discrete Algorithms*, pages 823–829. Society for Industrial and Applied Mathematics, 2005.

[27] Andrew J. Parkes. Clustering at the phase transition. In *In Proc. of the 14th Nat. Conf. on AI*, pages 340–345. AAAI Press / The MIT Press, 1997.

[28] Will Perkins. Random k-SAT and the power of two choices. *arXiv preprint arXiv:1209.5313*, 2012.

[29] Ely Porat. An optimal bloom filter replacement based on matrix solving. In *Computer Science-Theory and Applications*, pages 263–273. Springer, 2009.

[30] Lukas Radvilavicius, Vaidotas Marozas, and Antanas Cenys. Overview of real-time antivirus scanning engines. *Journal of Engineering Science and Technology Review*, **5**(1):63–71, 2012.

[31] Thomas J Schaefer. The complexity of satisfiability problems. In *STOC*, **78**, pages 216–226, 1978.

[32] Bart Selman, Henry Kautz, and Bram Cohen. Local search strategies for satisfiability testing. *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*, **26**:521–532, 1993.

[33] Sasu Tarkoma, Christian Esteve Rothenberg, and Eemil Lagerspetz. Theory and practice of Bloom filters for distributed systems. *Communications Surveys & Tutorials, IEEE*, **14**(1):131–155, 2012.

[34] Alden Walker. Filters. Undergraduate thesis, Haverford College, 370 W Lancaster Ave, Haverford, PA 19041, 2007.