Approximate Membership of Sets: A Survey

Elias Szabo-Wexler
szabowexler@cmu.edu
Carnegie Mellon University

***Abstract*—The task of representing a set so as to support membership queries is a common one in computer science. If space is no object, a complete dictionary may provide accurate QUERYing in good runtime. If space is at a premium and the size of the set uncertain, Bloom filters and hash compaction provide good approximations. If the size is completely unknown, a modified Bloom filter provides very good runtime, with minimal extra error. This survey briefly explores the applications of dictionaries, Bloom filters, and hash compaction techniques to the set membership problem, observing details of their implementations and tradeoffs in their accuracy. We conclude with a recommendation for when to use each, respectively.**

***Index Terms*—Data structures, Bloom filters, Bloom-g filters, hash compaction, membership testing, on-line algorithms, dynamic set approximation**

I. Introduction

THE question of determining membership shows up commonly in algorithmic design, in topics as diverse as probabilistic verification [1], network protocol analysis [2], routing-table lookup, online traffic measurement, peer-to-peer systems, cooperative caching, firewall design, intrusion detection, bioinformatics, database QUERY processing, stream computing, and distributed storage systems [3]. Informally, we are frequently concerned with answering questions of the form "Is element x a member of set $\mathcal{S}$"? In the *static* variant, the size of the set $\mathcal{S}$ is specified ahead of time, while in the *dynamic* variant, the elements of the set are specified one by one, in an on-line fashion. Obviously, answering the question correctly all of the time with good runtime is optimal from an accuracy perspective, motivating the classical dictionary approach, however it comes with a trade-off in memory utilization. Namely, we note the memory required by a classical dictionary grows linearly in the size of the set. For such static problems as verification, where the phenomenon of state explosion yields sets of tremendous size, or the dynamic variant on large data sets, even linear memory usage may be too much [1].

In the static variant, if we wish to optimize for the smallest memory footprint or fastest runtime, we are advised to employ a Bloom-g filter [3] (defined below). If instead we wish to optimize for accuracy, then an exhaustive dictionary is the clear choice. Hash compaction [4] provides a decent middle ground, when a tradeoff between the two is desired.

In the dynamic variant, Qiao shows in [3] that the Bloom-g filter may still perform well even in the dynamic setting, offering a good default.

In this paper, we provide a collection of applications of the set membership problem. With these as motivation, we then explore the static and dynamic variants of the problem, surveying the three major algorithmic techniques and providing a high level algorithmic description of each, along with pertinent characterization descriptors, including runtime and memory footprint.

A. Motivation

In Broder and Mitzenmacher's seminal survey [2] of applications of Bloom filters, several problems based on the set approximation problem are discussed. All of these instances employ Bloom filters, but in each such case there is a tradeoff, and any of the three techniques enumerated above might suffice. Historically, UNIX spell checkers utilized Bloom filters [2]. In the field of networking, the Summary Cache protocol proposed by [5] utilized Bloom filters for web cache sharing. Bloom filters were also applied by several different authors [2] to locate resources in a peer-to-peer network, by [6] for resource routing in a resource discovery service, and by [7] for implementing IP traceback, to trace the route of a packet through a network.

Hash compaction has been applied to explicit state verification , as well as explicit state space exploration [8]–[10], and model checking for verification of software and hardware [10].

Together, these represent a broad host of applications which require the ability to perform set membership queries, both in static and dynamic contexts. The ability to manage the tradeoff between memory utilization and accuracy is invaluable, as modern applications deal with data sets of ever increasing size. We are well motivated to explore alternatives which approach optimality in accuracy and requisite memory as well.

B. Notation

In this paper, we discuss the *set approximation problem*, namely the task of checking if some element x is contained in the set $\mathcal{S}$. We denote the universe from which elements are drawn by $\mathcal{U}$, and assume that $|\mathcal{S}| \ll |\mathcal{U}|$, that is, that the universe is much larger than the set itself. In the static variant, the size of the set is paramaterized by $n \in \mathbb{N}$, where $|\mathcal{S}| = n$. In the dynamic variant, the size of the set is unknown, but the elements of $\mathcal{S}$ are all drawn from some universe $\mathcal{U}$. We paramaterize the problem by $u \in \mathbb{N}$, where $u = |\mathcal{U}|$ and $0 < \epsilon < 1$, where ϵ is the accuracy of a solution, which is the probability of a "false positive," namely the probability that an element $y \notin \mathcal{S}$ should be reported to be an element of $\mathcal{S}$.

We denote the probability of an event E occurring by $\Pr[E]$. Similarly, we denote the expected value of a random variable X by $\mathbb{E}[X]$. We use $\mathbb{I}_X$ for the indicator variable for the event E. Finally, we use $\log$ for the base two logarithm. Therefore,

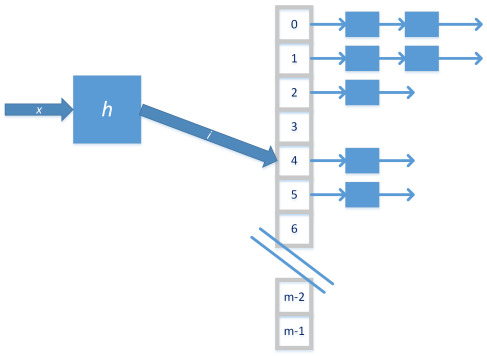

Fig. 1. x is hashed via $h(x)$ to an index i in the table, for both insertion and QUERYing. In the exhaustive hash table dictionary, elements of $\mathcal{S}$ are stored directly in the table, so each block stored is the actual element itself, contributing an overall $\Theta(n)$ term in memory utilization.

unless otherwise stated, $\log x = \log_2 x$.

We indicate scalars via normal typeface, e.g. in the numeral 5, or the variable x. We indicate vectors via bold typeface, as in the vector $\mathbf{x}$.

II. Implementations

We begin by offering an overview of the differing algorithms, along with their associated memory and runtime costs. Our algorithms provide both INSERT and QUERY operations.

A. Complete Dictionary

A dictionary is a complete mapping or enumeration of elements in S. A common implementation is the hash table - elements in the set are stored in some distribution of m buckets or slots indexed via a hash function, $h : \mathcal{U} \to [m]$, which is used both for storage and retrieval. The table is therefore an array of buckets, each of which may contain zero, one, or more elements in the set. More formally, a dictionary is defined by a hash function h, the number of buckets, m, and a collision resolution scheme. This can be visualized as in Figure 1, above.

The hash table function INSERT(x) is defined by:

1) Calculate bucket index $0 \leq i = h(x) < m$
2) Update slot/bucket at index i in table with x', utilizing resolution scheme if required

The hash table QUERY(x) is similarly defined as:

1) Calculate bucket index $0 \leq i = h(x) < m$
2) Search slot/bucket at index i in the table for x'. If present, report $x \in \mathcal{S}$, else report $x \notin \mathcal{S}$

From [11] we have that the memory utilized by a dictionary representing a set of size n taken from the universe $\mathcal{U}$ of size u is bounded by $\log \binom{u}{n} = n \log(u/n) + \Theta(n)$ bits of space. Utilizing a hash table to implement a dictionary, we may perform QUERY in $\Theta(n)$ worst case, and both QUERY and INSERT in $O(1)$ expected time [12], with $\epsilon = 0$ (i.e. with no errors). However, the hidden constants may be large, since many accesses to disk may be made if the elements stored are large. In this case, despite the good expected runtime, the hard drive accesses may become a bottleneck. Since real machines are built with a hierarchical memory structure, and layered caches, these bottlenecks translate to real world slow downs. If they can be minimized, a minimal memory footprint is practically translated into a runtime boost.

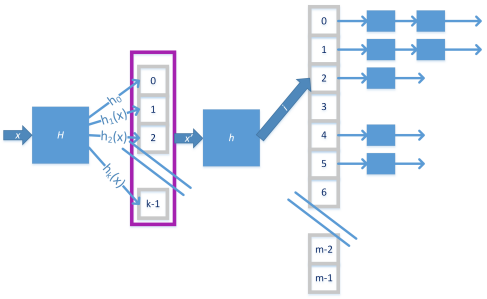

Fig. 2. x is hashed via $H(x)$ to a new fixed-width representation, $x' = H(x)$ of length $|x'| = k$. Then, the table hashing function is applied to generate the index $i = h(x')$, and the resulting slot or bucket is utilized for insertion or QUERY.

B. Hash Compaction

Hash compaction, introduced by Wolper and Leroy in [4], attempts to build on the concept of a hash table by addressing the issue of large element size. The hash table stores every element in its entirety, resulting in a linear relationship between the size of the table and the elements inside. For a large number of large elements, this is prohibitively expensive. To resolve the issue, we may use the hash compaction scheme to store a compressed version of each element in the table, commonly resulting in a ten-fold decrease in space utilization [8]. Hash compaction uses a hashing function, $H : \mathcal{U} \mapsto \{0,1\}^l$, where l is the number of bits used for the compressed representation [9], to generate a compressed representation from each element. Thus, we store only the compressed version, forcing the table to have a smaller footprint. The revised hash table is diagrammed in Figure 2, above.

The algorithms for INSERT and QUERY do not change overly much. The hash compacted INSERT(x) is:

1) Compute hash compacted element $x' = H(x)$
2) Calculate bucket index $0 \leq i = h(x') < m$
3) Update slot/bucket at index i in table with x', utilizing resolution scheme if required

The hash compacted QUERY(x) is similarly defined as:

1) Compute hash compacted element $x' = H(x)$
2) Calculate bucket index $0 \leq i = h(x) < m$
3) Search slot/bucket at index i in the table for x'. If present, report $x \in \mathcal{S}$, else report $x \notin \mathcal{S}$

The new parameter $k \in \mathbb{N}$ offers a tradeoff between memory footprint and accuracy. Observe that if $k = \log n$ and the hashing function is uniformly random, then in expectation the hash compacted algorithm will behave exactly as an exhaustive dictionary would. However, if $k < \log n$, then multiple elements must map to identical compacted elements (i.e. there is overlap). This means that the hash compacted version may occasionally erroneously report membership, even when the queried element is not in the set. The chance of this occurring is parameterized by $0 < \epsilon < 1$, i.e. the probability that an input will receive a false positive (though the choice of ϵ fixes

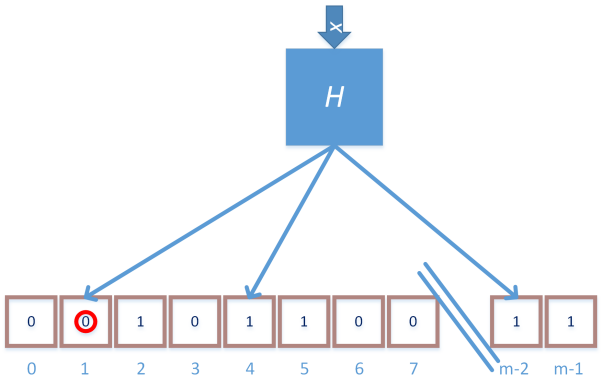

Fig. 3. Basic Bloom filter: x is hashed via $H(x)$ to a vector of indices, $\mathbf{I} = H(x)$ of length $|I| = m$. If all bits at those indices are asserted, x is in the set, otherwise it is not. Note that there is a circled 0 at index 1, so in this example either $\text{QUERY}(x) = x \notin \mathcal{S}$, or x is being inserted for the first time.

bounds on the choice of k). Observe that this is bounded by the probability that a given input hashes to the same representative vector. This is given by $\Pr[H(x_1) = H(x_2)]$. Assuming that H represents k universal hash functions, this is bounded by $\frac{1}{2^k}$ (where 2^k represents the number of possible hashed vectors). Thus, $\epsilon \leq \frac{1}{2^k}$ [13]. The tradeoff is simple - as k grows, ϵ drops rapidly, as the memory utilizing slowly increases. An optimal tuning is one where k pushes ϵ low enough for a given application, and no is no larger, to be efficient with memory.

We note that this tradeoff comes at no asymptotic change in runtime. The compression is done in constant time, and the table lookup/insertion remains asymptotically the same. Thus, the only difference between hash compaction and the exhaustive dictionary is the memory utilization, and the chance for false positives. In the next section, we explore another alternative to the exhaustive dictionary which is asymptotically faster.

C. Bloom Filter

The Bloom filter, introduced by Bloom in [14], presents an even lighter weight alternative than the method of hash compaction. Rather than utilizing a table to store a representation of each element in $\mathcal{S}$, we instead maintain a large array $\mathbf{A}$, with $|\mathbf{A}| = m$ bits. Elements are hashed to form a fixed-width vector of *indices*, which in turn index bits in the array. An element is present in the set if and only if every bit indexed by its hash is asserted. More formally, an input $x \in \mathcal{U}$ is hashed via a set of k hashing functions $H(x)$ to produce an index vector, $\mathbf{I}$, and we have that $x \in \mathcal{S}$ if and only if for each $i \in \mathbf{I}$, $A_i = 1$.

This suggests very natural algorithms for INSERT and QUERY. Namely, the Bloom filter $\text{INSERT}(x)$ is defined by:

1) Compute index vector, $\mathbf{I} = H(x)$
2) Set each $A_i = 1$, for all $i \in \mathbf{I}$

It is matched by the Bloom filter $\text{QUERY}(x)$, defined by:

1) Compute index vector, $\mathbf{I} = H(x)$
2) Check if $A_i = 0$, for any $i \in \mathbf{I}$. If yes, report $x \notin \mathcal{S}$, otherwise report $x \in \mathcal{S}$

Clearly there are significant savings in terms of memory and throughput. Namely, we need only store m bits to categorize the entire set, rather than the entire table. Further, assuming our hash functions can be computed in constant time, then an insertion or QUERY may be performed in constant time, i.e. $O(k)$. Note that for a given set of size n, and a filter of size m, the optimal choice of k is forced (and by extension of ϵ), and that both are fixed constants.

We proceed as in [2] in our analysis. Consider the vector $\mathbf{A}$ after all the elements of $\mathcal{S}$ are inserted, and recall that $|\mathbf{A}| = m, |\mathcal{S}| = n$, and we have k hashing functions. Then the probability that any bit is a 0, assuming uniformly distributed and independent hashing functions, is the probability that an element of $\mathcal{S}$ does not hash to a given bit index i, taken over all of the n elements, and each of the k hashing functions. This is given by

$$\left(1 - \frac{1}{m}\right)^{kn} \approx e^{-kn/m}$$

Letting $p = e^{-kn/m}$, the probability of a false positive is then

$$\left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k = (1-p)^k$$

Analysis then reveals that the false positive rate is *minimized globally* by a particular selection of k, namely

$$k = \ln 2 \cdot \frac{m}{n}$$

In this case, the false positive rate is

$$\epsilon = \left(\frac{1}{2}\right)^k = (0.6185)^{\frac{m}{n}}$$

Of particular note here is that we may minimize the false positive rate only when we know ahead of time the size of the set $\mathcal{S}$. When that is not known, it is impossible to minimize the error in this way. In the next section, we consider modifying the Bloom filter to make fewer memory accesses (i.e. lowering the hidden constant factor).

D. Bloom-g Filters

The Bloom-g filter, proposed by Qiao, is a variant on the conventional Bloom filter which bounds the number of words which must be fetched from memory for a QUERY or insertion, at no cost in additional error. A *Bloom-g* filter maps each member x to g words, instead of one, and spreads its k membership bits evenly in the g words [3]. To understand why this arrangement may be superior, consider an instance of a Bloom filter, f, with m bits in the filter and k membership bits per element. If $m \gg k$, then in the worst case for a QUERY we may need to fetch k words from memory, or write k words to memory. This operation may present a bottleneck, since reading from and writing to hard disk are much slower than processing cached data. In the Bloom-g filter, we map x to g words, which must be fetched from memory, and then map x to its k membership bits within those words. In the worst case, we must fetch or write at most g words from memory, and we may set $g \ll k$. Since the optimal number of membership bits

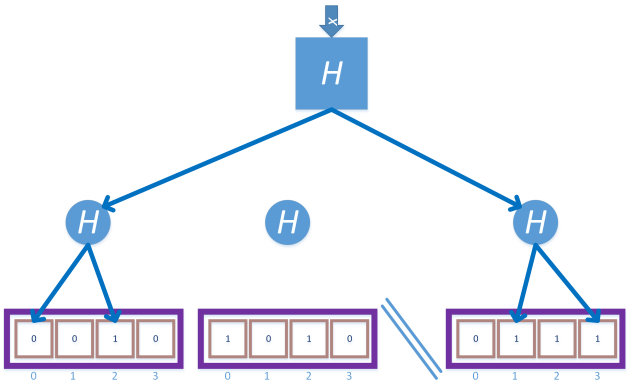

Fig. 4. Bloom-g filter, with l words, $g = 2$, and $k = 4$: x is hashed via $H(x)$ to a vector of g words. Then, x is hashed to a set of k indices, distributed evenly among the g words. If all bits at those indices are asserted, x is in the set, otherwise it is not. In this example, x is not a member of the set.

k is a function of the load factor of the filter (i.e. more entries stored requires more membership bits to maintain a good error rate), the worst case number of memory reads or writes for a regular Bloom filter grows linearly as the filter expands. With the Bloom-g filter, it is bounded by a constant, representing a significant improvement [3]. Qiao shows that the error for the Bloom-g filter is a bit more complex than for the generic Bloom filter. In the trivial case, the Bloom-k filter (i.e. the Bloom-g filter with one bit per word) behaves exactly as the Bloom filter. However, when $g \neq k$, the error is more complex to analyze. We proceed with a brief analysis, as in [3].

Consider a Bloom-g filter, with g words per element, a total of l words, each with w bits in them, and n members. Consider an arbitrary word D in the array, and let X be the number of times this word is selected as an encoded word during the filter setup. When an element selects a word to encode membership in, D has a $\frac{1}{l}$ probability of being selected. Let c be a constant in the range $[0, gn]$. Then,

$$\Pr[X = c] = \binom{gn}{c} \left(\frac{1}{l}\right)^c \left(1 - \frac{1}{l}\right)^{gn-c}.$$

We may utilize this fact to analyze the likelihood of a nonmember x' being labeled a member, by analyzing the probability that a single membership word reports x' a member, and taking this probability over all membership words. The result is that the probability of a false positive, ϵ, is given by

$$\left[\sum_{c=0}^{gn} \left(\binom{gn}{c} \left(\frac{1}{l}\right)^c \left(1 - \frac{1}{l}\right)^{gn-c} \left(1 - \left(1 - \frac{1}{w}\right)^{c\frac{k}{g}}\right)^{\frac{k}{g}}\right)\right]^g$$

Experimentally, we find that Bloom-g filters require fewer membership bits to match Bloom filters for the same error with a given load factor. For complete results, and a full derivation, see [3].

However, we may also utilize Bloom-g filters in a dynamic environment. We observe that the number of memory accesses is independent of the number of membership bits, so even as we increase the count of membership bits, we may have constant lookup costs. In order to ensure good results, we may cap the the number of elements in the filter (i.e. bound the load factor). This would ensure that we may report accurately to within some threshold, for a given number of inputs. If we know in advance the order of magnitude we will be handling, we may choose decent values for the size of the filter and its load factor, to ensure good access behavior. If the input is unbounded (i.e. we have no idea), we may set it high to start, and expand the filter dynamically if we rise above our load threshold. In this case, we would have to throw out the existing filter and reinitialize it. If the input set is large enough, this would not be an issue, though we may lose some data this way. In the next section, we move away from the hash table representation, and briefly commend on some more sophisticated constructions.

E. Advanced Techniques

We have discussed complete dictionaries, hash compaction, basic Bloom filters, and Bloom-g filters. There are other, yet more sophisticated constructions, which are also worth considering, though beyond the scope of this survey.

Naor proposes a sliding Bloom filter in [15], which builds a Bloom filter to approximate only the most recent m elements in an m-element window on an unbounded stream. The technique abandons elements seen in the distant past, but for applications which may desire only recently seen elements (e.g. a least recently used cache eviction protocol), this may actually be advantageous. In this case, Naor's filter has $O(1)$ QUERY and INSERT in the worst case with high probability, and space consumption of $(1 + o(1))(n \log \frac{1}{\epsilon} + n \cdot \max\{\log \frac{n}{m}, \log\log \frac{1}{\epsilon}\})$ bits, with a window size of n elements, slackness parameter of m, and error of ϵ.

Pagh proposes a dynamic dictionary based approximation construction in [11]. He utilizes techniques to modify existing dynamic dictionary constructions for the approximation problem, and then utilizes a clever scheme of representing a subset of elements from the input to answer QUERY in constant time in the worst case with high probability, and INSERT in constant time in the worse case with high probability, utilizing $(1 + o(1))n \log(\frac{1}{\epsilon}) + O(n \log\log n)$ bits of space.

Cohen proposes a spectral Bloom filter extending Bloom filters to multisets in [16]. The result is a generalized Bloom filter, with query time in $\Theta(\log(\frac{1}{\epsilon}))$.

Chazelle proposes the Bloomier filter in [17], associating additional satellite data with elements in the filter. The result is a filter within a constant factor of optimal in space usage, when utilized on a static set. The lookup time is constant, assuming access to truly random hash functions.

III. Fixed Size Approximation

We briefly review those constructions best suited to approximating a set of a fixed size, i.e. a set $\mathcal{S}$ for which $|\mathcal{S}| = n$ is known in advance. We considered the complete hash-table backed dictionary for perfect set representation, which offers QUERY and INSERT in $O(1)$ expected time, requires

space in $\Theta(n)$, and ensures $\epsilon = 0$. We then considered the hash compaction hash-table for approximate set representation, which offers QUERY and INSERT in $O(1)$ expected time, requires space in $\Theta(n)$ with a smaller constant factor than the complete hash-table, and has probability of false positive $\epsilon \leq \frac{1}{2^k}$. We concluded with the Bloom filter, which offers QUERY and INSERT in $O(1)$ worst time, requires space in $\Theta(1)$ (though the constant may be very large), and has the minimized probability of false positive $\epsilon = (0.6185)^{\frac{m}{n}}$.

Clearly, when perfect accuracy is required, some form of complete dictionary must be utilized. In this case, we would use the hash-table backed dictionary. If perfect accuracy is not absolutely required, then we choose between the Bloom filter and the hash compaction hash-table. Here, we consider both accuracy and runtime. If runtime is most important, we are well advised to use the Bloom filter, as operations are performed in constant worst case time, rather than amortized constant time. If memory is of utmost importance, we are again advised to use the Bloom filter, as we may fix the memory utilization directly, by selecting the size of the representing bit vector.

IV. Dynamic Size Approximation

In the dynamic case where $|\mathcal{S}|$ is not known in advance, the set of constructions best suited is somewhat different. The complete hash-table remains a viable alternative, but suffers the same linear dependence on the size of the set, and polylogarithmic rather than constant QUERY and INSERT. We considered the Bloom$-g$ filter, a modified variant of the Bloom filter. The Bloom-g filter offers QUERY and INSERT in constant time in the worst case, with a fixed number of memory accesses per QUERY. The Bloom-g filter may be adapted to the dynamic environment by imposing a limit on the load factor (i.e. ratio of storage bits to elements stored). If it grows too high, more bits may be allocated, and the filter reinitialized. In this manner, it is possible to bound the error, and maintain good runtime with a small footprint.

V. Conclusion

In this paper, we surveyed the problem of set approximation. We considered hash-table complete dictionaries, hash compacted hash-table dictionaries, Bloom filters, and Bloom-g filters. We analyzed these data structures on three axes: memory footprint, runtime, and accuracy. With these as our core metrics, we made recommendations for the best choice in each of the static and dynamic variants of the set approximation problem. Finally, we recommended some more sophisticated constructions which are theoretically interesting.

While there is great design freedom, we are generally advised to utilize the hash-table complete dictionary to address any problem in which perfect accuracy is required. Otherwise, the Bloom-g filter provides very good performance for the static or dynamic variant.

Acknowledgment

The author would like to thank Asa Frank, for providing meaningful guidance in the preparation of this survey and for conversations about the topics, and Lenore Blum, for instructing the Formal Languages, Automata, and Computation course at Carnegie Mellon University and for providing the impetus to write this paper.

References

[1] P. Dillinger and P. Manolios, "Bloom filters in probabilistic verification," in *Formal Methods in Computer-Aided Design*, ser. Lecture Notes in Computer Science, A. Hu and A. Martin, Eds. Springer Berlin Heidelberg, 2004, vol. 3312, pp. 367–381. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-30494-4_26

[2] A. Broder and M. Mitzenmacher, "Network applications of bloom filters: A survey," *Internet Mathematics*, vol. 1, no. 4, pp. 485–509, 2004. [Online]. Available: http://www.tandfonline.com/doi/abs/10.1080/15427951.2004.10129096

[3] Y. Qiao, T. Li, and S. Chen, "Fast bloom filters and their generalization," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 25, no. 1, pp. 93–103, Jan 2014.

[4] P. Wolper and D. Leroy, "Reliable hashing without collision detection," in *IN COMPUTER AIDED VERIFICATION. 5TH INTERNATIONAL CONFERENCE*. Springer-Verlag, 1993, pp. 59–70.

[5] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: A scalable wide-area web cache sharing protocol," *IEEE/ACM Trans. Netw.*, vol. 8, no. 3, pp. 281–293, Jun. 2000. [Online]. Available: http://dx.doi.org/10.1109/90.851975

[6] S. E. Czerwinski, B. Y. Zhao, T. D. Hodes, A. D. Joseph, and R. H. Katz, "An architecture for a secure service discovery service," 1999, pp. 24–35.

[7] A. C. Snoeren, C. Partridge, L. A. Sanchez, C. E. Jones, F. Tchakountio, S. T. Kent, and W. T. Strayer, "Hash-based ip traceback," *SIGCOMM Comput. Commun. Rev.*, vol. 31, no. 4, pp. 3–14, Aug. 2001. [Online]. Available: http://doi.acm.org/10.1145/964723.383060

[8] U. Stern and D. L. Dill, "Combining state space caching and hash compaction," in *In Methoden des Entwurfs und der Verifikation digitaler Systeme, 4. GI/ITG/GME Workshop*. Shaker Verlag, 1996, pp. 81–90.

[9] M. Westergaard, L. Kristensen, G. Brodal, and L. Arge, "The comback method extending hash compaction with backtracking," in *Petri Nets and Other Models of Concurrency ICATPN 2007*, ser. Lecture Notes in Computer Science, J. Kleijn and A. Yakovlev, Eds. Springer Berlin Heidelberg, 2007, vol. 4546, pp. 445–464. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-73094-1_26

[10] J. Barnat, J. Havlek, and P. Rokai, "Distributed {LTL} model checking with hash compaction," *Electronic Notes in Theoretical Computer Science*, vol. 296, no. 0, pp. 79 – 93, 2013, proceedings the Sixth International Workshop on the Practical Application of Stochastic Modelling (PASM) and the Eleventh International Workshop on Parallel and Distributed Methods in Verification (PDMC). [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1571066113000376

[11] R. Pagh, G. Segev, and U. Wieder, "How to approximate a set without knowing its size in advance," *CoRR*, vol. abs/1304.1188, 2013.

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. The MIT Press, 2009.

[13] U. Stern and D. L. Dill, "A new scheme for memory-efficient probabilistic verification," in *in IFIP TC6/WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification*, 1996, pp. 333–348.

[14] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970. [Online]. Available: http://doi.acm.org/10.1145/362686.362692

[15] M. Naor and E. Yogev, "Sliding bloom filters," in *Algorithms and Computation*, ser. Lecture Notes in Computer Science, L. Cai, S.-W. Cheng, and T.-W. Lam, Eds. Springer Berlin Heidelberg, 2013, vol. 8283, pp. 513–523. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-45030-3_48

[16] S. Cohen and Y. Matias, "Spectral bloom filters." in *SIGMOD Conference*, A. Y. Halevy, Z. G. Ives, and A. Doan, Eds. ACM, 2003, pp. 241–252.

[17] B. Chazelle, J. Kilian, R. Rubinfeld, and A. 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*, ser. SODA '04. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2004, pp. 30–39. [Online]. Available: http://dl.acm.org/citation.cfm?id=982792.982797