The *Perfect Hash Filter*

A fast and efficient implementation of the *random positive approximate set* abstract data types.

Alexander Towell

atowell@siue.edu

Abstract

We define the semantics of the random approximate set and derive an immutable data structure that implements the s*static* random approximate set, denoted the Perfect Hash Filter. We analyze the theoretical properties of the Perfect Hash Filter, demonstrating that it is a *succinct* data structure that asymptotically obtains the information-theoretic lower bound on the average space complexity. Finally, we provide an implementation and compare it to the theoretical analysis.

Contents

List of Figures

List of Algorithms

1 Introduction

The Perfect Hash Filter is a data structure that is suitable to implement two types of sets, *positive approximate sets*[3] and *oblivious sets*[4]. Informally, positive approximate sets generate false positives at specifiable rate and oblivious sets are a type of approximate set with additional *confidentiality* guarantees.

In **??**, we precisely define concept of a set. Then, we define the approximate and oblivious positive sets. In §8, we derive the Perfect Hash Filter, a data structure that implements the approximate positive set and prove various properties, such as its absolute space efficiency. Following from that, in **??** we show how to tweak the implementation of the Perfect Hash Filter[1] to implement the *oblivious set*.

2 Approximate and oblivious positive sets

A set is given by the following definition.

Definition 2.1. *A set is an unordered collection of distinct elements from a universe of elements.*

A countable set is a *finite set* or a *countably infinite set*. A *finite set* has a finite number of elements. For example,

$$\mathcal{S} = \{1, 3, 5\}$$

is a finite set with three elements. A *countably infinite set* can be put in one-to-one correspondence with the set of natural numbers. The cardinality of a set $\mathcal{S}$ is a measure of the number of elements in the set, denoted by

$$|\mathcal{S}|\,. \tag{1}$$

[1]For instance, by varying the load factor.

The cardinality of a *finite set* is a non-negative integer and counts the number of elements in the set, e.g.,

$$|\{1,3,5\}| = 3\,.$$

The abstract data type of the immutable *positive* approximate set[3] is given by the following definition.

Definition 2.2. *The abstract data type of the* positive approximate set *over a countably infinite universe $\mathcal{U}$ has the following operations defined:*

$$\in\colon\ \mathcal{P}(\mathcal{U}) \times \mathcal{U} \mapsto \{\mathbf{true}, \mathbf{false}\}\,. \tag{2}$$

Let an element that is selected uniformly at random from the universe $\mathcal{U}$ be denoted by X*. A set $\mathcal{S}^\varepsilon(\varepsilon)$ is a* positive approximate set *of a set $\mathcal{S}$ with a false positive rate ε if the following conditions hold:*

(i) $\mathcal{S}$ is a subset of $\mathcal{S}^\varepsilon$. This condition guarantees that no false negatives *may occur.*

(ii) If X *is* not *a member of $\mathcal{S}$, it is a member of $\mathcal{S}^\varepsilon$ with a probability ε,*

$$\mathrm{P}\big[\mathrm{X} \in \mathcal{S}^\varepsilon \;\big|\; \mathrm{X} \notin \mathcal{S}\big] = \varepsilon\,. \tag{3}$$

The optimal space complexity of *countably infinite* positive approximate sets is given by the following postulate.

Postulate 2.1. *The* optimal *space complexity of a data structure implementing the* positive approximate set *over* countably infinite *universes is independent of the type of elements and depends only the false positive rate ε as given by*

$$-\log_2 \varepsilon \text{ bits/element}\,. \tag{4}$$

The immutable abstract data type of the *oblivious set*[4] is given by the following definition.

Definition 2.3. *Assuming that the only information about a set of interested $\mathcal{S}$ is given by another set $\check{\mathbb{S}}^+$, $\check{\mathbb{S}}^+$ is an* oblivious positive set *of $\mathcal{S}$ with a false positive rate ε if the following conditions hold:*

(i) $\check{\mathbb{S}}^+$ is a positive approximate set of $\mathcal{S}$ with a false positive rate ε.

(ii) The false positives are uniformly distributed over $\mathcal{U} \setminus \mathcal{S}$.

(iii) There is no efficient way to enumerate the elements in $\check{\mathbb{S}}^+$.[2]

(iv) Any estimator of the cardinality of $\mathcal{S}$ may only be able determine an approximate upper and lower bound, where the uncertainty may be traded for space-efficiency.

The absolute space efficiency of a data structure implementing an oblivious positive set consisting of m true positives with a false positive rate ε and an entropy β is given by

$$\mathtt{e}^*(\varepsilon, m, \beta) = \mathbb{E}\left[\frac{-(m+\mathrm{X})\log_2 \varepsilon}{\ell\big(\mathtt{make\_oblivious\_set}(m+\mathrm{X}, \varepsilon)\big)}\right]\,, \tag{5}$$

[2]That is, the true positives and false positives.

where

$$\mathtt{X} \sim \mathtt{DU}(0, 2^{\beta} - 1)\,. \tag{6}$$

The relative efficiency of the *optimal* positive oblivious set with entropy β to the *optimal* positive approximate set ($\beta = 0$) has an expectation given by

$$\mathtt{RE}(\,\cdot\,, m, \beta) = 2^{-\beta} \sum_{k=0}^{2^{\beta}-1} \left(1 + \frac{k}{m}\right)^{-1}\,. \tag{7}$$

For a fixed β, as $m \to \infty$ the relative efficiency goes to 1.

3 Perfect hash filters that trade time for space

$\mathtt{ph}_{\mathcal{S}} \colon \mathcal{U} \mapsto [0, \ldots, k]$
  $\mathrm{f} \colon \mathbb{N} \mapsto \{0, 1, \ldots, N\}^{[}t]$
  Compose two functions
  $\mathtt{ph}_{\mathcal{S}} \colon \mathcal{U} \mapsto \{0, 1, \ldots, N\}^{[}t] = \mathrm{f} \circ \mathtt{ph}_{\mathcal{S}}$

4 A C++ implementation

Random positive approximate sets are a well-defined concept. Any data structure T and set of functions dependent upon T required by the concept is an implementation of the model. Any generic algorithm (generic programming) parameterized by R which assumes R models a random approximate set may be applied to the data structure T.

We show a simple implementation.

4.1 Dynamic polymorphism

TODO: break up each function/class/maybe method into separate listings and then talk about each separately.

We may also use *dynamic polymorphism*. A dynamic polymorphic C++ interface of the *random approximate set* abstract data type is detailed next.

Assuming the elements of an approximate set are *non-enumerable*, i.e., only membership tests may be performed using the member-of $\in \colon \mathcal{U} \times \mathcal{P}(\mathcal{U})$ interface, set-theoretic operations may be implemented by storing each approximate set *as-is* and composing them together. For example, if we are given two approximate sets $\mathcal{S}_1^{\sigma}$ and $\mathcal{S}_2^{\sigma}$, the union $\mathcal{S}_1^{\sigma} \cup \mathcal{S}_2^{\sigma}$ is implemented by performing membership tests on both $\mathcal{S}_1^{\sigma}$ and $\mathcal{S}_2^{\sigma}$ and returning **true** if either membership test returns **true**, i.e.,

$$x \in (\mathcal{S}_1^{\sigma} \cup \mathcal{S}_2^{\sigma}) \equiv (x \in \mathcal{S}_1^{\sigma}) \lor (x \in \mathcal{S}_2^{\sigma})\;. \tag{8}$$

Any set-theoretic composition may be implemented as a combination of unions and complements as described in **??**. A C++ implementation for taking the union of two approximate sets is given by listing 1 and a C++ implementation for taking the complement of an approximate set is given by listing 2.

Other compositions may be implemented by composing set-union and set-complement, e.g., set-intersection is given by

Listing 1: C++ implementation of the union of approximate sets.

Listing 2: C++ implementation of the complement of approximate sets.

```
auto make_intersection(auto s1, auto s2) =
{
    return make_complement(make_union(
        make_complement(s1),
        make_complement(s2)));
}
```

5 Abstract data type of random approximate sets

A *type* is a set and the elements of the set are called the *values* of the type. An *abstract data type* is a type and a set of operations on values of the type. For example, the *integer* abstract data type is defined by the set of integers and standard operations like addition and subtraction.

A *data structure* is a particular way of organizing data and may implement one or more abstract data types. Two different data structures that implement the abstract data type should be *exchangable* in a computer program without changing the outward behavior of the program.

See §4 to see how a C++ interface for the random approximate set data type may be defined. A popular implementation of an algorithm and data structure that generates approximate sets consistent with the model is known as the *Bloom filter*.

6 Deterministic algorithm

By **????**, each independent observation of $\mathcal{S}^\sigma$ generates an uncertain number of false positives and false negatives. In practice, this may or may not be the case.

Suppose we have a deterministic algorithm that generates approximate sets with a specified expected false positive rate ε and false negative rate ω. The algorithm, being deterministic, necessarily generates a certain set $\mathcal{X}$ given an objective set $\mathcal{S}$. That is to say, the algorithm is a *total function*, $\mathrm{f}\colon \mathcal{P}(\mathcal{U}) \mapsto \mathcal{P}(\mathcal{U})$, with an *image*

$$\mathtt{image}(\mathrm{f}) = \{\, \mathrm{f}(\mathcal{A}) \mid \mathcal{A} \in \mathcal{P}(\mathcal{U}) \,\} \subseteq \mathcal{P}(\mathcal{U})\,. \tag{9}$$

Since the function f may map more than one input to the same output, and some sets in the codomain may not be mapped to by any set in the domain, f is (possibly) a non-surjective, non-injective function.

6.1 Algebraic properties

The only thing we can say with certainty *a priori*[3] about the image of f is that its members are subsets of $\mathcal{U}$ and it contains the empty set $\varnothing$ and universal set $\mathcal{U}$.

Definition 6.1. *A σ-algebra is closed under...*

The image of f is not necessarily a σ-algebra. However, the subsets of $\mathcal{U}$ that may be constructed by countable complements, unions, and intersections for elements of the image is by definition a σ-algebra and is denoted by $\sigma(\mathrm{f})$.

[3]A priori knowledge is independent of experience.

Since $\sigma(\mathrm{f})$ is a set of sets closed under unions, intersections, and complements, it is a Boolean algebra defined by the six-tuple $\left(\sigma(\mathrm{f}), \cup, \cap, \overline{}, \varnothing, \mathcal{U}\right)$, e.g., set-theoretic operations over the above Boolean algebra are of the form

$$\sigma(\mathrm{f}) \times \sigma(\mathrm{f}) \mapsto \sigma(\mathrm{f})\,. \tag{10}$$

Note that neither *positive* nor *negative* approximate sets are closed under complementation and thus are not Boolean algebras, but rather semi-rings under unions and intersections.

Suppose we have two distinct Boolean algebras, $\left(\sigma(\mathrm{f}), \cup, \cap, \overline{}, \varnothing, \mathcal{U}\right)$ and $\left(\sigma(\mathrm{g}), \cup, \cap, \overline{}, \varnothing, \mathcal{U}\right)$, where f and g are distinct deterministic algorithms that generate approximate sets for $\mathcal{P}(\mathcal{U})$. Set-theoretic operations over both Boolean algebras is the Boolean algebra $\left(\Sigma_{f \cup g}, \cup, \cap, \overline{}, \varnothing, \mathcal{U}\right)$, where $\Sigma_{f \cup g} = \sigma\big(\sigma(\mathrm{f}) \cup \sigma(\mathrm{g})\big)$. Note that $\sigma(\mathrm{f}), \sigma(\mathrm{g}) \subseteq \Sigma_{f \cup g}$, so if we take the limit $\lim_{n \to \infty} \Sigma_{f_1 \cup \cdots \cup f_n}$, where $f_1, \ldots, f_n$ are distinct mappings, $\Sigma_{f_1 \cup \cdots \cup f_n}$ converges to $\mathcal{P}(\mathcal{U})$.

In practice, it is often trivial to implement a large family of random approximate set generators with distinct Boolean algebras given a single implementation, e.g., "randomly" seeding a Bloom filter's hash function. As the size of this family goes to infinity, "randomly" selecting one of them each time we generate an approximate set generates a random sample consistent with the probability distribution given by **??**.

6.2 Compatibility with the probabilistic model

How do we reconcile the fact that the algorithm is a *function* with the *probabilistic model*? In this context, the notion of *probability* quantifies our *ignorance*:

1. Given an objective set $\mathcal{S}$, we do not have complete *a priori* knowledge about the set $\mathcal{X}$ it maps to (the approximate set model only provides *a priori* knowledge about the probability distribution $\mathcal{S}^\sigma$). We acquire *a posterior* knowledge[4] by applying the algorithm and *observing* $\mathcal{X}$ as its output.

2. Given a set $\mathcal{X}$ that approximates some unknown objective set $\mathcal{S}$, we do not have complete *a priori* knowledge about $\mathcal{S}$. According to the probabilistic model, the only *a priori* knowledge we have is given by the specified *expected* false positive and false negative rates. We may acquire a posteriori knowledge by applying the algorithm to each set in $\mathcal{P}(\mathcal{U})$ and remembering the sets that map to $\mathcal{X}$.[5] However, since f is (possibly) non-injective, one or more sets may map to $\mathcal{X}$ and thus this process may not completely eliminate uncertainty. Additionally, the domain $\mathcal{P}(\mathcal{U})$ has a cardinality $2^{|\mathcal{U}|}$ and thus exhaustive maps are impractical to compute even for relatively small domains.[6] However, we may still reduce our uncertainty by mapping some subset of interest.

Suppose $\mathcal{U}$ is finite. There are $2^{|\mathcal{U}|}$ subsets of $\mathcal{U}$, and each set in $\mathcal{P}(\mathcal{U}) \setminus \{\varnothing, \mathcal{U}\}$ may map (assuming both false positives and false negatives are possible) to *any* of subset. There are a total of

$$2^{2(|\mathcal{U}|-1)} \tag{11}$$

possible functions that map perform this mapping.[7] Each of these functions is compatible with the probabilistic model. For instance, a Bloom filter (positive approximate set) may have a family of

[4] A posteriori knowledge is dependent on experience.

[5] If the approximate set is the result of the union, intersection, and complement of two or more approximate sets, then we must consider the closure.

[6] In the case of *countably infinite* domains, it is not even theoretically possible.

[7] There are a countably infinite number of algorithms that may carry out the computations that implement any particular function.

hash function that, for a particular binary coding of the elements of a given universal set, maps *every* element in the universal set to the same hash. Thus, for instance, no matter the objective set $\mathcal{X} \subseteq \mathcal{U}$, it will map to $\mathcal{U}$. The Bloom filter had a theoretically sound implementation, but only after empirical evidence was it discovered that it was not suitable. This is an extremely unlikely outcome in the case of large universal sets, but as the cardinality of the universal set decreases, the probability of such an outcome increases. Indeed, at $|U| = 2$, the probability of this outcome is ?.

Thus, *a priori* knowledge, e.g., a theoretically sound algorithm, is not in practice sufficient (although for large universal sets, the probability is negligible). The suitability of an algorithm can only be determined by acquring *a posterior* knowledge.

We could explore the space of functions in the family and only choose those which, on some sample of objective sets of interest, generates the desired expectations for the false positive and false negative rates with the desired variances. Most of them will if constructed in the right sort of way.

A family of functions that are compatible with the probabilistic model is given by observing a particular realization $\mathcal{X} = \mathcal{S}^\sigma$ and outputting $\mathcal{X}$ on subsequent inputs of $\mathcal{S}$, i.e., caching the output of a *non-deterministic* process that conforms to the probabilistic model. This is essentially how well-known implementations like the Bloom filter work, where the pseudo-randomness comes from mechanical devices like hash functions that approximate random oracles.

The false positive rate of the approximate set corresponding to objective set $\mathcal{X}$ is given by

$$\grave{\varepsilon}(\mathcal{X}) = \frac{1}{n} \sum_{x \in \overline{\mathcal{X}}} \mathbb{1}_{\mathrm{f}(\mathcal{X})}(x)\,, \tag{12}$$

where $n = |\overline{\mathcal{X}}|$.

Let $\mathcal{U}_p$ denote the set of objective sets with cardinality p. The *mean* false positive rate,

$$\overline{\varepsilon} = \frac{1}{|\mathcal{U}_p|} \sum_{\mathcal{X} \in \mathcal{U}_p} \grave{\varepsilon}(\mathcal{X})\,, \tag{13}$$

is an unbiased estimator of ε and the population variance

$$s_{\varepsilon}^2 = \frac{1}{|\mathcal{U}_p|} \sum_{\mathcal{X} \in \mathcal{U}_p} \grave{\varepsilon}(\mathcal{X})\,, \tag{14}$$

is an unbiased estimator of $\mathrm{Var}[\mathcal{E}_n] = \varepsilon(1-\varepsilon)/n$.

Proof. We imagine that the function f caches the output of a *non-deterministic* process that conforms to the probabilistic model. Thus, each time the function maps an objective set $\mathcal{X}$ of cardinality p to its approximation, the algorithm *observes* a realization of $\mathcal{E}_n = \grave{\varepsilon}$. Thus,

$$\begin{aligned} \overline{\varepsilon} &= \frac{1}{|\mathcal{U}_p|} \sum_{\mathcal{X}_\rangle \in \mathcal{U}_p} \grave{\varepsilon}(\mathcal{X}_\rangle) && \text{(a)} \\ &= \frac{1}{|\mathcal{U}_p|} \sum_{\mathcal{X}_\rangle \in \mathcal{U}_p} \mathrm{E}\Big[\mathcal{E}_n^{(i)}\Big] = \varepsilon\,. && \text{(b)} \end{aligned}$$

□

7 Space complexity

If the finite cardinality of a universe is u and the set is *dense* (and the approximation is also dense, i.e., the false negative rate is relatively small), then

$$\mathcal{O}(u) \text{ bits} \tag{15}$$

are needed to code the set, which is independent of m, the false positive rate, and the false negative rate.

The lower-bound on the *expected* space complexity of a data structure implementing the *approximate set* abstract data type where the elements are over a *countably infinite* universe is given by the following postulate.

Postulate 7.1. *The* information-theoretic lower-bound *of a data structure that implements the countably infinite* approximate set *abstract data type has an* expected *bit length given by*

$$-(1-\omega)\log_2 \varepsilon \text{ bits/element}\,, \tag{16}$$

where $\varepsilon > 0$ is the false positive rate and ω is the false negative rate.

The *relative space efficiency* of a data structure X to a data structure Y is some value greater than 0 and is given by the ratio of the bit length of Y to the bit length of X,

$$\mathtt{RE}(X,Y) = \frac{\ell(Y)}{\ell(X)}\,, \tag{17}$$

where ℓ is the bit length function. If $\mathtt{RE}(X,Y) < 1$, X is less efficient than Y, if $\mathtt{RE}(X,Y) > 1$, X is more efficient than Y, and if $\mathtt{RE}(X,Y) = 1$, X and Y are equally efficient. The absolute space efficiency is given by the following definition.

Definition 7.1. *The absolute space efficiency of a data structure X, denoted by $\mathtt{E}(X)$, is some value between 0 and 1 and is given by the ratio of the bit length of the theoretical lower-bound to the bit length of X,*

$$\mathtt{E}(X) = \frac{\ell}{\ell(X)}\,, \tag{18}$$

where $\ell(X)$ denotes the bit length of X and ℓ denotes the bit length of the information-theoretic lower-bound. The closer $\mathtt{E}(X)$ is to 1, the more space-efficient the data structure. A data structure that obtains an efficiency of 1 is optimal.[8]

The *absolute* space efficiency of a data structure X implementing an approximate set $\mathcal{S}^\sigma$ with a false positive rate ε and false negative rate ω is given by

$$\mathtt{E}(X) = \frac{-m(1-\omega)\log_2 \varepsilon}{\ell(X)}\,, \tag{19}$$

where $m = |\mathcal{S}|$. The most useful sort of asymptotic optimality is with respect to the parameter m.

A well-known implementation of countably infinite *positive approximate set* is the Bloom filter[1] which has an expected space complexity given by

$$-\frac{1}{\ln 2}\log_2 \varepsilon \text{ bits/element}\,, \tag{20}$$

[8]Sometimes, a data structure may only obtain the information-theoretic lower-bound with respect to the limit of some parameter, in which case the data structure *asymptotically* obtains the lower-bound with respect to said parameter.

thus the absolute efficiency of the Bloom filter is $\ln 2 \approx 0.69$. A practical implementation of the *positive approximate set* abstract data type is given by the *Perfect Hash Filter*[**?**], which compares favorably to the Bloom filter in may circumstances.

The *Singular Hash Set*[**?**] is a *theoretical* data structure that optimally implements the abstract data types of the *approximate set* and the *oblivious set*[4] abstract data types.

7.1 Space efficiency of *unions* and *differences*

As a way to implement *insertions* and *deletions*, we consider the space efficiency of the set-theoretic operations of unions and differences of approximate sets.

Let $\mathcal{S}_1 = \{x_{j_1}, \ldots, x_{j_m}\}$ and suppose we wish to insert the elements $x_{k_1}, \ldots, x_{k_p}$ into $\mathcal{S}_1$. If X_1 is a mutable object, then an *insertion* operator may be applied on X_1 for each x_{k_i}, $i = 1, \ldots, p$.

Alternatively, if X_1 is immutable, then we may construct another object, X_2, that implements the set $\mathcal{S}_2 = \{x_{k_1}, \ldots, x_{k_p}\}$, and then apply the union function,

$$X_1 \cup X_2 \,. \tag{21}$$

If we replace X_1 and X_2 by objects that implement *positive approximate sets* of $\mathcal{S}_1$ and $\mathcal{S}_2$ respectively, then by **??**, the false positive rate of the resulting approximate set is $\grave{\varepsilon}_1 + \grave{\varepsilon}_2 - \grave{\varepsilon}_1\grave{\varepsilon}_2$.

The space efficiency of this positive approximate set is given by the following theorem.

Theorem 7.1. *Given two countably infinite positive approximate sets $\mathcal{S}_1^\varepsilon$ and $\mathcal{S}_2^\varepsilon$ respectively with false positive rates $\grave{\varepsilon}_1$ and $\grave{\varepsilon}_2$, the approximate set $\mathcal{S}_1^\varepsilon \cup \mathcal{S}_2^\varepsilon$, which has an induced false positive rate $\grave{\varepsilon}_1 + \grave{\varepsilon}_2 - \grave{\varepsilon}_1\grave{\varepsilon}_2$, has an expected upper-bound on its absolute efficiency given by*

$$\mathtt{E}(\grave{\varepsilon}_1, \grave{\varepsilon}_2 | \alpha_1, \alpha_2) = \frac{\log_2(\grave{\varepsilon}_1 + \grave{\varepsilon}_2 - \grave{\varepsilon}_1\grave{\varepsilon}_2)}{\alpha_1 \log_2 \grave{\varepsilon}_1 + \alpha_2 \log_2 \grave{\varepsilon}_2}\,, \tag{22}$$

where

$$\begin{aligned} 0 < \alpha_1 &= \frac{|\mathcal{S}_1|}{|\mathcal{S}_1 \cup \mathcal{S}_2|} \leq 1\,, \\ 0 < \alpha_2 &= \frac{|\mathcal{S}_2|}{|\mathcal{S}_1 \cup \mathcal{S}_2|} \leq 1\,, \\ 1 &\leq \alpha_1 + \alpha_2\,. \end{aligned} \tag{23}$$

As $\grave{\varepsilon}_j \to 1$ or $\grave{\varepsilon}_j \to 0$ for $j = 1, 2$, or $(\grave{\varepsilon}_1, \grave{\varepsilon}_2) \to (1, 1)$, the absolute efficiency goes to 0. [9]

Proof. The proof comes in two parts. First, we prove eq. (22), and then we prove the bounds on α_1 and α_2 given by eq. (23).

Let X and Y denote optimally space-efficient data structures that respectively implement positive approximate sets $\mathcal{S}_1^\varepsilon$ and $\mathcal{S}_2^\varepsilon$ with false positive rates $\grave{\varepsilon}_1$ and $\grave{\varepsilon}_2$. By **??**, their union has an induced false positive rate given by

$$\grave{\varepsilon}_1 + \grave{\varepsilon}_2 + \grave{\varepsilon}_1\grave{\varepsilon}_2\,. \tag{a}$$

The information-theoretic lower-bound of the approximate set of $\mathcal{S}_1 \cup \mathcal{S}_2$ with the above false positive rate is given by

$$-\,|\mathcal{S}_1 \cup \mathcal{S}_2| \log_2(\grave{\varepsilon}_1 + \grave{\varepsilon}_2 + \grave{\varepsilon}_1\grave{\varepsilon}_2) \text{ bits}\,. \tag{b}$$

Since we assume we only have X and Y and it is not possible to enumerate the elements in either, we must implement their union by storing and separately querying both X and Y. Since X and Y

[9] As $(\grave{\varepsilon}_1, \grave{\varepsilon}_2) \to (0, 0)$, the absolute efficiency depends on the path taken. In most cases, it goes to 0.

are optimal, $\ell(X) = -|\mathcal{S}_1| \log_2 \grave{\varepsilon}_1$ and $\ell(Y) = -|\mathcal{S}_2| \log_2 \grave{\varepsilon}_2$. Making these substitutions yields an absolute efficiency ???

$$\mathtt{E} = \frac{|\mathcal{S}_1 \cup \mathcal{S}_2| \log_2(\grave{\varepsilon}_1 + \grave{\varepsilon}_2 + \grave{\varepsilon}_1\grave{\varepsilon}_2)}{|\mathcal{S}_1| \log_2 \grave{\varepsilon}_1 + |\mathcal{S}_2| \log_2 \grave{\varepsilon}_2}\,. \tag{c}$$

Letting

$$\alpha_1 = \frac{|\mathcal{S}_1|}{|\mathcal{S}_1 \cup \mathcal{S}_2|} \text{ and } \alpha_2 = \frac{|\mathcal{S}_2|}{|\mathcal{S}_1 \cup \mathcal{S}_2|}\,, \tag{d}$$

we may rewrite eq. (c) as

$$\frac{\log_2(\grave{\varepsilon}_1 + \grave{\varepsilon}_2 - \grave{\varepsilon}_1\grave{\varepsilon}_2)}{\alpha_1 \log_2 \grave{\varepsilon}_1 + \alpha_2 \log_2 \grave{\varepsilon}_2}\,. \tag{22 revisited}$$

In the second part of the proof, we prove the bounds on α_1 and α_2 as given by eq. (23). Both α_1 and α_2 must be non-negative since each is the ratio of two positive numbers (cardinalities). If $|\mathcal{S}_1| \ll |\mathcal{S}_2|$, then $\alpha_1 \approx 0$. If $\mathcal{S}_1 \supset \mathcal{S}_2$, then $\alpha_1 = 1$. A similar argument holds for α_2. Finally,

$$\alpha_1 + \alpha_2 = \frac{|\mathcal{S}_1| + |\mathcal{S}_2|}{|\mathcal{S}_1 \cup \mathcal{S}_2|} \tag{e}$$

has a minimum value by assuming that $\mathcal{S}_1$ and $\mathcal{S}_2$ are disjoint sets (i.e., their intersection is the empty set), in which case

$$\alpha_1 + \alpha_2 = \frac{|\mathcal{S}_1| + |\mathcal{S}_2|}{|\mathcal{S}_1| + |\mathcal{S}_2|} = 1\,. \tag{f}$$

□

See **??** for a contour plot of the expected lower-bound as a function of $\grave{\varepsilon}_1$ and $\grave{\varepsilon}_2$. As $\grave{\varepsilon}_1 \to 0$ or $\grave{\varepsilon}_2 \to 0$, the efficiency goes to 0.

The lower-bound on the efficiency of the union of approximate sets is given by the following corollary.

Corollary 7.1.1. *Given two positive, non-enumerable approximate sets with false positive rates $\grave{\varepsilon}_1$ and $\grave{\varepsilon}_2$, their union is an approximate set that has an efficiency that is expected to be greater than the lower bound given by*

$$\min \mathtt{E}(\grave{\varepsilon}_1, \grave{\varepsilon}_2) = \frac{\log_2(\grave{\varepsilon}_1 + \grave{\varepsilon}_2 - \grave{\varepsilon}_1\grave{\varepsilon}_2)}{\log_2 \grave{\varepsilon}_1\grave{\varepsilon}_2}\,. \tag{24}$$

Corollary 7.1.2. *If $\varepsilon_1 = \varepsilon_2 = \varepsilon$, then the absolute efficiency is given by*

$$\mathtt{E}(\grave{\varepsilon}|\alpha) = \left(1 + \frac{\log_2(2 - \grave{\varepsilon})}{\log_2 \grave{\varepsilon}}\right)\left(1 - \frac{\alpha}{2}\right)\,, \tag{25}$$

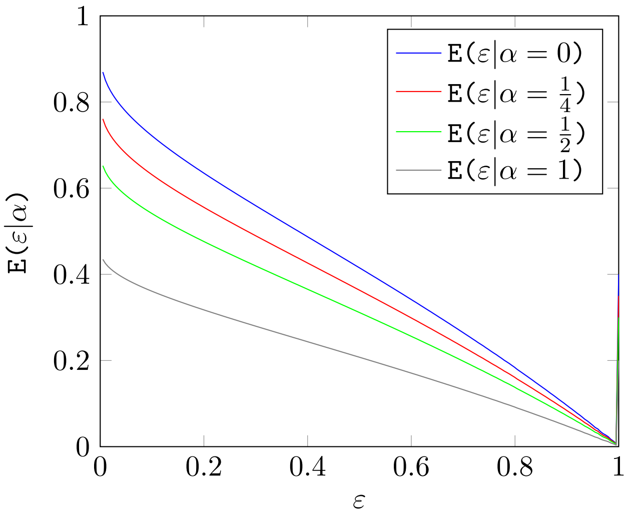
where

$$0 \leq \alpha = \frac{|\mathcal{S}_1 \cap \mathcal{S}_2|}{|\mathcal{S}_1 \cup \mathcal{S}_2|} \leq 1\,, \tag{26}$$

which is a monotonically decreasing function with respect to ε and α with limits given by $\lim_{\grave{\varepsilon} \to 0} \mathtt{E}(\varepsilon) = 1$ and $\lim_{\varepsilon \to 1} \mathtt{E}(\varepsilon) = 0$.

See fig. 1 for a graphic illustration.

Figure 1: Expected lower-bound on efficiency of the union of two approximate sets with the same false positive rate ε, neither of which can be enumerated.



8 Perfect hash filter: a practical implementation of the random positive approximate set model

Notation. The *bit set* $\{0, 1\}$ is denoted by $\mathcal{B}$. The set of all bit strings of length n is the Cartesian product $\mathcal{B}^n$ where $|\mathcal{B}^n| = 2^n$. The *empty string* is denoted by ϵ where $\mathcal{B}^0 = \{\epsilon\}$. The set of all bit strings of length 0 to n is denoted by $\mathcal{B}^{\leq n} = \bigcup_{j=0}^{n} \mathcal{B}^j$ where $|\mathcal{B}^{\leq n}| = 2^{n+1} - 1$. The countably infinite set of all bit strings (of any length) is denoted by $\mathcal{B}^* = \lim_{n\to\infty} \mathcal{B}^{\leq n}$.

The *Bloom filter*[1] is a well-known data structure that implements a *random positive approximate set* over $\mathcal{B}^*$ and obtains a space complexity given approximately by

$$-1.44 \log_2 \varepsilon \text{ bits/element}\,. \tag{27}$$

We consider an alternative data structure, denoted the Perfect Hash Filter that also implements the *random positive approximate set* over $\mathcal{B}^*$ and compares favorably to the Bloom filter in some ways.

A hash function is related to countable sets $\mathcal{B}^*$ and $\mathcal{B}^n$ and is given by the following definition.

Definition 8.1. *A* hash function $\mathtt{h} \colon \mathcal{B}^* \mapsto \mathcal{B}^n$ *is a function such that all bit strings of arbitrary-length are mapped (hashed) to bit strings of fixed-length n.*

For a given $x \in \mathcal{B}^*$, $y = \mathtt{h}(x) \in \mathcal{B}^*_n$ is denoted the *hash* of x. The Perfect Hash Filter depends on a perfect hash function as given by the following definition.

Definition 8.2. *A perfect hash function of the set $\mathcal{S}$, denoted by*

$$\mathtt{ph}_{\mathcal{S}} \colon \mathcal{B}^* \mapsto \mathbb{Z}^+\,, \tag{28}$$

is a hash function such that the mapping from $\mathcal{S}$ to $\mathbb{Z}^+$ is one-to-one.

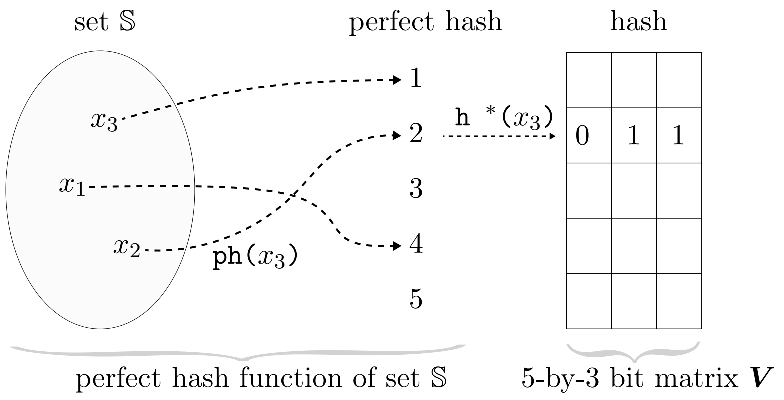
The *load factor* of $\mathtt{ph}_{\mathcal{S}}$ is given by the following definition.

Definition 8.3. *The load factor of $\mathtt{ph}_{\mathcal{S}}$ is a rational number given by*

$$r = \frac{m}{N}\,, \tag{29}$$

where $m = |\mathcal{S}|$, $N \geq m$, and N is the maximum hash of $\mathtt{ph}_{\mathcal{S}}$.

Figure 2: A perfect hash filter implementing a static approximate set $\{x_1, x_2, x_3\}$ with a false positive rate $\varepsilon = 2^{-3}$ and a load factor $r = \frac{3}{5}$.



Definition 8.4. *A* minimal *perfect hash function has a load factor* 1*, i.e., the mapping from $\mathcal{S}$ to $\mathbb{Z}^+$ is* one-to-one *and* onto*.*

The Perfect Hash Filter is a data structure given by the following definition.

Definition 8.5. *The Perfect Hash Filter is a data structure that implements*[10] *the abstract data type of the* positive approximate set *where the universe of elements is given by $\mathcal{B}^*$.*

The *Perfect Hash Filter* is a product type PerfectHashFilter = BitMatrix × PerfectHash and a set of functions

1. `contains`: $\mathcal{B}^* \times$ PerfectHashFilter]

tuple of type [BitMatrix] that implements[11] the abstract data type of the *positive approximate set* where the universe of elements is given by $\mathcal{B}^*$.

The *minimal* perfect hash filter given by the following definition.

Definition 8.6. *The* minimal *perfect hash filter is a perfect hash filter with a load factor* 1*, i.e., every element x in the set being approximated is* perfectly hashed*.*

Replacing the *perfect hash function* with a *k-perfect hash function*, where disjoint subsets of up to k elements may collide, generalizes the Perfect Hash Filter into an approximate set where both *false positives* and *false negatives* are possible.[12]

Algorithms 3 and 4 describe the implementations of `make_perfect_hash_filter` and `contains` respectively.

Example 1 *Consider a set $\mathcal{S} = \{x_1, x_2, x_3\}$ and suppose we wish to approximate this set with a false positive rate $\varepsilon = 2^{-3}$. The perfect hash filter depicted by fig. 2 is a representation of this* frozen approximate set*.*

We make the following set of assumptions. See [5] for more on these assumptions.

[10] In theorem 8.1, we prove the Perfect Hash Filter implements the static approximate set.

[11] In theorem 8.1, we prove the Perfect Hash Filter implements the static approximate set.

[12] We may also simply approximate a subset of a set $\mathcal{S}$ to support non-zero false negative rates, but *k-perfect hashing* generates a more efficient solution.

Algorithm 1: Implementation of `make_perfect_hash_filter`

```
input  :
    $\mathcal{S}$ The finite set to approximate.
    $\varepsilon$ The false positive rate.
    $r$ The load factor.
output : A Perfect Hash Filter that approximates $\mathcal{S}$ with a false positive rate $\varepsilon$ and a load
         factor $r$.
function make_perfect_hash_filter($\mathcal{S}$, $\varepsilon r$)
    ph$_{\mathcal{S}}$ ← make_perfect_hash($\mathcal{S}$, $r$);
    $N \leftarrow \lceil |\mathcal{S}|/r \rceil$;
    $M \leftarrow \lceil -\log_2(\varepsilon) \rceil$;
    $\boldsymbol{V} \leftarrow N$-by-$M$ bit matrix;
    for $x \in \mathcal{S}$ do
        $i \leftarrow$ ph$_{\mathcal{S}}(x)$;
        $h \leftarrow$ h $^*(x) \mod M$;
        $\boldsymbol{V}[i] \leftarrow h$;
    end
    // This tuple is sufficient to code the Perfect Hash Filter.
    return $(\boldsymbol{V}, \text{ph}_{\mathcal{S}})$;
```

Assumption 1. *The set $\mathcal{S}$ consists of m elements, where each element is drawn uniformly at random without replacement from $\mathcal{B}^*$.*

Assumption 2. *The perfect hash function* ph$_{\mathcal{S}}$ *approximates a random oracle.*

Assumption 3. *The hash function* h $^*\colon \mathcal{B}^* \mapsto \mathcal{B}_n^*$ *approximates a random oracle.*

The Perfect Hash Filter is an implementation of a *static approximate set* as given by the following theorem.

Theorem 8.1. *A Perfect Hash Filter is an implementation of* static approximate set *with a false positive rate that is a rational number given by*

$$\varepsilon \in \left\{2^{-k} \colon k \in \mathbb{Z}^+\right\} \subset (0,1]\ . \tag{30}$$

Proof. For the Perfect Hash Filter to be an implementation of approximate set $\mathcal{S}^\varepsilon \supset \mathcal{S}$, by definition 2.2 the following conditions must be met:

1. For all $x \in \mathcal{S}$, $x \in \mathcal{S}^\varepsilon$ returns **true**.
2. For any $x \notin \mathcal{S}$, $x \in \mathcal{S}^\varepsilon$ returns **true** with probability ε and **false** with probability $1 - \varepsilon$.

First, we prove condition 1. For some $x \in \mathcal{B}^*$, let $i =$ ph$_{\mathcal{X}}(x)$. For `contains`($\mathcal{S}^\varepsilon$,x) to return **true**, row $\boldsymbol{V}[i]$ must equal the hash given by `h`(x). Since ph $_{\mathcal{S}}$ perfectly hashes all $y \in \mathcal{S}$ and $x \in \mathcal{S}$, no other member of $\mathcal{S}$ hashes ?, and $\boldsymbol{V}[i]$ is explicitly set to the hash h(x) during the construction of the Perfect Hash Filter as given by algorithm 3.

Algorithm 2: Implementation of `contains`

```
input :
    $\mathcal{S}^\varepsilon$ The Perfect Hash Filter data structure that approximates $\mathcal{S}$ with a false positive rate $\varepsilon$ and a load factor $r$.
    $x$ The bit string to test membership of in $\mathcal{S}^\varepsilon$.
output : true if $x \in \mathcal{S}$, otherwise true with probability $\varepsilon$ and false with probability $1-\varepsilon$.
1 function contains($\mathcal{S}^\varepsilon, x$)
2     $M \leftarrow$ columns($\boldsymbol{V}$);
3     $N \leftarrow$ rows($\boldsymbol{V}$);
4     $i \leftarrow \mathtt{ph}_{\mathcal{S}}(x)$;
5     $h \leftarrow \mathtt{h}(x) \mod N$;
6     return $\boldsymbol{V}[i] = h$;
```

Second, we prove condition 2. Observe that $\mathtt{ph}_{\mathcal{S}}$ is a hash function that takes any $x \in \mathcal{B}^*$ and maps it to an integer between 1 and N. The corresponding row of $\boldsymbol{V}$ consists of M bits. There are 2^M possible bit strings of length M and, by assumption 3, `h` uniformly distributes over these possibilities. Thus, for $x \notin \mathcal{S}$, the probability that `h(`x`)` is equal to the corresponding M bits is given by the false positive rate

$$\varepsilon = 2^{-M}\,. \tag{a}$$

Since M is any non-negative integer, the false positive rate is given by eq. (30). □

8.1 Space complexity

The minimum possible bit length of a data structure implementing the abstract data type of the *approximate set* is given by the following postulate.

Postulate 8.1. *The information-theoretic lower-bound of a data structure that implements the* approximate set *abstract data type has an* expected *bit length given by*

$$-\log_2 \varepsilon \text{ bits/element}\,, \tag{31}$$

where ε is the false positive rate.

The *expected* bit length of the *minimal* Perfect Hash Filter has a lower-bound given by the following theorem.

Theorem 8.2. *The* expected *bit length of the Perfect Hash Filter with a load factor r has a lower-bound given by*

$$\log_2 e - \frac{1}{r}\log_2 \varepsilon - \left(\frac{1}{r} - 1\right)\log_2\left(\frac{1}{1-r}\right) \text{ bits/element}\,, \tag{32}$$

where ε is the false positive rate and r is the load factor.

Proof. The proof comes in two parts. First, according to Theorem 3 in [5], the *expected* lower-bound for a perfect hash function with a load factor r is given by

$$\log_2 \mathrm{e} - \left(\frac{1}{r} - 1\right)\log_2\left(\frac{1}{1-r}\right) \text{bits/element}\,. \tag{a}$$

Second, a perfect hash function with a load factor r maps each $x \in \mathcal{S}$ to a unique integer between 1 and $\frac{m}{r}$, where $m = |\mathcal{S}|$, which requires a bit matrix $\boldsymbol{V}$ with m rows. Also, as shown in the proof for theorem 8.1, a false positive rate ε requires $M = -\log_2 \varepsilon$ bits. Thus, the bit matrix $\boldsymbol{V}$ has m rows and $-\log_2 \varepsilon$ columns, resulting in a total of

$$-\frac{m}{r} \log_2 \varepsilon \text{ bits}\,. \tag{b}$$

Adding both parts of the proof together results in

$$-\frac{m}{r} \log_2 \varepsilon + m \log_2 \mathrm{e} - m\left(\frac{1}{r} - 1\right) \log_2\left(\frac{1}{1-r}\right) \text{bits}\,. \tag{c}$$

Since we are interested in the bits per element, we divide by m yielding the result

$$-\frac{1}{r} \log_2 \varepsilon + \log_2 \mathrm{e} - \left(\frac{1}{r} - 1\right) \log_2\left(\frac{1}{1-r}\right) \text{bits/element}\,. \tag{d}$$

□

Corollary 8.2.1. *The* expected *bit length of the* minimal *Perfect Hash Filter has a lower-bound given by*

$$\log_2 \frac{e}{\varepsilon} \text{ bits/element}\,, \tag{33}$$

where ε is the false positive rate.

Proof. First, 1.44bits is the *expected* bits per element for optimally space-efficient *minimal perfect hash functions.*

Second, a *minimal* perfect hash function maps each $x \in \mathcal{S}$ to a unique integer between 1 and $m = |\mathcal{S}|$ which requires a bit matrix $\boldsymbol{V}$ with m rows. Also, as shown in the proof for theorem 8.1, a false positive rate ε requires $M = -\log_2 \varepsilon$ bits. Thus, the bit matrix $\boldsymbol{V}$ has m rows and $-\log_2 \varepsilon$ columns, resulting in a total of

$$-m \log_2 \varepsilon \text{ bits}\,. \tag{a}$$

Adding both parts of the proof together results in

$$-m \log_2 \varepsilon + 1.44m \text{ bits}\,. \tag{b}$$

Since we are interested in the bits per element, we divide by m yielding the result

$$1.44 - \log_2 \varepsilon \text{ bits/element}\,. \tag{c}$$

□

The bit length of an object x is denoted by

$$\ell(x)\,, \tag{34}$$

e.g., the bit length of any $x \in \mathcal{B}_n^*$ is given by $\ell(x) = n$. The relative space efficiency metric is given by the following definition.

Definition 8.7. *The relative space efficiency of a data structure X to a data structure Y is given by the ratio of the bit length of Y to the bit length of X as given by*

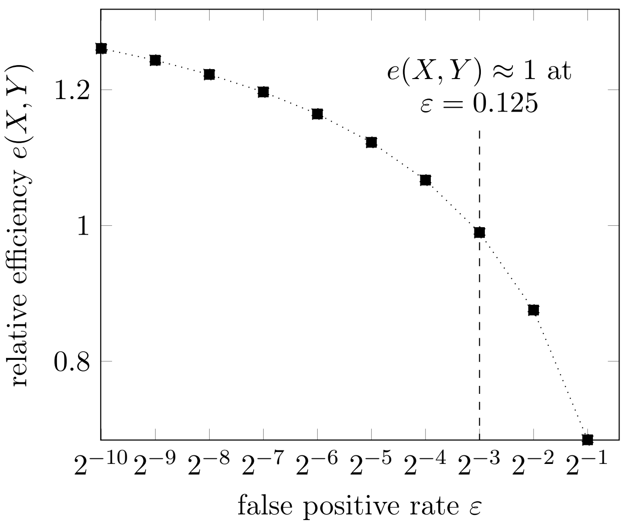
$$e(X, Y) = \frac{\ell(Y)}{\ell(X)}\,, \tag{35}$$

where ℓ is the bit length function. There are three possibilities:

Figure 3: Optimal load factor r^* for a given false positive rates ε for Perfect Hash Filter X. We compute the corresponding absolute efficiency and the relative efficiency to Bloom filter Y.

ε	r^*	$e^*(X)$	$e(X,Y)$
2^{-1}	0.768	0.474	0.684
2^{-2}	0.898	0.607	0.875
2^{-3}	0.952	0.686	0.990
2^{-4}	0.976	0.740	1.067
2^{-5}	0.988	0.778	1.123
2^{-6}	0.994	0.807	1.164
2^{-7}	0.997	0.830	1.197
2^{-8}	0.999	0.847	1.223
2^{-9}	0.999	0.862	1.244
2^{-10}	1.000	0.874	1.261
2^{-20}	1	0.933	1.346
2^{-30}	1	0.954	1.377
2^{-60}	1	0.977	1.409
2^{-120}	1	0.988	1.426

Figure 4: Relative efficiency of the Perfect Hash Filter X to the Bloom filter Y



1. $\mathtt{RE}(X, Y) < 1 \iff X$ *is less efficient than* Y*.*

2. $\mathtt{RE}(X, Y) = 1 \iff X$ *and* Y *are equally efficient.*

3. $\mathtt{RE}(X, Y) > 1 \iff X$ *is more efficient than* Y*.*

The *absolute* space efficiency of an approximate set X with a false positive rate ε is given by

$$\mathtt{e}^*(X) = e(X, -\log_2 \varepsilon)\,. \tag{36}$$

An asymptotically optimal space-efficient data structure has an absolute efficiency that goes to 1 as the cardinality m of the set being approximated goes to ∞. The absolute efficiency of the Perfect Hash Filter is not asymptotically optimal since it has an asymptotic limit less than 1 that is a function of the false positive rate ε as shown in fig. 3. However, as ε goes to 0 the absolute efficiency goes to 1.

Comparisons to the *Bloom filter*. The Perfect Hash Filter is most suitable as an immutable data structure. It is optimized for read-many, write-once usage patterns. Once it is constructed, only on the order of

$$\mathcal{O}\left(1\right) \tag{37}$$

hash functions need to be computed to test an element for membership. The Bloom filter is suitable as a mutalbe data structure; elements may be inserted into it at any point at a cost of increasing the false positive rates. However, it requires computing on the order of

$$\mathcal{O}\left(-\log \varepsilon\right) \tag{38}$$

hash functions per membership test.

The *Bloom filter* has an expected encoding size given approximately by

$$-1.44 \log_2 \varepsilon \text{ bits/element}\,. \tag{39}$$

The absolute efficiency of the Bloom filter is approximately $\ln 2 \approx 0.69$.

The relative efficiency of the Perfect Hash Filter to the Bloom filter is a function of the false positive rate ε as shown in figs. 3 and 4. If $\varepsilon < 2^{-3}$, the Bloom filter is more space efficient; otherwise the Perfect Hash Filter is more space efficient. As ε goes to 0, the relative efficiency of the Perfect Hash Filter to the Bloom filter goes to 1.44.

8.2 Entropy

If the load factor is known, the cardinality of $\mathcal{S} \subset \mathcal{S}^\varepsilon$ is given by

$$|\mathcal{S}| = Nr\,. \tag{40}$$

If the load factor is not known, then the cardinality may be probabilistically estimated. According to [], the probability mass of the perfect hash function is given by

$$\mathrm{N} \sim \mathrm{p_N}(n \mid m, r) = q^{2^n - 1}\left(1 - q^{2^n}\right) \tag{41}$$

where

$$\begin{aligned} q &= 1 - \frac{(m/r)!}{(m/r)^m((m/r) - m)!}\,, \\ m &= |\mathcal{S}|\,, \end{aligned} \tag{42}$$

It does contain some information. We know that if the bit matrix of hashes $\boldsymbol{V}$ has m rows and k columns, then $|\mathcal{S}| \leq m$ and the expected false positive rate is $\varepsilon = 2^{-k}$.

Theorem 8.3. *The random bit string that codes the Perfect Hash Filter is a* maximum entropy *coder for the* approximate set *abstract data type.*

Proof. For a random bit X to have maximum entropy, it must be Bernoulli distribution with equiprobability, i.e.,

$$\mathrm{X} \sim \texttt{BER}\left(p = \frac{1}{2}\right) . \tag{a}$$

The Perfect Hash Filter of $\mathcal{S}$ is coded by two parts. The first party is a *cryptographic* hash of m elements in $\mathcal{S}$ concatenated with a bit string b_n, denoted by h_k. The *cryptographic* hash function uniformly distributes over $\mathcal{B}^k$, and so h_k has maximum entropy.

The second part is a bit string b_n of bit length n. □

The only information contained in the code, prior to memmbership test, is given by the probality mass function of the random bit length N, which is a function of the cardality of the finite set it is approximating.

The random bit length N of the string has a probability mass concentrated around

$$-\, m \log_2 \varepsilon + 1.44m \, . \tag{43}$$

Theorem 8.4. *An estimator of the cardinality of an approximated set represented by a Perfect Hash Filter is given by*

$$\hat{m} = -\frac{\ell(\mathcal{S}^{\varepsilon})}{-\log_2 \varepsilon + 1.44} \, , \tag{44}$$

were ℓ is the bit length function and ε is the false positive rate.

Proof. The *expected* bit length is given by

$$\mathrm{E}[\mathrm{N}] = -m \log_2 \varepsilon + 1.44m \, , \tag{?? revisited}$$

where m is the cardinality of the approximated set. Thus, the maximumn likelihood estimator is given by assuming the bit length of the Perfect Hash Filter realizes this expectation,

$$\ell(\mathcal{S}^{\varepsilon}) = -\hat{m} \log_2 \varepsilon + 1.44\hat{m} \, . \tag{a}$$

Solving for $\hat{m}$ results in the estimator

$$\hat{m} = \frac{\ell(\mathcal{S}^{\varepsilon})}{-\log_2 \varepsilon + 1.44} \, . \tag{b}$$

□

Theorem 8.5. *A* maximum likelihood estimator *of the cardinality of an approximated set represented by a Perfect Hash Filter is given by*

$$\hat{m} = -\frac{\ell(\mathcal{S}^{\varepsilon})}{-\log_2 \varepsilon + 1.44} \, , \tag{45}$$

were ℓ is the bit length function and ε is the false positive rate.

Proof. The *expected* bit length is given by

$$\mathrm{E}[\mathrm{N}] = -m \log_2 \varepsilon + 1.44m\,, \tag{?? revisited}$$

where m is the cardinality of the approximated set. Thus, the maximumn likelihood estimator is given by assuming the bit length of the Perfect Hash Filter realizes this expectation,

$$\ell(\mathcal{S}^\varepsilon) = -\hat{m} \log_2 \varepsilon + 1.44\hat{m}\,. \tag{a}$$

Solving for $\hat{m}$ results in the estimator

$$\hat{m} = \frac{\ell(\mathcal{S}^\varepsilon)}{-\log_2 \varepsilon + 1.44}\,. \tag{b}$$

□

However, there is actually a simpler method. Simply count the number of rows in the matrix $\boldsymbol{V}$.

We made trade space complexity for entropy if desired. For instance, if we find a bit string of a length around $t > -m \log_2 \varepsilon$ that codes the Perfect Hash Filter, then an estimate of the cardinality is given by

$$|\hat{\mathcal{S}}| = -\frac{t}{\log_2 \varepsilon}\,. \tag{46}$$

Thus, without knowing how t was chosen, we can only infer that the approximated set has a cardinality approximately between 0 and $-\frac{t}{\log_2 \varepsilon}$, which has an approximate entropy of $\log_2 t$. The coding efficiency is now given by

$$0 < \frac{m}{t} \leq 1\,, \tag{47}$$

which is the *maximum* efficiency possible for an approximate set with $\log_2 t$ entropy for the cardinality.

8.3 C++ implementation

The theoretical Perfect Hash Filter based off of the random oracle as described by [5] is not tractable. Its purpose was to demonstrate theoretical properties.

We use a fast state-of-the-art algorithm [2] for generating perfect hash functions that requires $\alpha(r)$ bits/element, where α is a function of the load factor r which does not obtain the *expected* lower-bound. The expected bit length of this Perfect Hash Filter is given by

$$-\frac{1}{r} \log_2 \varepsilon + \alpha(r) \text{ bits/element} \tag{48}$$

and its absolute efficiency is given by

$$\frac{-\frac{1}{r} \log_2 \varepsilon + \alpha(r)}{-\log_2 \varepsilon} \tag{49}$$

The relative efficiency of the Bloom filter to the perfect hash filter is given by

$$0.69 - \frac{\alpha(r)}{1.44 \log_2 \varepsilon} \tag{50}$$

As $\varepsilon \to 0$, the relative efficiency goes to 0.69. In the case of a the *minimal* perfect hash filter,

$$\alpha(1) = 2.07\,. \tag{51}$$

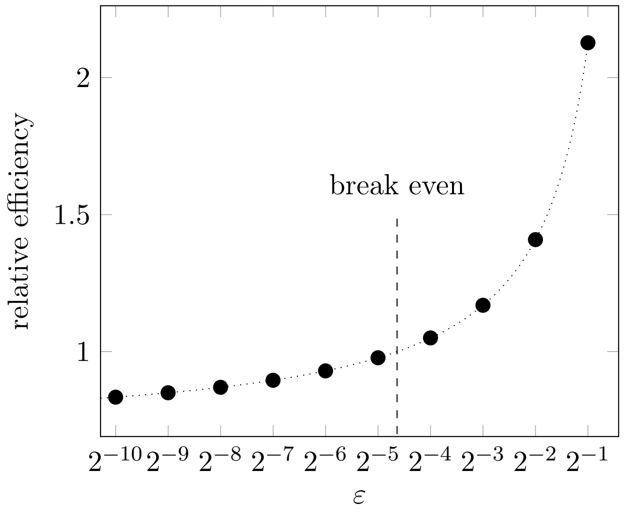
Listing 1: The C++ interface of the *static approximate set* abstract data type

```
// An approximate set S* of a set S may be considered an
// approximate set of S where:
//     (1) An element that is not a member of S is a
//         member of S* with probability false_positive_rate().
//     (2) An element that is a member of S is not a
//         member of S* with probability false_negative_rate().
template <typename X>
class approximate_set
{
public:
    // A probability_type is a probability and
    // thus should take on a value between 0 and 1.
    typedef double probability_type;

    // The cardinality_type is able to represent
    // the cardinality of sets. Since an approximate
    // set may have an uncertain number of false
    // positives or true positives, the
    // cardinality_type is a real number type.
    typedef double cardinality_type;

    // The value_type is the type of elements
    // in the set.
    typedef X value_type;

    // If x is in the approximated set S, returns
    // true with probability
    //     1 - false_negative_rate().
    // Otherwise, returns true with probability
    //     false_positive_rate().
    virtual
    bool contains(const X& x) const = 0;

    // The cardinality of the approximate set.
    // If the approximate set has an uncertain number
    // of false positives or true positives, the
    // expected cardinality may be returned.
    virtual
    cardinality_type cardinality() const = 0;

    // Returns the false positive rate,
    // the probability that an element
    // not in the approximated set S
    // is in the approximate set S*.
    virtual
    probability_type false_positive_rate() const = 0;

    // Returns the false negative rate,
    // the probability that an element
    // in the approximated set S
    // is not in the approximate set S*.
    virtual
    probability_type false_negative_rate() const = 0;
};
```

Listing 2: The C++ interface of the Perfect Hash Filter

```cpp
// The Perfect Hash Filter (PHF) is a positive
// approximate set. A positive approximate set
// S* is a superset of S where an element not
// in S is in S* with a probability
//     false_positive_rate().
template <class H>
class perfect_hash_filter:
    public approximate_set<byte_string>
{
public:
    typedef H perfect_hash_function;
    typedef load_factor_type;
    using approximate_set<byte_string>::value_type;
    using approximate_set<byte_string>::probability_type;
    using approximate_set<byte_string>::cardinality_type;


    // perfect_hash_function is a type that implements the
    // perfect hash function abstract data type. It should
    // implement two components:
    //     - a build method with a load factor
    //       parameter
    //     - an iterable collection of elements of
    //       type const char* or types that
    //       can be converted to const char*.
    perfect_hash_function get_perfect_hash_function() const;

    // Construct a Perfect Hash Filter (a positive approximate
    // set) with a false positive rate v and load factor r
    // for an iterable collection of byte_string types from
    // begin to end, where a byte_string is a sequence of bytes.
    //
    // The iterable collection is not mutated. If the operation
    // fails, a perfect_hash_filter_build_exception is thrown
    // and the state of the Perfect Hash Filter instance is
    // undefined.
    template <class Iter>
    perfect_hash_filter(
        load_factor_type r,
        probability_type v,
        Iter begin,
        Iter end)

    bit_matrix get_hashes() const;

    load_factor_type load_factor() const;

    // ****************************************
    // * approximate_set<byte_string> interface *
    // ****************************************
    bool contains(
        const char*& x) const;

    probability_type false_positive_rate() const;

    // The cardinality of an approximate set over the
    // countably infinite universe of byte_strings
    // is countably infinite.
    double cardinality() const;

private:
    // we again assume bytes, and thus the false positive
    // rate is the set given by
```

Figure 5: Relative efficiency of the Bloom filter to the *minimal* perfect hash filter



The relative efficiency of the Bloom filter to the *minimal* perfect hash filter is given by

$$\ln 2 - \frac{1.4375}{\log_2 \varepsilon}\,. \tag{52}$$

The point at which they are equally efficient efficient is $\varepsilon \approx 2^{-4}$. Thus, the *minimal* perfect hash filter is more efficient if $\varepsilon < 2^{-4}$ and otherwise the Bloom filter is more efficient. See fig. 5 for a visualization.

9 The *Typed* Perfect Hash Filter

The Perfect Hash Filter described in §8 is an random approximate set over the universe of bit strings $\mathcal{B}^*$. However, injections from a countable set $\mathcal{X}$ to the set $\mathcal{B}^*$ exist. Let us denote such an injection by

$$\texttt{encode}\colon \mathcal{X} \mapsto \mathcal{B}^*\,, \tag{53}$$

which maps every element $x \in \mathcal{X}$ to a unique element $y \in \mathcal{B}^*$. Note that in the Perfect Hash Filter, iterating over (or retrieving) elements is not supported and thus `encode` does not need to be decodable.

For instance, if elements of type $\mathcal{X}$ are a data structure, then `encode` may be its serialization as a sequence of bits.[13] Alternatively, if the data structure is intended to be efficiently inserted into a Perfect Hash Filter, then something like a simple *hash code* may be used since it does not need to be *decodable*.[14]

Consequently, we may map any set $\mathcal{S} \subset \mathcal{X}$ to one or more elements in the set $\mathcal{B}^*$ so that the Perfect Hash Filter may approximate it. We denote this the Perfect Hash Filter of type $\mathcal{X}$. The Perfect Hash Filter of type $\mathcal{X}$ may approximate any *finite set* $\mathcal{S}$ where the elements are members of the countable set $\mathcal{X}$. See algorithm 3 for an algorithmic description of `make_perfect_hash_filter` and algorithm 4 for an algorithmic description of `contains`.

[13]The *memory-resident* representation is machine and process-dependent and thus not a suitable representation if the *typed* Perfect Hash Filter is suppose to work across multiple machines or processes.

[14]If each element of $\mathcal{X}$ does not uniquely map to a hash code, then the false positive rate as given by ε is only an approximation.

Algorithm 3: Implementation of `make_perfect_hash_filter` for the *typed* Perfect Hash Filter

```
input  :
    $\mathcal{S}$ The set to approximate where the universe of elements is $\mathcal{X}$.
    $\varepsilon$ The false positive rate.
    $r$ The load factor.
output : A typed Perfect Hash Filter $\mathcal{S}^{\varepsilon}(\varepsilon)$ with a load factor $r$.
1 function make_typed_perfect_hash_filter($\mathcal{S}$, $\varepsilon \mid r$)
2     $\mathcal{A} \leftarrow \{$ encode$(x) \in \mathcal{B}^* \mid x \in \mathcal{S} \}$;
3     $\mathcal{S}^{\varepsilon}_{\mathcal{B}^*} \leftarrow$ make_perfect_hash_filter($\mathcal{A}$, $\varepsilon \mid r$);
4     return $\mathcal{S}^{\varepsilon}_{\mathcal{B}^*}$;
```

Algorithm 4: Implementation of `contains` for the *typed* Perfect Hash Filter

```
input  :
    $\mathbb{S}^{+}_{\mathcal{X}}$ The typed Perfect Hash Filter that approximates $\mathcal{S}_{\mathcal{X}} \subset \mathcal{X}$ with a false positive rate $\varepsilon$ and a
        load factor $r$.
    $u$ The element in $\mathcal{X}$ to test membership in $\mathbb{S}^{+}_{\mathcal{X}}$.
output : Returns true if $x \in \mathcal{S}_{\mathcal{X}}$. Otherwise, returns true with probability $\varepsilon$ and false
         with probability $1 - \varepsilon$.
1 function contains($\mathbb{S}^{+}_{\mathcal{X}}$, $x$)
2     $b \leftarrow$ encode$_{\mathcal{X}(x)}$;
3     $q \leftarrow$ contains($\mathbb{S}^{+}_{\mathcal{B}^*}$, $b$);
4     return $q$;
```

9.1 C++ implementation

Stuff for C++ goes here.

References

[1] Michael Mitzenmacher Andrei Broder. Network applications of bloom filters: A survey. 2005.

[2] Martin Dietzfelbinger Djamal Belazzougui, Fabiano C. Botelho. Compress, hash and displace. URL http://cmph.sourceforge.net/papers/esa09.pdf.

[3] Alexander Towell. The approximate set abstract data type. . URL http://example.com.

[4] Alexander Towell. The oblivious set abstract data type. . URL http://example.com.

[5] Alexander Towell. Random oracles that perfectly hash and are expected to obtain the lower-bound on space complexity. . URL http://example.com.