Incremental Sparse Binary Vector Similarity Search in High-Dimensional Space

Kirill Levchenko* Justin Ma† Zhen Xiao‡ Yin Zhang§

Abstract

Given a sparse binary matrix A and a sparse query vector x, can we efficiently identify the large entries of the matrix-vector product Ax? This problem occurs in document comparison, spam filtering, network intrusion detection, information retrieval, as well as other areas. We present an exact deterministic algorithm that takes advantage of the sparseness of A and x. Although in the worst case, the query complexity is linear in the number of rows of A, the amortized query complexity for a sequence of several similar queries depends only logarithmically on the size of A when the non-zero entries of A and the queries are distributed uniformly.

1 Introduction

Do the works of William Shakespeare contain any proverbs from the Book of Proverbs? How might we go about answering such a question? Given the complete works of Shakespeare and the Book of Proverbs in electronic form, we could, of course, simply search for the occurrence of each proverb. However searching for the exact occurrence won't quite do it: there are many translations of the Book of Proverbs into English, Shakespeare could have slightly changed the wording, and of course, errors could have been introduced in the process of digitizing the text. What we're really after is identifying "similar" fragments of text in Proverbs and Shakespeare. There are several ways of measuring text similarity: edit (Levenshtein) distance, LZ distance [9], q-gram distance [18], as well as many others.

In our case, let us use word frequency, used in document comparison (see, for example, [5]), to determine when two text fragments are similar: we will consider two fragments, of roughly the same size, to be similar if they have many words in common. This similarity measure has the advantage of being easier to compute compared to metrics such as edit distance, and it allows us to search our dictionary of proverbs efficiently.

More specifically, for each text fragment, we work with its characteristic vector: a binary vector with 1 in the i-th position if word i, from a canonical list of all n words in the English language, occurs in the fragment. We begin by constructing a dictionary of characteristic vectors for each proverb. Next, we scan each work of Shakespeare, considering a fragment of twenty-or-so consecutive words, form its characteristic vector, and then try to find a similar characteristic vector in the dictionary. A similar vector in the dictionary indicates a matching proverb.

*University of California San Diego, La Jolla, California. Part of this work was done while the author was at AT&T Labs, Florham Park, New Jersey. E-mail: klevchen@cs.ucsd.edu.

†University of California San Diego, La Jolla, California. E-mail: jtma@cs.ucsd.edu.

‡IBM T. J. Watson Research Center, Hawthorne, New York. This work was done while the author was at AT&T Labs, Florham Park, New Jersey. E-mail: xiaozhen@us.ibm.com.

§University of Texas, Austin, Texas. This work was done while the author was at AT&T Labs, Florham Park, New Jersey. E-mail: yzhang@cs.utexas.edu.

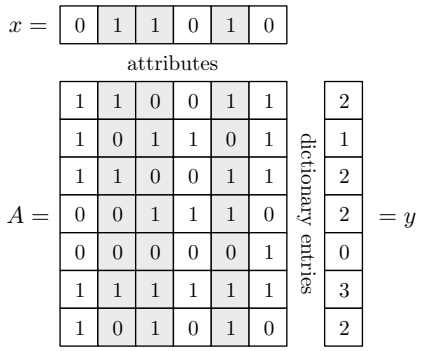

Figure 1: We can think of our problem as finding the indices of the vector $Ax = y$ that exceed the threshold t.

1.1 Problem Description

Formally, let A be a binary $m \times n$ matrix where each row has at most k ones. In our example, n is the number of words in the English language (about 500,000), m is the number of verses in the Book of Proverbs (about 900), and k is the text fragment size (about 20). After pre-processing A, we are given a query vector $x \in \{0,1\}^n$ with at most k ones (represented as a list of non-zero positions) and asked to return the set of all i such that $A_i x \geq t$, equivalently, the elements of the vector $y = Ax$ that are at least t. (see Fig. 1). Furthermore, motivated by our application, we would like the query algorithm to be incremental, so changing a constant number of bits in the query vector should not incur the full cost of a query. This allows us to slide a twenty-word fragment window over Shakespeare searching for matches incrementally, where each consecutive characteristic vector differs in two positions.

1.2 Other Applications

The problem of identifying approximate occurrences of strings occurs in areas well beyond the analysis of literary influences.

In a document comparison system (e.g., [5, 6, 17]) we are interested in determining if a given document contains a portion of a document from a known collection of documents (e.g., to detect plagiarism). The collection of known documents or their fragments make up the dictionary and correspond to rows of matrix A. Broder [5] suggests using *shingles*, sequences of a few (2-4) words, as the underlying vocabulary of a document so that each dictionary entry is a set of all shingles in the document. The conceptual size of the vector, namely the dimension n, is then in the order of $500,000^q$, where q is the number of words in a shingle.

The text matching problem also occurs in spam filtering, where the goal is to identify (and block) unsolicited commercial email. One approach to spam detection is for users to share *signatures* of unsolicited mail they receive, so that an unwanted message need only be seen by a small number of participants before it is blocked [1, 3, 12]. Early systems used a simple hash of the message body as the signature, which was easily evaded by adding random text, leading to an "arms race" between message obfuscation and signature robustness. To improve the latter, we may use vectors of lexical features as the signature, allowing the system to efficiently identify similar messages.

A similar approach is used within a network intrusion detection system monitoring incoming traffic for known computer viruses and worms. With a dictionary of known malicious payloads,

the system attempts to match incoming traffic payloads to dictionary entries to identify an attack (e.g., Bro [15] and Snort [2]). Because attackers can easily obfuscate the payload in order to avoid an exact match in the dictionary, intrusion detection system designers are turning to approximate matching.

1.3 Related Work

The more general problem of finding similar vectors has been studied in Information Retrieval (see, e.g. [16, 19]), where dictionary vectors represent document keywords and the target vector represents query keywords. The goal of an information retrieval system is to retrieve the top-r documents relevant to a query, where r is on the order of 10-100. Unlike our binary vector similarity measure, called *coordinate matching* in the Information Retrieval literature, typical systems use the cosine similarity measure. Most Information Retrieval systems answer such queries using an *inverted index*, a list from keywords to documents containing those keywords. The system only considers documents in the inverted indices of the query terms, greatly reducing the candidate search space. Search optimization has focused on how to combine these lists—typically by storing and traversing them in a certain order—with stopping conditions that guarantee that the top-r documents have been identified; for an overview, see [7]. Our algorithm may be regarded as belonging to the class of inverted index algorithms, however it cannot efficiently handle all similarity measures used in Information Retrieval. Additionally, our inverted index is stored as a tree rather than a list, and may not benefit from much of the work on compressing inverted indices.

Our problem can also be seen as a case of the *nearest neighbors* problem, which has been studied extensively for low-dimensional spaces and metric spaces. Recent work using hashing and projections [10, 11, 13, 14] attacks the high-dimensional case. Unfortunately, these approaches may not be completely suitable because of two characteristics of our problem:

Low Metric Information While the problem can be stated using a distance metric[1], such a metric does not provide enough information to take advantage of the triangle inequality. Intuitively, two vectors constructed as above will likely have no 1's in common and have a similarity of 0, leading to an exhaustive search through the dictionary.

Large Dimension The large dimensionality of the problem means that classic low-dimensional, and even new high-dimension algorithms with running times polynomial in the dimension size, are too costly.

Our algorithm may also be viewed as a generalization of the Aggregate Bit Vector algorithm of Baboescu and Varghese [4]. They were interested in efficiently computing the conjunction of several large bit vectors. Rather than computing the conjunction explicitly, in time linear in the size of the vectors, their algorithm first computes the conjunction of their *aggregate bit vectors*, where each element of this aggregate is a disjunction of a range of elements of the original vector; then, only parts of the original vectors corresponding to ones in the conjunction of the aggregates need to be examined.

Finally, work by Cohen and Lewis [8] bears resemblance to our work. They show how to approximate the matrix-vector product Ax, which gives the cosine similarity between the vector x and the rows of A. Applied to our problem, their algorithm works by sampling entries from the inverted indices such that the probability of choosing a document is proportional to the number of indices it appears in. The probability of correctly identifying a matching document depends on the number of documents in the inverted indices. For a fixed probability of correct identification,

[1] E.g., $d(u, v) = n - A_i x$.

the number of samples should be proportional the number of documents in the inverted indices. In contrast, our algorithm is deterministic, and avoids some of the problem cases of the sampling algorithm. A hybrid algorithm, applying the sampling algorithm after preprocessing with our algorithm, may combine the advantages of both—this is left as future work.

1.4 Organization

The remainder of the paper is organized as follows: We describe our algorithm in Section 2. Section 3 analyzes the complexity of the algorithm, followed by Section 4 which discusses extensions and improvements to the algorithm. Section 5 concludes the paper.

2 The Algorithm

Our algorithm has two stages: pre-processing the dictionary and answering queries. In the pre-processing stage, we construct an inverted index of A by representing each column as a *simple tree*. To answer a query, the simple trees corresponding to the non-zero elements of the query vector are combined into a *composite tree* which is used to form the result. We begin with some notation.

2.1 Notation

Let A, an $m \times n$ binary matrix whose rows constitute the dictionary vectors and where each row has at most k ones, be given. Without loss of generality, let m be a power of two.

We will be working with rooted, ordered binary trees. Define $\perp$ to be the empty tree. Define $\mathrm{N}(w)$ to be a function returning a new node with weight w and no children. Define $\mathrm{L}(u)$ to be the left child of tree node u and $\mathrm{R}(u)$ to be its right child; also, define $\mathrm{L}(\perp)$ and $\mathrm{R}(\perp)$ to be $\perp$. Define $\mathrm{W}(u)$ to be the weight of node u, and define $\mathrm{W}(\perp) = 0$. We will treat a tree and its root node as interchangeable.

Define the path label of tree node u, denoted $\pi(u)$, to be a binary string coding the path from the root of the tree to u, where 0 denotes a left edge and 1 denotes a right edge. We will use the Greek letters α and β to denote binary sequences. For a positive integer $i \in \{1, \ldots, m\}$, we use $\langle i \rangle$ to denote the $(\lg m)$-bit binary representation of $i - 1$, most significant bit first; e.g., $\langle 2 \rangle = 001$ for $m = 8$.

2.2 Pre-Processing

The matrix A may be regarded as a dictionary whose entries are the rows of A. The pre-processing stage consists of constructing an inverted index of A so that the product Ax may be computed efficiently from the sparse representation of x. For each column of A, we construct a *simple tree*: a complete binary tree on $2m - 1$ nodes, with leaves corresponding to the m elements of the column. Assign weight 1 to the i-th leaf of the simple tree if the i-th element of the column is 1 and assign weight 0 if the i-th element is 1. Weight each internal node with the maximum weight of its children (see Fig. 2). Denote by S_j the simple tree constructed from the j-th column of A. By construction, leaf u of S_j with path label $\pi(u) = \langle i \rangle$ has weight A_{ij}.

2.3 Answering Queries

Given a query vector x and a threshold t, we would like to return the set of all i where $A_i x \geq t$. We do this by building a *composite tree* from at most k simple trees corresponding to the non-zero elements of x. Without loss of generality, let the first k simple trees correspond to the non-zero

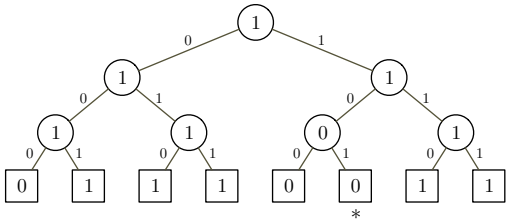

Figure 2: A simple tree corresponding to the column vector $(0, 1, 1, 1, 0, 0, 1, 1)^T$. The node labeled $*$ has path label 101.

elements of x. Conceptually, the composite tree T is constructed by "summing" the simple trees $S_1, \ldots, S_k$ and then removing the nodes whose weight is less than t. However by building the tree recursively starting from the root, we can avoid constructing subtrees with weight less than t. Formally:

```
    Algorithm Cmp(S_1, ..., S_k, t):
1       w ← 0,
2       For j from 1 to k, do
3           w ← w + W(S_j),
4       If w ≥ t, then
5           T ← N(w),
6           If S_1, ..., S_k are not leaves, then
7               L(T) ← Cmp(L(S_1), ..., L(S_k), t),
8               R(T) ← Cmp(R(S_1), ..., R(S_k), t)
9           end,
10          Return T
11      else
12          Return ⊥
13      end
14  end.
```

Starting at the root, the Cmp algorithm sets the weight of the composite node to the sum of the weights of the simple tree nodes. It then recursively constructs the left and right subtree using the left and right subtrees, respectively, of the simple trees $S_1, \ldots, S_k$ (lines 7 and 8). Construction terminates when the algorithm reaches a leaf (line 6) or the sum of the node weights of simple tree is less than t (line 4).

The leaves of the composite tree T at depth $\lg m$ correspond to the rows of A_i where $A_i x \geq t$. We prove this claim in Theorem 4, however we first need the following definition and two lemmas about the algorithm.

Lemma 1. *Let u be the node of the composite tree created on line 5 upon some invocation of the algorithm during the construction of the full composite tree, and let $u_1, \ldots, u_k$ be the root nodes of subtrees $S_1, \ldots, S_k$ with which the algorithm was invoked. Then $\pi(u) = \pi(u_1) = \cdots = \pi(u_k)$.*

Proof. Immediate from the recursion structure of the algorithm. □

Definition 2. *Define $\sigma(\alpha)$ to be the sum of the weights of nodes in $S_1, \ldots, S_k$ whose path label is α. That is, $\sigma(\alpha) = \mathrm{W}(u_1) + \cdots + \mathrm{W}(u_k)$, where $u_1, \ldots, u_k$ are in $S_1, \ldots, S_k$, respectively, and $\pi(u_1) = \cdots = \pi(u_k) = \alpha$.*

Lemma 3 (Monotonicity of σ)**.** *If α and β are binary strings with combined length at most k, then $\sigma(\alpha) \geq \sigma(\alpha\beta)$.*

Proof. By construction, the node weights along a path in a simple tree from a node to the root are non-decreasing. It follows that the sum of such nodes across k trees is non-decreasing also. □

We are now ready to prove the correctness of the Cmp algorithm.

Theorem 4. *Let $T = \mathsf{Cmp}(S_1, \ldots, S_k, t)$ and $y = Ax$. Then T has a leaf u with length-$(\lg m)$ path label $\pi(u) = \langle i \rangle$ if and only if $A_i x \geq t$.*

Proof. Consider a leaf u of T at depth $\lg m$ with path label $\pi(u) = \langle i \rangle$, and let $u_1, \ldots, u_k$ be the leaves of $S_1, \ldots, S_k$ with the same path label, i.e., $\pi(u) = \pi(u_1) = \cdots = \pi(u_k)$. From line 4 of the algorithm, it follows that $\sigma(\pi(u)) = w \geq t$, so at least t of the weights of $u_1, \ldots, u_k$ must be 1 (recall that simple tree weights are binary), so row A_i must have at least t ones in the first k positions. This proves the forward direction.

Toward a contradiction, let $A_i x \geq t$, and let T not contain a node with path label $\langle i \rangle$. Since $A_i x \geq t$, row A_i has at least t ones in the first k positions, so at least t of the simple trees $S_1, \ldots, S_k$ have a node with weight 1 with path label $\langle i \rangle$. But then $s(\langle i \rangle) \geq t$. Let u' be an internal node of T such that $\pi(u')$ is a prefix of $\langle i \rangle$ and such that u' does not have a child whose path label would also be a prefix of $\langle i \rangle$. Intuitively, u' is the last node created in T on a hypothetical path to the non-existent leaf. Without loss of generality, let this non-existent child be a left child, so that it's prefix is $\pi(u')0$. After constructing u', Cmp was called recursively on the subtrees of $S_1, \ldots, S_k$ rooted at nodes $u''_1, \ldots, u''_k$, respectively, where $\pi(u''_1) = \cdots = \pi(u''_k) = \pi(u')0$. But $s(\pi(u')0) \geq s(\langle i \rangle) \geq t$ by Lemma 3, so the recursive call above would have constructed a node (lines 4 and 5), contradicting u' being the node in T with the longest prefix of $\langle i \rangle$. □

2.4 Incremental Construction

It turns out that the composite tree is amenable to incremental construction. Given a composite tree T for a query vector x and a new query vector x' that differs from x in one bit position, we can update T without building a new composite tree. Without loss of generality, let x and x' differ in the first position and have ones in positions 2 through k. If $x'_1 = 0$, then we need to update T by "removing" the simple tree S_1. If $x'_1 = 1$, then we need to update T by "adding" the simple tree S_1.

Let $x'_1 = 1$. The following algorithm takes a composite tree T, constructed for query x from $S_2, \ldots, S_k$, and returns a composite tree T' for query x', using subtrees of T where T and T' are identical.

```
    Algorithm Add(T, S_1, ..., S_k, t):
1     If W(S_1) = 0, then
2       Return T
3     else
4       w ← 0,
5       For i from 1 to k, do
6         w ← w + W(S_i),
7       If w ≥ t, then
8         T' ← N(w),
9         If S_1, ..., S_k are not leaves, then
10          L(T') ← Add(L(T), L(S_1), ..., L(S_k), t),
11          R(T') ← Add(R(T), R(S_1), ..., R(S_k), t)
12        end,
```

```
13            Return T'
14          else
15            Return ⊥
16          end
17        end.
```

For brevity, we omit the Rem algorithm for the case $x'_1 = 0$.

Theorem 5. *The output of*

$$\mathsf{Cmp}(S_1, \ldots, S_k, t) \quad \textit{and}$$
$$\mathsf{Add}(\mathsf{Cmp}(S_2, \ldots, S_k, t), S_1, \ldots, S_k, t)$$

is identical. The output of

$$\mathsf{Cmp}(S_2, \ldots, S_k, t) \quad \textit{and}$$
$$\mathsf{Rem}(\mathsf{Cmp}(S_1, \ldots, S_k, t), S_1, \ldots, S_k, t)$$

is identical.

Proof. We will prove the second statement. The proof of the first statement is similar. The proof is by induction of the height of the trees $T, S_1, \ldots, S_k$. The base case is height 0, i.e., $T = S_1 = \cdots = S_k = \bot$, in which case the output is identical. Now consider an invocation of Add with parameters of height h, and assume that the algorithm is correct for trees of height less than h.

If $\mathrm{W}(S_1) = 0$ in line 1 of the algorithm, then all the nodes in the subtree S_1 have weight 0, so $T = \mathsf{Cmp}(S_2, \ldots, S_k, t)$ is identical to $\mathsf{Cmp}(S_1, \ldots, S_k, t)$.

If $\mathrm{W}(S_1) = 1$ in line 1 of the algorithm, then lines 4-15 are executed. By induction, the result of lines 10 and 11 is identical to lines 7 and 8 of Cmp, and therefore lines 4-15 are identical to executing $\mathsf{Cmp}(S_1, \ldots, S_k, t)$. □

3 Analysis

Ideally, we would like the query complexity to depend only on the size of the result $\{i \mid A_i x \geq t\}$. Unfortunately, there exist dictionaries and queries where our algorithm takes time linear in the size of the dictionary. Consider the following: Choose k even, $t = k/2 + 1$, $m = 2^{k/2}$, $n \geq k$. Let each row of A have exactly $k/2$ ones in the first k positions and zero in the remaining $n - k$ positions, such that no two rows have the same arrangement of ones. Let the query vector consist of k ones in the first k positions, and zeroes in the remaining $n - k$ positions. No row alone has $t = k/2 + 1$ ones, however the disjunction of any two rows has at least t. Running the Cmp algorithm would result in a complete tree of depth $\lg m - 1$, even though the query result is empty.

Recall, however, that we are interested in performing many similar queries, modifying the composite tree incrementally using Add and Rem. We begin by noting the time complexity of the algorithms.

Lemma 6. Cmp *takes time $O(k|T|)$, where T is the resulting composite tree.* Add *and* Rem *take time $O(k|S_1|)$.*

Let c_j be the number of non-zero entries in column j of A. Then,

Lemma 7. *The size of S_j is at most $c_j \lg m$.*

Now consider a sequence of queries

$$\vec{0} = v_0, v_1, \ldots, v_\ell,$$

each differing from the preceding in exactly one position. As usual, let $v_1, \ldots, v_\ell$ each have at most k ones. Let b_j be the number of times the value of element j changes between consecutive vectors.

We make note of the following two equalities.

$$\sum_{j=1}^{n} b_j = \ell \qquad \sum_{j=1}^{n} c_j \leq km.$$

We are now ready to prove the main theorem of this section.

Theorem 8. *The amortized cost per query is*

$$O\Big((\sum_{j=1}^{n} b_j c_j)\frac{k \lg m}{\ell}\Big).$$

Proof. Let j_r be the element of the query vector that changes at time r. The total cost of the ℓ queries is

$$\begin{aligned}
\sum_{r=1}^{\ell} O(k|S_{j_r}|) &= \sum_{r=1}^{\ell} O(kc_{j_r} \lg m) \qquad (*) \\
&= \sum_{j=1}^{n} O(b_j c_j k \lg m) \\
&= O\Big((\sum_{j=1}^{n} b_j c_j) k \lg m\Big),
\end{aligned}$$

where $(*)$ follows by Lemmas 6 and 7. Dividing by the number of queries gives the result. □

Corollary 9. *Let the non-zero entries be distributed uniformly across the columns of A and let each element of the query vectors $v_1, \ldots, v_\ell$ change equally often. Then the amortized cost per query is*

$$O(\frac{mk^2 \lg m}{n}).$$

Proof. Set $b_j = \frac{\ell}{n}$ and $c_j = \frac{km}{n}$. □

Real-world distributions are rarely uniform. In many cases, a handful of attributes occur much more often than the rest. We model this by considering the set of "popular" attributes separately: set γ to be the fraction of attributes that together represents half of all attribute occurrences; e.g., set $\gamma = 0.1$ if 10% of the words in an English dictionary make up half of the words in a book.

Corollary 10. *Let $0 < \gamma \leq \frac{1}{2}$. Let the b_j be distributed as follows*

$$b_j = \begin{cases} \frac{1}{2}\ell \cdot \frac{1}{\gamma n} & \text{if } j \leq \gamma n, \\ \frac{1}{2}\ell \cdot \frac{1}{(1-\gamma)n} & \text{if } j > \gamma n. \end{cases}$$

Let the c_j be distributed as follows

$$c_j = \begin{cases} \frac{1}{2}km \cdot \frac{1}{\gamma n} & \text{if } j \le \gamma n, \\ \frac{1}{2}\ell \cdot \frac{1}{(1-\gamma)n} & \text{if } j > \gamma n. \end{cases}$$

Then the amortized cost per query is

$$O(\frac{mk^2 \lg m}{\gamma n}).$$

Proof. Applying the definitions of b_j and c_j in the statement,

$$\begin{aligned} 4\sum_{j=1}^{n} b_j c_j &= \sum_{j=1}^{\gamma n} \frac{\ell}{\gamma n} \cdot \frac{km}{\gamma n} \\ &\quad + \sum_{j=\gamma n+1}^{n} \frac{\ell}{(1-\gamma)n} \cdot \frac{\ell}{(1-\gamma)n} \\ &= \frac{\ell km}{n^2}\Big(\sum_{j=1}^{\gamma n} \frac{1}{\gamma^2} + \sum_{j=\gamma n+1}^{n} \frac{1}{(1-\gamma)^2}\Big) \\ &= \frac{\ell km}{n}\Big(\frac{1}{\gamma(1-\gamma)}\Big). \end{aligned}$$

□

4 Extensions

In some cases, it is desirable to assign weights to attributes (columns of A). Such an extension requires a trivial change of line 3 of algorithm Cmp to a weighted sum. It is also possible to have individually weighted attributes within each vector, such as when an attribute occurs more than once. In this case, the simple tree nodes need to be weighted accordingly, rather than binary. In general, however, when the weight distribution is far from uniform, the sampling approach of Cohen and Lewis [8] may be more appropriate.

5 Conclusion

We presented and analyzed an algorithm for finding similar sparse vectors in high-dimensional space. The sparseness of the vectors allowed us to use an inverted index to limit the number of candidates. By arranging the inverted lists into trees with the same partitioning at each level, we are able to eliminate additional candidates. Furthermore, the composite tree representing the set of query results can also be maintained incrementally, allowing us to execute a sequence of similar queries more efficiently. Specifically, when the number of dimensions is on the order of the number of entries, i.e., $m = O(n)$ and the attributes are distributed uniformly, the amortized cost per query depends polynomially only on k.

5.1 Future Work

The incremental algorithm is being implemented in an experimental intrusion detection system, where it is used to match byte fragments, represented as q-gram vectors, against a dictionary of

known malicious payloads. With a dynamically constructed dictionary, the system will also be used to extract frequently-occurring payload fragments.

A drawback of our algorithm is its potentially linear (in the size of the dictionary) worst-case query time, even when the size of the result set is empty. For the purposes of improving this worst-case behavior, we are considering a hybrid approach combining the work of Cohen and Lewis [8].

References

[1] Distributed checksum clearinghouse. `http://www.rhyolite.com/anti-spam/dcc/`.

[2] Snort. `http://www.snort.org/`.

[3] Vipul's razor. `http://razor.sourceforge.net/`.

[4] F. Baboescu and G. Varghese. Scalable packet classification. *Proc. of the 2002 ACM SIGCOMM Conference*, pages 199–210, 2001.

[5] A. Broder. On the resemblance and containment of documents. In *Proceedings of Compression and Complexity of Sequences*, 1997.

[6] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic clustering of the web. In *Proceedings of the Sixth International Conference on World Wide Web*, pages 1157–1166, 1997.

[7] C. Buckley and A. F. Lewit. Optimization of inverted vector searches. In *Proceedings of the Eighth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 97–110, 1985.

[8] E. Cohen and D. D. Lewis. Approximating matrix multiplication for pattern recognition tasks. In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 682–691. SIAM, 1997.

[9] G. Cormode, M. Paterson, S. C. Sahinalp, and U. Vishkin. Communication complexity of document exchange. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 197–206, 2000.

[10] D. Dolev, Y. Harari, N. Linial, N. Nissan, and M. Parnas. Neighborhood preserving hashing and approximate queries. *SIAM Journal on Discrete Mathematics*, 15(1):73–85, 2002.

[11] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proceedings of the 25th VLDB Conference*, pages 518–529, 1999.

[12] A. Gray and M. Haahr. Personalized, collaborative spam filtering. In *Proceedings of the First Conference on Email and Anti-Spam*, 2004.

[13] J. M. Kleinberg. Two algorithms for nearest-neighbor search in high dimensions. In *Proceedings of the 29th Symposium on the Theory of Computing*, pages 599–608, 1997.

[14] E. Kushilevitz, R. Ostrovsky, and Y. Rabani. Efficient search for approximate nearest neighbor in high dimensional spaces. *SIAM Journal on Computing*, 30(2):457–474, 2000.

[15] V. Paxson. Bro: A system for detecting network intruders in real-time. *Computer Networks*, 31(23-24):2435–2463, December 1999.

[16] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, 1983.

[17] N. Shivakumar and H. Garcia-Molina. SCAM: A copy detection mechanism for digital documents. In *Proceedings of the Second International Conference on Theory and Practice of Digital Libraries*, 1995.

[18] E. Ukkonen. Approximate string-matching with q-grams and maximal matches. *Theoretical Computer Science*, 92(1):191–211, 1992.

[19] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, 2nd edition, 1999.