Succinct Interval-Splitting Tree for Scalable Similarity Search of Compound-Protein Pairs with Property Constraints

Yasuo Tabei
Japan Science and Technology Agency, Japan
tabei.y.aa@m.titech.ac.jp

Akihiro Kishimoto
IBM Research, Dublin, Ireland
AKIHIROK@ie.ibm.com

Masaaki Kotera
Kyoto University, Japan
kot@kuicr.kyoto-u.ac.jp

Yoshihiro Yamanishi
Kyushu University, Japan
yamanishi@bioreg.kyushu-u.ac.jp

ABSTRACT

Analyzing functional interactions between small compounds and proteins is indispensable in genomic drug discovery. Since rich information on various compound-protein interactions is available in recent molecular databases, strong demands for making best use of such databases require to invent powerful methods to help us find new functional compound-protein pairs on a large scale. We present the succinct interval-splitting tree algorithm (SITA) that efficiently performs similarity search in databases for compound-protein pairs with respect to both binary fingerprints and real-valued properties. SITA achieves both time and space efficiency by developing the data structure called interval-splitting trees, which enables to efficiently prune the useless portions of search space, and by incorporating the ideas behind wavelet tree, a succinct data structure to compactly represent trees. We experimentally test SITA on the ability to retrieve similar compound-protein pairs/substrate-product pairs for a query from large databases with over 200 million compound-protein pairs/substrate-product pairs and show that SITA performs better than other possible approaches.

Categories and Subject Descriptors

H.3.1 [**Content Analysis and Indexing**]: Indexing methods

General Terms

Algorithms, Design

Keywords

Succinct data structure, wavelet tree, similarity search

1. INTRODUCTION

Since most drugs are small compounds that interact with their target proteins and modulate the biological functions of the proteins, analyzing functional interactions between small compounds and proteins plays an important part in genomic drug discovery [28, 9]. A vast number of small compounds [5, 6], all possible proteins coded in human genomes [8, 10], and various functional interactions of compound-protein pairs [17, 14, 33, 13] are stored in many recent databases. There is therefore a strong demand for developing powerful methods to make best use of such databases and to find new functional compound-protein pairs on a large scale.

Searching for similar compound-protein pairs in databases for a given query is an example of effective use of databases with rich information. However, finding such pairs in a short time remains a challenge, since the number of all possible compound-protein pairs is calculated by the product of the number of compounds and the number of proteins, which makes the database preserving compound-protein pairs extremely large. So far, although many databases support options of similarity search for either compounds [5, 6] or proteins [8, 10], no database supports queries for similar compound-protein pairs due to a large amount of memory and excessive runtime required by existing techniques.

Fingerprint, defined as a binary bit string, is a powerful representation of various bio-molecules including small compounds and proteins [31]. In practice, the fingerprint representation is commonly used in molecular databases [5, 6], because of its facility of recording the presence or absence of molecular substructures and physicochemical features. Jaccard similarity (aka Tanimoto similarity) is the *de facto* standard criterion [18] to evaluate similarity of compounds based on their fingerprints in chemoinformatics and pharmacology. Similarity search of compounds for a query is usually performed with Jaccard similarity. Despite many attempts [16, 29, 3, 22, 1], leveraging *multibit tree* (MT) is the most efficient approach using the upperbounds of Jaccard similarity [16]. By splitting the database into clusters of fingerprints and then building binary trees that recursively split clustered fingerprints with their upperbound information, MT can prune out useless portions of the search space when searching for similar fingerprints to a query.

Bio-molecules have several important properties such as LogP, topological polar surface area, and autocorrelation polarizability. These properties are represented by real values rather than binary values. In the context of compound-protein pair search, it would be more rational to find similar compound-protein pairs in terms of both binary fingerprints and real-valued properties. For example, PubChem supports such a property similarity search, but the usability is currently limited to searching for compounds only [5].

When an MT-based algorithm is used to search for similar fingerprints with similar properties for a query, the algorithm first narrows down candidate fingerprints by using the aforementioned MT-based technique for finding similar fingerprints. It then checks whether each candidate fingerprint also has a similar property to the query or not. However, this approach suffers from performance degradation caused by intensively checking one-by-one the similarity of the property when the number of candidate fingerprints is large. Although any algorithm might inherently have this problem with large databases or with small thresholds of Jaccard similarity, the problem is especially pressing in MT-based approaches. Additionally, despite many attempts to achieve high-performance similarity search algorithms in data mining (e.g., [19, 21, 20]), these algorithms must also perform a large number of one-by-one checks when they are adapted to similarity search for compound-protein pairs.

We present a novel method called SITA that efficiently performs similarity search for compound-protein pairs with respect to both binary fingerprints and real-valued properties. As in MT, the database is split into clusters of fingerprints to narrow down candidate fingerprints. However, unlike MT, SITA safely excludes the fingerprints that never have similar properties by sorting the fingerprints in order of properties and by considering the necessary *interval* of the sorted fingerprints. Besides, SITA's recursive search in a binary tree with two pruning conditions bypasses the procedure of checking one-by-one the similarity of the fingerprint and its property. Finally, while preserving the same time complexity, SITA successfully reduces the memory usage by using the rank dictionary [25] of the wavelet tree [12, 7] that is a succinct, memory-efficient data structure. By synthesizing these techniques, SITA's time complexity is *output-sensitive*. That is, the smaller the threshold is, the more quickly SITA terminates than the existing algorithms.

Experiments were performed on retrieving similar compound-protein/substrate-product pairs for a query from large databases with over 210 million pairs. The performance comparison with other algorithms demonstrates SITA's superiority.

2. SIMILARITY SEARCH PROBLEM FOR COMPOUND-PROTEIN PAIRS

We formulate the similarity search problem for compound-protein pairs with the fingerprint representation. A fingerprint is a fixed-length bit string and is conceptually equivalent to the set that contains element i if i-th bit of the fingerprint is 1. For clarity, notation x_i denotes the bit string representation of a fingerprint, while notation W_i corresponds to its set representation. A fingerprint database contains n compound-protein pairs $x_1, \cdots, x_n$, where each compound-protein pair is represented by a fingerprint concatenating the compound fingerprint and the protein fingerprint. A word w is an element in W_i and $|W_i|$ denotes the cardinality of W_i, i.e., the number of words in W_i. $F(W)$ is the property of a compound-protein pair W where F maps W to a real number. There are a number of ways to define F in practice (e.g., acidic group count and ALOGP descriptors) [27].

Jaccard similarity for W and W' is defined as $J(W, W') = \dfrac{|W \cap W'|}{|W \cup W'|}$. Given a query compound-protein pair Q, the task of similarity search is to retrieve from the database all the identifiers of fingerprints $P_{NC} = \{i_1, i_2, \cdots, i_k\}$ that satisfy the following two constraints for each j $(1 \le j \le k)$:

$$\epsilon \le J(W_{i_j}, Q), \tag{1}$$

$$F(Q) - \delta \le F(W_{i_j}) \le F(Q) + \delta, \tag{2}$$

where ϵ and δ are user-defined thresholds of Jaccard similarity and the similarity of the property, respectively.

3. LITERATURE REVIEW

Several efficient algorithms have been presented to find similar fingerprints satisfying only constraint (1) in both chemoinformatics and data mining. The following subsections review the literature in these research areas.

3.1 Related Work on Chemoinformatics

To our best knowledge, in chemoinformatics, most algorithms try employing better bounds of Jaccard similarity to reduce the number of similarity checks. For example, tight bounds are computed with XOR operations in [3]. Approaches to divide the fingerprint database into several blocks are presented [22, 16, 1]. They reduce candidate fingerprints since a tighter bound is obtained by applying Baldi et al.'s method to each block. Despite their advantage, their scalability is limited with respect to the database size.

Kristensen et al. [16] present indexing data structures called *multibit tree* (MT) and overcome the scalability problem. Their algorithm first clusters fingerprints with the same cardinality into a block and computes Swamidass and Baldi's bound [29] for each block to exclude useless blocks that never satisfy constraint (1). To efficiently search for similar fingerprints to query Q in the unfiltered blocks, each block is represented as a binary tree where each node represents a set of fingerprints. In the binary tree, fingerprints for a node are disjointly and recursively split to the left and right children according to whether a word chosen by entropy maximization is included in each fingerprint or not until each leaf contains at most a certain number of fingerprints. In search of similar fingerprints in the block, the algorithm recursively traverses the binary tree from the root and computes at each node the upperbound of the similarity based on [2]. If the upperbound proves that the fingerprints there are not similar to Q, the subtree rooted at that node is pruned. If the algorithm reaches a leaf, fingerprints at that leaf are included as candidates of similar ones. After the tree traversal, the algorithm finally calculates Jaccard similarity between each pair of those fingerprints and a query for validating the constraint (1). These procedures are repeated until all the similar fingerprints are retrieved from the database. Recently, Nasr et al. [23] theoretically analyze MT, and derived a pruning probability of the search space. At present, MT still remains as the most efficient algorithm in chemoinformatics.

3.2 Related Work on Data Mining

Table 1: Summary of similarity search methods

	Memory requirement	Data structure	Accuracy of solutions
Multibit Tree [16]	$O((2\|B^c\|-1)\log(2\|B^c\|-1)+2W^{max}+N^c\log W^{max})$	Tree	Exact
DivideSkip [19]	$O(N^c\log\|B^c\|)$	Inverted index	Exact
b-bit minhash [21, 20]	$O(\|B^c\|+N^c\log W^{max})$	String	Approximate
SITA	$O(N^c\log\|B^c\|)$	Succinct tree	Exact

Several algorithms presented in data mining can be used to find similar fingerprints satisfying constraint (1).

DivideSkip [19] uses *inverted index*, an associative array whose key is a word and whose value is a list of fingerprint ids containing that word. Inverted-index-based methods have also been proposed to find all the pairs of similar fingerprints from a given collection of fingerprints [34, 26].

Although DivideSkip was originally used to solve the problem of finding all the fingerprints sharing at least T common words to query Q, the task of finding all the fingerprints satisfying constraint (1) is essentially the same as the problem in [19]. DivideSkip obtains $|Q|$ lists by retrieving the list of fingerprint ids for each word in query Q. Those lists are then divided into a set L_l of *long lists* and the set L_s of $|Q|-|L_l|$ of *short lists*. DivideSkip relies on the fact that fingerprint id i appears at least $T-|L_l|$ times on the short lists if two fingerprints W_i and Q share at least T common words. For each of such ids found on L_s, DivideSkip checks if the id appears at least T times on L_l. If this is the case, the fingerprint with that id has at least T common words with Q. If $|L_s|$ is small, DivideSkip performs efficiently by filtering out the ids that appear less than $T-|L_s|$ times. However, DivideSkip often suffers from performance degradation caused by large $|L_s|$ with a large-scale database containing over hundreds of millions of fingerprints (see Section 5).

The b-bit minwise hashing (b-bit minhash) algorithm [21, 20] performs a random projection [4] from a fingerprint to a fixed-length string. Each character of the string consists of a b-bit symbol. Jaccard similarity between two fingerprints is then *approximately* equal to the Hamming distance between the strings of the fingerprints. In the prepossessing phase, all the fingerprints in the database are projected to their corresponding strings calculated by b-bit minhash and they are sorted in lexicographical order. When finding similar fingerprints to query Q, this approach first calculates the string for Q by using b-bit minhash. It then performs binary search to find the strings of which Hamming distances are similar to Q's string. Although b-bit minhash is known to be both time and memory efficient, there are cases of which strings with similar Hamming distances are not similar with respect to Jaccard similarity (i.e., false positive) and of which fingerprints that are similar in terms of Jaccard similarity do not have similar Hamming distances (i.e., false negative). To filter out false positives, b-bit minhash must check if each "similar" fingerprint is similar to Q with respect to Jaccard similarity, which may result in a slowdown of the speed. Additionally, because of the possibility of false negatives, b-bit minhash might fail to return similar fingerprints to Q. Although the probability of occurring false negatives can be reduced by increasing the value of b, this increase may slow down the speed of the algorithm.

Despite the importance of considering the similarity of both fingerprints and properties, no prior work exists considering constraints (1) and (2) in chemoinformatics and data mining. The current best possible approach is to first exclude the fingerprints that break constraint (1) by using algorithms reviewed in this section and then perform checking whether or not each unfiltered fingerprint satisfies constraint (2). However, this one-by-one-check procedure for constraint (2) must be performed many times, if there are a large number of fingerprints satisfying constraint (1), which occurs with a large database or with small ϵ, resulting in intensive computation.

4. SITA

SITA splits the database into blocks to examine only a necessary set of blocks. Each block is constructed as a binary tree with the notion of intervals and depth-first search is performed with subtree pruning schemes to improve the efficiency. SITA further incorporates the ideas behind the inverted index and the wavelet tree to reduce the memory requirement and to preserve the search efficiency.

Table 1 summarizes the characteristics of the representative algorithms and SITA, where B^c is a block of fingerprints with cardinality c in the database, W^{max} is the maximum word in all W_i belonging to B^c and N^c is the total number of words in B^c. Unlike b-bit minhash, SITA guarantees correctness. Additionally, in terms of space complexity, SITA requires a similar amount of memory to DivideSkip, which requires less memory than MT. Furthermore, SITA is more time-efficient than the other algorithms for finding fingerprints that satisfy constraints (1) and (2) (see Section 5).

4.1 Database Partitioning

Swamidass and Baldi [29] show that $\epsilon|Q| \le |W| \le \frac{|Q|}{\epsilon}$ holds if $J(W,Q) \ge \epsilon$. This indicates that $P_1 = \{i; \epsilon|Q| \le |W_i| \le \frac{|Q|}{\epsilon}\}$ must contain all elements in P_{NC} (i.e., $P_{NC} \subseteq P_1$). That is, a fingerprint id that is not in P_1 is never a member of P_{NC}. As in [29, 16], excluding such useless fingerprints can be performed efficiently by partitioning the database into blocks each of which contains fingerprint ids with the same cardinality. More specifically, let block $B^c = \{i; |W_i| = c\}$, which contains all the fingerprints with cardinality c in the database. SITA then need to examine no element in B^c if either $c < \epsilon|Q|$ or $c > \frac{|Q|}{\epsilon}$ holds.

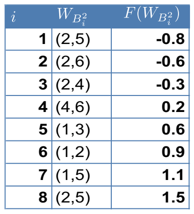
4.2 Interval-Splitting Tree and Efficient Similarity Search

4.2.1 Interval-Splitting Tree

Once blocks B^c are selected that satisfy $\epsilon|Q| \le c \le \frac{|Q|}{\epsilon}$, SITA bypasses one-by-one checks with constraint (2) for each element in B^c.

First, fingerprint ids in B^c are sorted in ascending order of properties and are saved as an array (see the left of Figure 1). For simplicity, let B^c be the sorted block with cardinality c.

A binary tree T^c called the *interval-splitting tree* is then built on each B^c beforehand. When a query is given, T^c is traversed with pruning schemes to efficiently select all the ids of fingerprints with cardinality c that satisfy constraints



i	$W_{B^2_i}$	$F(W_{B^2_i})$
1	(2,5)	-0.8
2	(2,6)	-0.6
3	(2,4)	-0.3
4	(4,6)	0.2
5	(1,3)	0.6
6	(1,2)	0.9
7	(1,5)	1.1
8	(2,5)	1.5

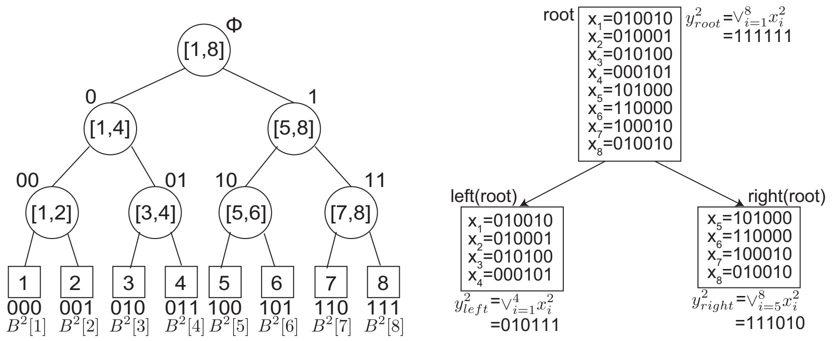


Figure 1: Fingerprints in block B^c sorted by properties (left), interval-splitting tree T^c (middle) and T^c's first two levels (right). The root interval $[1,8]$ is split into $[1,4]$ and $[5,8]$ for the left and right children. Each node v has a summary fingerprint y_v for the fingerprints in its interval.

(1) and (2). Each node in T^c represents a set of fingerprint ids by using an *interval* of the array of B^c, which is different from MT. Let $B^c[i]$ be the i-th fingerprint id in B^c and I^c_v be the interval of node v. Node v with interval $I^c_v = [s,e]$ contains fingerprint ids $B^c[s], B^c[s+1], \cdots, B^c[e]$. The interval of a leaf is always in the form of $[s,s]$ indicating that the leaf has only one id. The interval of the root is $[1, |B^c|]$, the whole range of B^c.

Let $left(v)$ and $right(v)$ be the left and right children of node v with interval $I^c_v = [s,e]$, respectively. When these children are generated, I^c_v is disjointly partitioned to $I^c_{left(v)} = [s, \lfloor (s+e)/2 \rfloor]$ and $I^c_{right(v)} = [\lfloor (s+e)/2 \rfloor + 1, e]$. The procedure of splitting the interval is recursively applied from the root to leaves (see the middle and right of Figure 1 illustrating intervals and sets of fingerprints at the root and its children of T^c).

Each node v is identified by a bit string (e.g., $v = 010$) indicating the path from the root to v; "0" and "1" denote the selection of left and right children, respectively. At each leaf v, the index of B^c is calculated by $int(v)+1$, where $int(\cdot)$ converts a bit string to its corresponding integer (see the middle of Figure 1 again).

4.2.2 *Pruning Based on Intervals and Summary Fingerprints*

Given query Q, SITA recursively examines T^c from the root in a depth-first manner. If SITA reaches a leaf and its fingerprint and property are similar to Q, the id of that fingerprint is included as one solution. To avoid exploring the whole T^c, we develop two schemes to prune the subtrees of nodes if all the fingerprints for these nodes are proven to break constraint (1) or (2).

The first pruning scheme refers to constraint (2). Before starting to examine T^c, by using constraint (2), SITA calculates the necessary condition of the interval I^c_q that indicates where candidate fingerprints for B^c should be located. $I^c_q = [i^c_l, i^c_r]$ is calculated as follows:

$$\begin{aligned} i^c_l &\leftarrow \min\{i; F(Q) - \delta \leq F(W_{B^c[i]}), i \in [1, |B^c|]\}, \\ i^c_r &\leftarrow \max\{i; F(W_{B^c[i]}) \leq F(Q) + \delta, i \in [1, |B^c|]\}. \end{aligned}$$

Binary search over B^c can efficiently calculate i^c_l and i^c_r with the $O(\log|B^c|)$ time complexity.

When SITA enters node v, it checks if fingerprints at v have similar properties. The ids of fingerprints with similar properties must be in I^c_q. If $I^c_v \cap I^c_q = \emptyset$ holds, which is easily verified without looking at any fingerprints, the properties of all fingerprints at v are proven to be different from Q and the subtree rooted at v is safely ignored.

The second pruning scheme is performed at each node v by using *summary fingerprint* y_v of which Jaccard similarity to Q is the upperbound of the fingerprints for v. When T^c is built, y^c_v is computed. Let $x^c_i \in \{0,1\}^M$ and $W_{B^c[i]}$ be a fingerprint of length M and its corresponding set representation, respectively. Then, y^c_v for v is defined as $y^c_v = \bigvee_{i \in I^c_v} x^c_i$, indicating that any word in x^c_i is in y^c_v (see the right of Figure 1 that represents y^2_v in the first two-level nodes of T^2).

Assume that SITA currently checks fingerprints for B^c (i.e., $|W| = c$ holds for any fingerprint). The following equivalent constraint is derived from constraint (1):

$$\begin{aligned} & J(W,Q) = \frac{|W \cap Q|}{|W \cup Q|} = \frac{|W \cap Q|}{|W| + |Q| - |W \cap Q|} \geq \epsilon \\ \iff & |W \cap Q| \geq \frac{\epsilon}{1+\epsilon}(|W| + |Q|) = \frac{\epsilon}{1+\epsilon}(c + |Q|). \end{aligned}$$

Let Y be the set representation of y^c_v and $y^c_v[i]$ be the i-th bit of y^c_v. $|W_{B^c[i]} \cap Q| \leq |Y \cap Q|$ holds for any $i \in I^c_v$. Therefore, if $|Y \cap Q| = \sum_{j \in Q} y^c_v[j] < \frac{\epsilon}{1+\epsilon}(c + |Q|)$ holds at node v, any fingerprint x_i at v breaks constraint (1). Thus, SITA can safely prune the subtree rooted at v, which results in improving the efficiency of similarity search.

Algorithm 1 shows the pseudo-code of SITA.

4.2.3 *Time and Space Complexities*

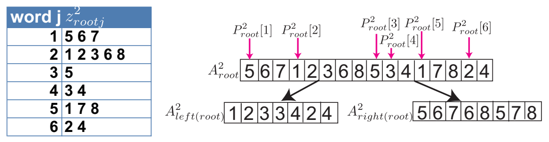
SITA's efficiency comes from the fact that the useless parts of T^c are pruned out. Let τ and m be the numbers of traversed nodes and fingerprints in the query, respectively. The time complexity of Algorithm 1 is $O(\tau m)$. SITA is particularly efficient with large ϵ and small δ, because of achieving the tighter bound regarding Jaccard similarity and narrower I^c_q. However, since DivideSkip and MT also improve the performance with large ϵ, the empirical performance differences among SITA, DivideSkip and MT tend to be smaller than that with large ϵ (see Section 5).

A crucial drawback is that T^c requires $M \sum^M_{c=1} |B^c| \log |B^c|$ bits due to the requirement of M bits for each y^c_v. Since M is large in practice ($M \geq 5000$ in our experiments), SITA requires a much larger amount of memory than modern PCs. The next two subsections describe approaches to reduce the memory usage while preserving the same time complexity.

Algorithm 1 Algorithm for finding similar fingerprints with similar properties to query Q

```
1: function SEARCH($Q$)
2:     for c satisfying $\epsilon|Q| \leq c \leq |Q|/\epsilon$ do
3:         $k \leftarrow \frac{\epsilon}{1+\epsilon}(c+|Q|)$, $I^c_{root} \leftarrow [1, |B^c|]$, $v \leftarrow \phi$
4:         $I^c_q \leftarrow [i^c_l, i^c_r]$ where $i^c_l \leftarrow \min\{i; F(Q) - \delta \leq F(W_{B^c[i]}), i \in I^c_{root}\}$ and $i^c_r \leftarrow \max\{i; F(W_{B^c[i]}) \leq F(Q) + \delta, i \in I^c_{root}\}$
5:         Recursion($v$,$I^c_{root}$,$Q$,$c$,$I^c_q$)
6:     end for
7: end function
8: function RECURSION($v$,$I^c_v$,$Q$,$c$,$I^c_q$)
9:     if $I^c_v \cap I^c_q = \phi$ then
10:        return
11:    else if $\sum_{j \in Q} y_v[j] < k$ then
12:        return
13:    end if
14:    if $|v| = \lceil \log |B_c| \rceil$ then                ▷ Leaf Node
15:        Output the index $id[int(v) + 1]$
16:    end if
17:    Recursion($v +' 0'$,$[s, \lfloor (s+e)/2 \rfloor]$,$Q$,$c$,$I^c_q$)    ▷ To left child. Note $I^c_v = [s, e]$.
18:    Recursion($v +' 1'$,$[\lfloor (s+e)/2 \rfloor + 1, e]$,$Q$,$c$,$I^c_q$)    ▷ To right child
19: end function
```

4.3 Representation of Summary Fingerprints with Inverted Index


$P^2_{root}[1]$
$P^2_{root}[2]$
$P^2_{root}[3]$
$P^2_{root}[4]$
$P^2_{root}[5]$
$P^2_{root}[6]$
A^2_{root} 5 6 7 1 2 3 6 8 5 3 4 1 7 8 2 4
$A^2_{left(root)}$ 1 2 3 3 4 2 4
$A^2_{right(root)}$ 5 6 7 6 8 5 7 8

word j	z^2_{rootj}
1	5 6 7
2	1 2 3 6 8
3	5
4	3 4
5	1 7 8
6	2 4

Figure 2: Example of A^c_v for T^c in Figure 1

To reduce the large memory requirement for preserving summary fingerprints, we use the inverted index to calculate the aforementioned upperbound on the similarity of fingerprints. The inverted index itself does not always reduce the memory requirement. However, with the help of the ideas behind the wavelet tree [12, 7], SITA compactly maintains only the minimum amount of information called the *rank dictionary* [25] decreasing the memory usage by a large margin (see the next subsection for details).

Our inverted index is an associative array that maps a word w to the set of all fingerprint ids that contain w. We build an inverted index for each node v with I^c_v in T^c. Let $z^c_{vj} = \{i; x^c_{B^c[i]}[j] = 1, i \in I^c_v\}$ for word j (i.e., all fingerprint ids containing j within I^c_v). The inverted index A^c_v for node v in T^c is a one-dimensional array that concatenates all z^c_{vj} in ascending order of j and is defined as $A^c_v = z^c_{v1} \cup z^c_{v2} \cup \cdots \cup z^c_{vM}$. Figure 2 shows the first two levels of the inverted indexes A^c_{root}, $A^c_{left(root)}$ and $A^c_{right(root)}$ in T^c in Figure 1.

Let $P^c_v[j]$ be the head of z^c_{vj} in A^c_v. If no fingerprint at node v has word j, $P^c_v[j+1] = P^c_v[j]$ holds. Assume that a query $Q = (q_1, \cdots, q_m)$ is given, $s_{vj} = P^c_v[q_j]$ and $t_{vj} = P^c_v[q_j+1] - 1$. Then, if $s_{vj} \leq t_{vj}$ holds, there is at least one fingerprint that contains q_j because of A^c_v's property. Otherwise, no fingerprint at v contains word q_j. Thus, $|Y \cap Q|$ in Subsection 4.2.2 is equal to $\sum_{j=1}^{m} \mathcal{I}[s_{vj} \leq t_{vj}]$ where $\mathcal{I}[cond]$ is the indicator function that returns one if *cond* is true and zero otherwise, and $|Y \cap Q|$ is safely used as a criterion to perform pruning. For example, in Figure 2, for $Q = (1, 2, 4)$ and $A^2_{left(root)}$, $|Y \cap Q| = \mathcal{I}[1 \leq 0] + \mathcal{I}[1 \leq 3] + \mathcal{I}[4 \leq 5] = 2$.

One advantage is that computing $|Y \cap Q|$ requires only s_{vj} and t_{vj}. Therefore, after A^c_v is built, by saving a pair of $(P^c_v[j], P^c_v[j+1] - 1)$ for each word j at each node v in T^c, A^c_v can be removed from the memory. However, since this approach still requires larger spaces than the available memory of modern PCs, the next subsection presents techniques to further reduce the memory usage.

4.4 Reduction of Memory Requirement with Succinct Data Structures

We describe a memory-efficient approach to combine the inverted index with the rank dictionary. After introducing the notion of the rank dictionary, we develop bit array representations for each node in T^c and techniques to check the similarity between summary fingerprints and a query.

4.4.1 Rank Dictionary

Rank dictionary is a data structure for a bit array B of length n [25] and supports the rank query $rank_c(B, i)$ that returns the number of occurrences of $c \in \{0, 1\}$ in $B[1, i]$. Although naive approaches require the $O(n)$ time to compute a rank, several data structures with only the $n + o(n)$ bit storage are presented to achieve the $O(1)$ time [24, 32]. We employ the *verbatim* rank dictionary [24] to calculate $\mathcal{I}[s_{vj} \leq t_{vj}]$ in Subsection 4.3 with the $O(1)$ time and thus to preserve the same time complexity as the aforementioned similarity search algorithm. We discuss only how to compute the $rank_1$ query, because $rank_0(B, i) = i + 1 - rank_1(B, i)$.

First, in verbatim the bit array is divided to large blocks of length $l = \log^2 n$ (see Figure 3). The ranks of the boundaries of large blocks are then recorded explicitly into an array $R_L[0, \ldots, n/l]$ using $O(n/\log^2 n \cdot \log n) = O(n/\log n)$ bits. Next, each large block is further divided into small blocks of length $s = \log n/2$. For all boundaries of small blocks, their ranks relative to the large block are recorded into $R_S[0, \ldots, n/s]$. In addition, the popcount data structure is used and allows to count the number of ones in $S[i, i+j]$ in constant time using a precomputed table of size $O(\sqrt{n} \log^2 n)$ [11]. Let $popcount(i, j)$ be the number of ones in $S[i, i+j]$. Then $rank_1(S, i)$ is computed as:

$$\begin{aligned} rank_1(S, i) = \ & R_L[\lfloor i/l \rfloor] + R_S[\lfloor i/s \rfloor] + \\ & popcount(s\lfloor i/s \rfloor, i \bmod s). \end{aligned}$$

Space complexity of auxiliary data structures for R_L, R_S, *popcount* is sublinear and negligible in the limit $n \rightarrow \infty$. Although popcount alone can construct a rank dictionary, the hierarchical construction of verbatim is much more succinct.

4.4.2 Jaccard Similarity Computation for Summary Fingerprints by Using Rank Dictionaries

Our idea is based on the wavelet tree (WT) [12, 7], a succinct data structure for efficiently accessing arrays with a rank dictionary. However, SITA maintains only the memory-efficient rank dictionary without preserving A^c_v in memory. Thus, SITA can efficiently compute $|Y \cap Q|$ in Subsection 4.3 with a small amount of memory.

A WT contains a collection of bit arrays to update the intervals in the constant time. Let b^c_v be a bit array for node v in T^c with size of $|A^c_v|$ and with interval I^c_v, $left(v)$

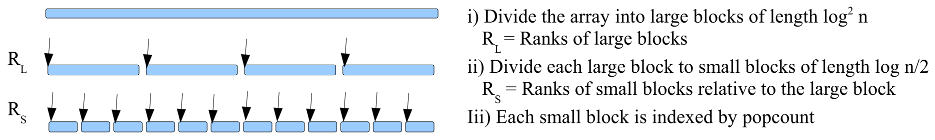

i) Divide the array into large blocks of length log² n
R_L = Ranks of large blocks
ii) Divide each large block to small blocks of length log n/2
R_S = Ranks of small blocks relative to the large block
Iii) Each small block is indexed by popcount
R_L
R_S

Figure 3: Construction of a rank dictionary from a bit array

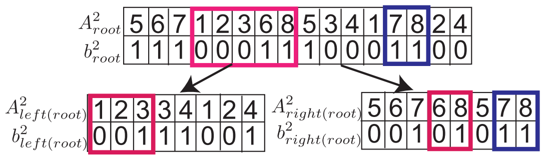

A^2_{root} 5 6 7 1 2 3 6 8 5 3 4 1 7 8 2 4
b^2_{root} 1 1 1 0 0 0 1 1 1 0 0 0 1 1 0 0
$A^2_{left(root)}$ 1 2 3 3 4 1 2 4
$b^2_{left(root)}$ 0 0 1 1 1 0 0 1
$A^2_{right(root)}$ 5 6 7 6 8 5 7 8
$b^2_{right(root)}$ 0 0 1 0 1 0 1 1

Figure 4: First two levels of wavelet tree for Figure 2

and $right(v)$ be v's left and right children, respectively. We use the verbatim rank dictionary to represent b^v_c.

$A^c_{left(v)}$ and $A^c_{right(v)}$ are constructed by moving each element of A^c_v to either $left(v)$ or $right(v)$ with keeping the order of elements in A^c_v by considering the fact that A^c_v is a concatenated inverted index of which fingerprint ids are restricted with I^c_v, $I^c_v = I^c_{left(v)} \cup I^c_{right(v)}$ and $I^c_{left(v)} \cap I^c_{right(v)} = \emptyset$. Bit $b^c_v[k]$ indicates that $A^c_v[k]$ should be moved to whether $left(v)$ or $right(v)$. If $b^c_v[k]$ is 0, $A^c_{left(v)}$ contains $A^c_v[k]$. If $b^c_v[k] = 1$, $A^c_{right(v)}$ inherits $A^c_v[k]$. Formally, $b^c_v[k]$ with $I^c_v = [i, i']$ is defined as:

$$b^c_v[k] = \begin{cases} 1 & \text{if } A^c_v[k] > \lfloor (i+i')/2 \rfloor \\ 0 & \text{if } A^c_v[k] \leq \lfloor (i+i')/2 \rfloor \end{cases} .$$

Figure 4 shows an example of the first two levels of the WT. For example, because $A^2_{root}[7] = 6 \in I^2_{right(root)} = [5, 8]$ and it is the fourth element of A^2_{root} that must go to the right child of the root, $b^2_{root}[7] = 1$ and $A^2_{right(root)}[4] = A^2_{root}[7] = 6$ hold.

Let $Q = (q_1, \cdots, q_m)$ be a query and s_{vj} and t_{vj} be locations of A^c_v which are used to compute $|Y \cap Q| = \sum_{j=1}^{m} \mathcal{I}[s_{vj} \leq t_{vj}]$ at node v as in Subsection 4.3. SITA preserves P^c_{root} in memory so that it can set $s_{vj} = P^c_v[q_j]$ and $t_{vj} = P^c_v[q_j + 1] - 1$ if v is the root. Then, by using s_{vj}, t_{vj}, the rank operations for b^c_v, locations $s_{left(v)j}$, $t_{left(v)j}$, $s_{right(v)j}$, $t_{right(v)j}$ are obtained with the $O(1)$ time complexity as follows:

$$\begin{aligned} s_{left(v),j} &= \text{rank}_0(b^c_v, s_{vj} - 1) + 1, \\ t_{left(v),j} &= \text{rank}_0(b^c_v, t_{vj}), \\ s_{right(v),j} &= \text{rank}_1(b^c_v, s_{vj} - 1) + 1, \\ t_{right(v),j} &= \text{rank}_1(b^c_v, t_{vj}). \end{aligned}$$

Note that SITA need to preserve only b^c_v to compute $|Y \cap Q|$ and does not require P^c_v if v is not the root.

As in the WT, SITA requires $(1+\alpha) \sum_{c=1}^{M} N^c \log |B^c|$ bits for managing b^c_v for all nodes v, where α is the overhead of the rank dictionary (α is about 0.62 in our case), and N^c is the total number of words in B^c, i.e., $N^c = \sum_{i \in B^c} |W_i|$. This is a better representation than managing the summary fingerprints in memory, which requires $M \sum_{c=1}^{M} |B^c| \log |B^c|$ bits where $M \geq 5000$ in our experiments. Although P^c_{root} requires $M \sum_{c=1}^{M} \log N^c$ bits, it is not an obstacle even for a larger database. The storage for P^c_{root} grows only logarithmically in the number of fingerprints.

5. EXPERIMENTS

5.1 Setups

We evaluated the performance of binary search (BIN), MT, b-bit minhash, DivideSkip, and SITA on one core of a quad-core AMD Opteron Processor 8393 SE (3.1GHz) machine with 512 GB memory. BIN is a strawman baseline that first excludes fingerprints breaking constraint (2) by sorting and binary search, and then calculates Jaccard similarity for each unfiltered fingerprint to find P_{NC}. As in Section 3, MT is the best possible algorithm in chemoinformatics, and DivideSkip and b-bit minhash are state-of-the-art methods in data mining. We implemented them in C++ and SITA used Vigna's rank dictionary implementation called rank9 [32].

We used 214, 636, 657 compounds-protein pairs in the STITCH database [17]. We represented each compound-protein pair by a fingerprint with the dimension of 5, 014, constructed by concatenating the compound substructure fingerprint (881 substructures) in PubChem [5] and the protein domain fingerprint (4, 133 domains) in PFAM [8, 10]. We used seven representative properties with a variety of mean values and standard deviations (see Table 2) to elucidate the behavior of each algorithm. In particular, SITA tends to perform better for a property of a large standard deviation, because of the higher possibility of pruning search spaces. We randomly sampled 2, 000 compound-protein pairs as queries.

It took 253 minutes to precompute SITA's necessary data structures for all compound-protein pairs. Once these data structures are computed, the query phase using SITA does not need to recompute them. The data structure construction time indicates that the overhead incurred in the construction phase is not a serious issue to use SITA in practice.

b-bit minhash is an approximate algorithm that has three parameters of string length ℓ, hashing value b and search width k. We tried all combinations of $\ell = \{5, 10, 50\}$, $b = \{2, 4, 8, 16\}$ and $k = \{10^2, ..., 10^7\}$, and chose the parameter that found solutions most quickly with at most the 5% false negative rate. As in [23], MT's single parameter, the maximum number of fingerprints associated with a leaf, was set to 10. DivideSkip's parameter μ used to choose the length of long lists was set to the best one by experimenting the cases of $\mu = \{10^{-3}, 10^{-2}, ..., 10^3\}$.

5.2 Results

Tables 3 and 4 show the results of each algorithm with various properties, two different property thresholds ($\delta = 0.5$ and 5) and $\epsilon = 0.6$, where the search time is the average over 2,000 queries with its standard deviation, $|P_1|$ is the number of candidate fingerprints chosen by database partitioning in Subsection 4.1, $|P_1 \cap [i_l, i_r]|$ is the number of candidates chosen by database partitioning plus sorting and binary search (i.e., the total size of I^c_q for unfiltered blocks in Subsection 4.2.2), #Rank is the number of rank operations, and $|P_{NC}|$ is the number of solutions.

Table 2: Means and standard deviations (sdev.) of properties

	ALOGP (1)	ALOGP (2)	XlogP	Topological PorlarSufaceArea (TPSA)	Autocorrelation Polarizability (AP) (1)	AP (2)	AP (3)
mean	15.21	98.04	2.07	102.79	1707.70	1989.37	2780.04
sdev.	106.95	85.54	3.64	109.48	1792.31	2105.02	3001.76

Table 3: Performance summary (average search time, 214,636,657 fingerprints, $\epsilon = 0.6$ and $\delta = 0.5$)

Method	ALOGP (1)	ALOGP (2)	XlogP	TPSA	AP (1)	AP (2)	AP (3)
SITA	4.89 ± 7.89	0.35 ± 0.29	2.22 ± 1.99	0.83 ± 0.78	0.26 ± 0.17	0.24 ± 0.15	0.22 ± 0.12
BIN	1257.43 ± 988.90	367.935 ± 88.62	421.95 ± 94.85	344.16 ± 95.37	385.09 ± 120.60	582.87 ± 178.40	568.744 ± 173.57
MT	746.96 ± 699.05	746.96 ± 699.05	746.96 ± 699.05	746.96 ± 699.05	746.96 ± 699.05	746.96 ± 699.05	746.96 ± 699.05
DivideSkip	$48,235.50 \pm 41,847.62$	$48,235.50 \pm 41,847.62$	$48,235.50 \pm 41,847.62$	$48,235.50 \pm 41,847.62$	$48,235.50 \pm 41,847.62$	$48,235.50 \pm 41,847.62$	$48,235.50 \pm 41,847.62$
b-bit minhash	$9,789.75 \pm 4,946.18$	$9,789.75 \pm 4,946.18$	$9,789.75 \pm 4,946.18$	$9,789.75 \pm 4,946.18$	$9,789.75 \pm 4,946.18$	$9,789.75 \pm 4,946.18$	$9,789.75 \pm 4,946.18$
$\|P_1\|$	96,293,752	96,293,752	96,293,752	96,293,752	96,293,752	96,293,752	96,293,752
$\|P_1 \cap [i_l, i_r]\|$	12,488,453	716,533	1,051,232	199,235	43,406	45,185	45,458
#Rank	200,662,960	11,930,629	87,537,624	5,258,371	2,163,349	2,062,009	1,989,990
$\|P_{NC}\|$	754,446	50,163	71,155	21,894	8,294	7,922	7,587

Table 4: Performance summary (average search time, 214,636,657 fingerprints, $\epsilon = 0.6$ and $\delta = 5$)

Method	ALOGP (1)	ALOGP (2)	XlogP	TPSA	AP (1)	AP (2)	AP (3)
SITA	17.76 ± 21.25	2.51 ± 2.54	10.39 ± 9.78	0.83 ± 0.78	0.26 ± 0.17	0.24 ± 0.15	0.22 ± 0.12
BIN	$5,400.28 \pm 2907.62$	895.44 ± 237.626	714.62 ± 161.27	608.02 ± 125.89	472.48 ± 130.99	578.66 ± 164.77	634.28 ± 191.07
MT	746.96 ± 699.05	746.96 ± 699.05	746.96 ± 699.05	746.96 ± 699.05	746.96 ± 699.05	746.96 ± 699.05	746.96 ± 699.05
DivideSkip	$48,235.50 \pm 41,847.62$	$48,235.50 \pm 41,847.62$	$48,235.50 \pm 41,847.62$	$48,235.50 \pm 41,847.62$	$48,235.50 \pm 41,847.62$	$48,235.50 \pm 41,847.62$	$48,235.50 \pm 41,847.62$
b-bit minhash	$9,789.75 \pm 4,946.18$	$9,789.75 \pm 4,946.18$	$9,789.75 \pm 4,946.18$	$9,789.75 \pm 4,946.18$	$9,789.75 \pm 4,946.18$	$9,789.75 \pm 4,946.18$	$9,789.75 \pm 4,946.18$
$\|P_1\|$	96,293,752	96,293,752	96,293,752	96,293,752	96,293,752	96,293,752	96,293,752
$\|P_1 \cap [i_l, i_r]\|$	51,248,152	7,065,457	5,102,828	1,891,708	372,228	323,858	245,625
#Rank	746,507,456	99,194,552	420,780,736	30,032,964	6,626,788	5,721,429	4,536,575
$\|P_{NC}\|$	2,805,425	420,160	308,234	124,494	27,587	23,715	18,837

Since MT, DivideSkip and *b*-bit minhash filter out useless candidates by only considering constraint (1), their performance remained unchanged with different δ and properties.

Although DivideSkip's efficiency had been verified in previous work in [19], its performance was evaluated with a much smaller database consisting of 2 million fingerprints. In contrast, when a large-scale dataset consisting of 214 million fingerprints was used with including an additional constraint on properties, DivideSkip did not perform well. It took $48,235$ seconds for DivideSkip to find P_{NC} on average.

Since BIN's performance depends on how many fingerprints are filtered out with δ, its performance was improved with smaller δ due to a smaller value of $|P_1 \cap [i_l, i_r]|$, which often resulted in outperforming MT. However, BIN still needed to perform many one-by-one checks for constraint (2) due to large differences between $|P_1 \cap [i_l, i_r]|$ and $|P_{NC}|$.

Although MT is the most efficient algorithm in the previous chemoinformatics literature and we observed that MT was faster than DivideSkip and *b*-bit minhash which are state-of-the-art in data mining, SITA performed much faster than MT. In case of considering properties, it was hard for MT to reduce the candidate fingerprints, while SITA effectively ignored fingerprints with dissimilar properties. In particular, SITA was 150-3390 times faster than MT if $\delta = 0.5$. With larger δ, the performance difference became smaller between SITA and MT, due to an excessive number of SITA's rank operations (see $\delta = 5$ and ALOGP(1) in Table 4). However, SITA still outperformed MT even in this case.

Figure 5 shows the search time of each algorithm for $\epsilon = 0.6$ and 0.8 with $\delta = 0.5$, three properties of various standard deviations and various database sizes. SITA consistently outperformed the second best algorithm by 2-3 orders of magnitude. Although the performance difference became smaller with different δ and other properties, we still observed the superiority of SITA to other approaches.

Figure 6 shows the average search time for various ϵ with $\delta = 0.5$ and TPSA. Again, SITA significantly outperformed the others, despite smaller performance differences with larger ϵ that increases the possibility of filtering out candidates for the others except BIN. BIN's performance remained unchanged because it performs pruning only with δ.

Figure 7 depicts the memory usage for each method. MT used the largest amount of memory and consumed 167 GB to preserve 214 million compound-protein pairs. Although a space-efficient version of MT was presented in [30], we verified that the search time of this approach drastically increased without any significant reduction of memory usage. In contrast, BIN kept fingerprints of compound-protein pairs by using only 16 bits per word in a fingerprint, which resulted in using only 60 GB memory to store all compound-protein pairs. *b*-bit minhash required a similar amount to BIN at the price of giving up correctness. With the help of succinct data structures behind the wavelet tree, our SITA consumed 85 GB memory, which was similar to DivideSkip, as estimated from the theoretical analysis (see Table 1).

Figure 8 shows an example of querying compound-protein pairs with XlogP. PubChem and Uniprot IDs are used to specify compounds and proteins, respectively, followed by the chemical structures of compounds, protein names and PFAM domains. Among proteins that could bind to compounds similar to the query compound, SITA found many ion-transport proteins that play important biological roles such as neurotransmitter. Considering the fast average search time (2.2 seconds) and small memory usage (85GB), the result demonstrated the feasibility to find similar compound-protein pairs in terms of both compounds and proteins, which would be beneficial to analyze compound-protein interactions by checking known compound-protein interactions that are similar to the compound-protein pair of interest.

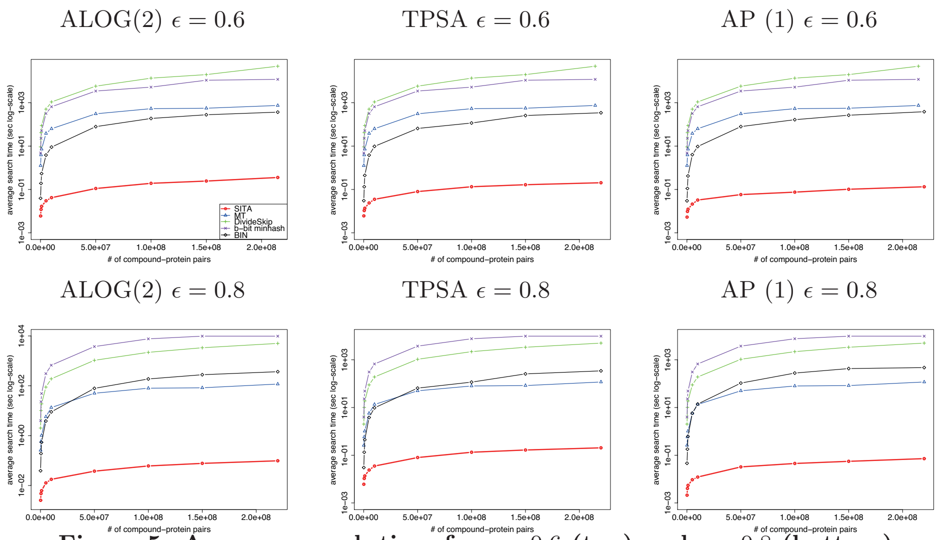


Figure 5: Average search time for $\epsilon = 0.6$ (top) and $\epsilon = 0.8$ (bottom)

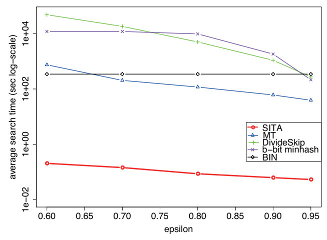


Figure 6: Search time for various ϵ

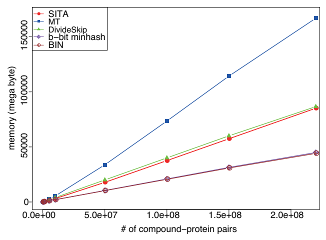


Figure 7: Memory usage for each method

As another application, we tested SITA, MT and BIN on their abilities to search for substrate-product pairs (compound-compound pairs converted to each other by enzymatic reactions). We calculated the average search time of $2,000$ queries per algorithm on a large-scale database consisting of $243,438,006$ compound-compound pairs, where each pair is represented by a fingerprint with the dimension of 1,758 based on differential chemical substructures [15]. In addition, we used the absolute difference of the molecular weights of each compound-compound pair as a property. With $\epsilon = 0.8$ and $\delta = 0.5$, the average search time of SITA, MT and BIN was 0.12, 461.38 and 651.15 seconds, respectively. SITA used 89 GB memory, while MT and BIN required 114 and 86 GB memory, respectively. This clearly indicates a superiority of SITA on a different dataset. Detailed results are supplementally available at `http://www.bioreg.kyushu-u.ac.jp/labo/systemcohort/kdd2013/supplement.pdf`.

6. CONCLUSIONS AND FUTURE WORK

We have presented SITA, a novel, time-and-memory-efficient algorithm to find all compound-protein pairs that are similar to a query in terms of Jaccard similarity and properties from a large-scale database. Experimental results demonstrated that SITA outperformed other state-of-the-art algorithms while reducing the memory requirement to 85 GB.

SITA can currently deal with Jaccard similarly and only one property at a time. One extension is therefore to find similarity with many properties that generate additional constraints. This would be more beneficial for users to find potential functional compound-protein pairs.

7. REFERENCES

[1] Z. Aung and S. Ng. An indexing scheme for fast and accurate chemical fingerprint database searching. In *Scientific and Statistical Database Management*, pages 288–305. Springer, 2010.

[2] P. Baldi and D. Hirschberg. An intersection inequality sharper than the tanimoto triangle inequality for efficiently searching large databases. *Journal of Chemical Information and Modeling*, 49:1866–1870, 2009.

[3] P. Baldi, D. Hirschberg, and R. Nasr. Speeding up chemical database searches using a proximity filter based on the logical exclusive-OR. *Journal of Chemical Information and Modeling*, 48:1367–1378, 2008.

[4] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise independent permutations. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, pages 327–336, 1998.

[5] B. Chen, D. Wild, and R. Guha. PubChem as a source of polypharmacology. *Journal of Chemical Information and Modeling*, 49:2044–2055, 2009.

[6] J. Chen, S. Swamidass, Y. Dou, J. Bruand, and P. Baldi. ChemDB: A public database of small molecules and related chemoinformatics resources. *Bioinformatics*, 21:4133–4139, 2005.

[7] F. Claude and G. Navarro. Practical rank/select queries over arbitrary sequences. In *Proceedings of the 15th edition of the International Symposium on String*

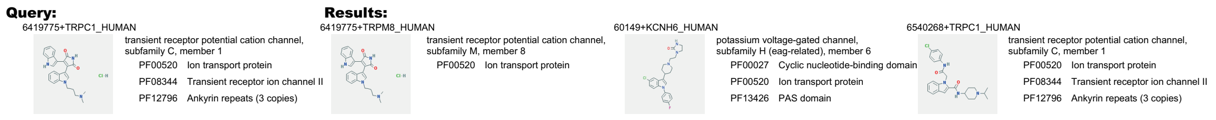

Query:
6419775+TRPC1_HUMAN
Cl·H
transient receptor potential cation channel, subfamily C, member 1
PF00520 Ion transport protein
PF08344 Transient receptor ion channel II
PF12796 Ankyrin repeats (3 copies)
Results:
6419775+TRPM8_HUMAN
Cl·H
transient receptor potential cation channel, subfamily M, member 8
PF00520 Ion transport protein
60149+KCNH6_HUMAN
potassium voltage-gated channel, subfamily H (eag-related), member 6
PF00027 Cyclic nucleotide-binding domain
PF00520 Ion transport protein
PF13426 PAS domain
6540268+TRPC1_HUMAN
transient receptor potential cation channel, subfamily C, member 1
PF00520 Ion transport protein
PF08344 Transient receptor ion channel II
PF12796 Ankyrin repeats (3 copies)

Figure 8: Example of querying compound-protein pairs

Processing and Information Retrieval, pages 176–187, 2008.

[8] T. U. Consortium. The universal protein resource (uniprot) in 2010. *Nucleic Acids Research*, 38:D142–D148, 2010.

[9] C. Dobson. Chemical space and biology. *Nature*, 432(7019):824–828, 2004.

[10] R. Finn, J. Tate, J. Mistry, P. Coggill, J. Sammut, H. Hotz, G. Ceric, K. Forslund, S. Eddy, E. Sonnhammer, and A. Bateman. The Pfam protein families database. *Nucleic Acids Research*, 36:D281–D288, 2008.

[11] R. Gonzalez, S. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *Proceedings of the 4th International Workshop on Efficient and Experimental Algorithms*, pages 27–28, 2005.

[12] R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 636–645, 2003.

[13] S. Gunther, M. Kuhn, M. Dunkel, M. Campillos, C. Senger, E. Petsalaki, J. Ahmed, E. Urdiales, A. Gewiess, L. Jensen, R. Schneider, R. Skoblo, R. Russell, P. Bourne, P. Bork, and R. Preissner. SuperTarget and Matador: Resources for exploring drug-target relationships. *Nucleic Acids Research*, 36:D919–D922, 2008.

[14] M. Kanehisa, M. Araki, S. Goto, M. Hattori, M. Hirakawa, M. Itoh, T. Katayama, S. Kawashima, S. Okuda, T. Tokimatsu, and Y. Yamanishi. KEGG for linking genomes to life and the environment. *Nucleic Acids Research*, 36:D480–D485, 2008.

[15] M. Kotera, Y. Tabei, Y. Yamanishi, T. Tokimatsu, and S. Goto. Supervised de novo reconstruction of metabolic pathways from metabolome-scale compound sets. In *Proceedings of ISMB/ECCB*, 2013. To appear.

[16] T. G. Kristensen, J. Nielsen, and C. N. S. Pedersen. A tree-based method for the rapid screening of chemical fingerprints. *Algorithms for Molecular Biology*, 5, 2010.

[17] M. Kuhn, D. Szklarczyk, A. Franceschini, M. Campillos, C. von Mering, L. Jensen, A. Beyer, and P. Bork. STITCH 2: An interaction network database for small molecules and proteins. *Nucleic Acids Research*, 38(suppl 1):D552–D556, 2010.

[18] A. Leach and V. Gillet. *An introduction to chemoinformatics*. Kluwer Academic Publishers, The Netherlands, Revised Edition, 2007.

[19] C. Li, J. Lu, and Y. Lu. Efficient merging and filtering algorithms for approximate string searches. In *Proceedings of the IEEE 24th International Conference on Data Engineering*, pages 257–266, 2008.

[20] P. Li and A. Christian König. Theory and applications of b-bit minwise hashing. *Communications of the ACM*, 54(8):101, 2011.

[21] P. Li and C. König. b-bit minwise hashing. In *Proceedings of the 19th International Conference on World Wide Web*, pages 671–680. ACM, 2010.

[22] R. Nasr, D. Hirschberg, and P. Baldi. Hashing algorithms and data structures for rapid searches of fingerprint vectors. *Journal of Chemical Information and Modeling*, 50:1358–68, 2010.

[23] R. Nasr, T. Kristensen, and P. Baldi. Tree and hashing data structures to speed up chemical searches: Analysis and experiments. *Molecular Informatics*, 30:791–800, 2011.

[24] D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proceedings of the Workshop on Algorithm Engineering and Experiments*, 2007.

[25] R. Raman, V. Raman, and S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 232–242, 2002.

[26] L. Ribeiro and T. Härder. Efficient set similarity joins using min-prefixes. In *Advances in Databases and Information Systems*, pages 88–102. Springer, 2009.

[27] C. Steinbeck, C. Hoppe, S. Kuhn, M. Floris, R. Guha, and E. L. Willighagen. Recent developments of the chemistry development kit (CDK) - an open-source Java library for chemo- and bioinformatics. *Current Pharmaceutical Design*, 12:2111–2120, 2006.

[28] B. Stockwell. Chemical genetics: Ligand-based discovery of gene function. *Nature Reviews Genetics*, 1:116–125, 2000.

[29] S. Swamidass and P. Baldi. Bounds and algorithms for exact searches of chemical fingerprints in linear and sublinear time. *Journal of Chemical Information and Modeling*, 47:302–317, 2007.

[30] Y. Tabei. Succinct multibit tree: Compact representation of multibit trees by using succinct data structures in chemical fingerprint searches. In *Proceedings of the 12th Workshop on Algorithms in Bioinformatics*, pages 201–213, 2012.

[31] R. Todeschini and V. Consonni. *Handbook of Molecular Descriptors*. Wiley-VCH, 2002.

[32] S. Vigna. Broadword implementation of rank/select queries. In *Proceedings of the 7th International Conference on Experimental Algorithms*, pages 154–168. Springer-Verlag, 2008.

[33] D. Wishart, C. Knox, A. Guo, D. Cheng, S. Shrivastava, D. Tzur, B. Gautam, and M. Hassanali. DrugBank: A knowledgebase for drugs, drug actions and drug targets. *Nucleic Acids Research*, 36:D901–D906, 2008.

[34] C. Xiao, W. Wang, X. Lin, J. X. Yu, and G. Wang. Efficient similarity joins for near duplicate detection. *ACM Transactions on Database Systems*, 36:15, 2011.