Noname manuscript No.
(will be inserted by the editor)

DRESS: Dimensionality Reduction for Efficient Sequence Search

Alexios Kotsifakos · Alexandra Stefan · Vassilis Athitsos · Gautam Das · Panagiotis Papapetrou

the date of receipt and acceptance should be inserted later

Abstract Similarity search in large sequence databases is a problem ubiquitous in a wide range of application domains, including searching biological sequences. In this paper we focus on protein and DNA data, and we propose a novel approximate method method for speeding up range queries under the edit distance. Our method works in a filter-and-refine manner, and its key novelty is a query-sensitive mapping that transforms the original string space to a new string space of reduced dimensionality. Specifically, it first identifies the t most frequent codewords in the query, and then uses these codewords to convert both the query and the database to a more compact representation. This is achieved by replacing every occurrence of each codeword with a new letter and by removing the remaining parts of the strings. Using this new representation, our method identifies a set of candidate matches that are likely to satisfy the range query, and finally refines these candidates in the original space. The main advantage of our method, compared to alternative methods for whole sequence matching under the edit distance, is that it does not require any training to create the mapping, and it can handle large query lengths with negligible losses in accuracy. Our experimental evaluation demonstrates that, for higher range values and large query sizes, our method produces significantly lower costs and runtimes compared to two state-of-the-art competitor methods.

Alexios Kotsifakos, Alexandra Stefan, Vassilis Athitsos, and Gautam Das
Department of Computer Science and Engineering
University of Texas at Arlington
E-mail: alexios.kotsifakos@mavs.uta.edu, astefan@uta.edu, athitsos@uta.edu, gdas@uta.edu

Panagiotis Papapetrou
Department of Computer and Systems Sciences
Stockholm University
E-mail: panagiotis@dsv.su.se

1 Introduction

Similarity search in large sequence databases is a frequently occuring problem. Despite the plethora of string matching methods currently available, there is still a high demand for new, robust, and scalable similarity search methods that can handle large queries and large similarity ranges. These methods could be highly beneficial to experts in different application domains, such as searching biological sequences, comparing large text corpora, and spell-checking.

In this paper, we study the problem of *whole sequence matching* in string databases. In other words, given a database of strings and a query, we want to efficiently identify those strings that are most similar to the query under the edit distance, which is the most commonly used measure for this problem. We particularly focus on the case where the query string is long, i.e., hundreds of characters or longer. More specifically, the problem at hand is to find, for a specific query string Q and a user-defined range r, all the database strings whose edit distance from Q is within this range. For convenience, we typically express r as a percentage of the query length $|Q|$, i.e., $r = \delta|Q|$. Thus, if the query length is 1000 and $\delta = 15\%$, we want to retrieve all database strings whose distance from Q is less than or equal to 150.

In many application domains, such as biology, being able to handle *large query lengths* and *large query-range values* can be highly beneficial.

- **Large query lengths:** In bioinformatics, a very common way of representing large genomes is by using *Expressed Sequence Tag (EST) databases*. Such databases contain portions of genes expressed as mature mRNA, where the length of each sequence is at least $500-800$ nucleotides long. In these databases, large scale searches need to be performed against other genomic databases to determine locations of genes (Jongeneel 2000). In practice, genes may vary in size from hundreds to thousands of nucleotides. Searches can also target whole chromosomes, where the goal is to find chromosome similarities across different organisms. Since chromosomes can be relatively large, such searches require algorithms that can handle large queries efficiently.
- **Large query-range values:** Due to high evolutionary divergence, the task of identifying distantly related gene or protein domains by sequence search techniques is not always that trivial. For example, during the mutation process, intermediate sequences may possess features of more than one protein and facilitate detection of remotely related proteins (Bhadra et al 2006). Hence, supporting large range queries, also known as *remote homology search* in bioinformatics, can be highly beneficial for searching proteins and genomes (Liu et al 2013; Bhadra et al 2006).

The importance of this problem is not limited to biology. For example, in text mining, given a collection of texts, an interesting problem is to find groups of texts with relatively high similarity. This can be useful in a plagiarism detection setting, where texts correspond to homework submitted by students, and highly similar texts could be considered suspicious for plagiarism.

Hence, it becomes evident that efficient similarity search in string databases where query length and range are both large is a crucial problem for various application

domains. At the same time, approximate methods could be used for substantially reducing the high computational cost, as long as they can achieve very high retrieval accuracy. Our focus in this paper is on biological sequences, and more specifically on searching protein and DNA databases under the edit distance.

Evaluating the similarity between two strings under the edit distance is linear to the product of the lengths of the two strings. This can be computationally expensive, especially as the size of the two strings grows. In addition, as far as our target application domain is concerned, it is known that the distribution of pairwise distances between long protein strings has low variance, which implies that such distances are very likely to have values relatively close to a certain mean distance. This property is a manifestation of the well-known curse of dimensionality and makes tree-like index structures (Traina et al 2000; Vieira et al 2004) ineffective for this problem. These structures are typically used for pruning the search space using the triangle inequality. Alternative methods define an index structure (e.g., q-gram inverted lists (Li et al 2008a)) that is used for identifying candidate matches efficiently, or use alternative string representations (e.g., alphabet reduction in Papapetrou et al (2009)). Nonetheless, the curse of dimensionality is present in both cases, with the former being unable to handle large query sizes and the latter requiring a tedious and computationally expensive training process in order to build the index.

In this paper, we propose an effective and efficient approximate method for similarity search in string databases. Our approach overcomes the curse of dimensionality and can handle large query sizes without requiring any substantial pre-processing. Specifically, our method defines an informative mapping that transforms the query and database strings into a new string space, where distances can be measured orders of magnitude faster. The construction of the mapping is query-sensitive and it is built online based on the appearances of a relatively small set of substrings within the query. These substrings are found based on how frequently they occur in the query, and the t most frequent ones of length l, called *codewords*, form the set of substrings. Using these codewords each string is converted to a new string, which basically shows the occurrences of the codewords in the original string. This representation captures a significant amount of information between strings, since the more similar two strings are, the more likely it is for them to have common substrings. Hence, the distance between the mapped strings is highly indicative of the distance between the original strings. Moreover, each original string is mapped into a much shorter string (assuming some appropriate choices in defining the mapping), making distance computation in the new string space much faster than in the original space.

Furthermore, we should ensure that the retrieval accuracy remains as high as possible, while speeding up similarity search. Towards this goal, we follow a filter-and-refine approach. In particular, for each query, the filter step has a fast, but less accurate, way of identifying a small set of candidate strings from the database. The mapping proposed in this paper is used to obtain an efficient filtering process. The refine step then performs the expensive similarity evaluations in the original space to determine which of the candidates match the desired search criteria.

In short, the main **contributions** of this paper are:

- We propose *DRESS* (shorthand for Dimensionality Reduction for Efficient Sequence Search), a novel filter-and-refine approximate method for speeding up similarity search under the edit distance. The key novelty of DRESS is that it does not require any training or computationally expensive pre-processing step compared to competitors, and can handle large query sizes and similarity ranges while maintaining competitive accuracy and retrieval runtime.
- We introduce an efficient way of representing strings in a new space, which is based on a set of codewords that are query-specific. In this new space, distance computation between strings is significantly faster than in the original space.
- We provide an extensive experimental evaluation of the proposed method against q-grams and a well-known reference-based string matching method on protein and DNA sequences. The experimental results on three large protein datasets and two large DNA datasets show that our method outperforms both competitors in terms of retrieval cost and runtime, while retaining most of the times an accuracy of over 99%.

2 Related Work

Several sequence matching algorithms have been proposed in the literature for sequence alignment. Depending on their objective, they can be divided into *global* alignment and *local* alignment methods. There is a fundamental difference between the two families of methods. Global alignment can be seen as a *full-sequence matching* approach for global optimization, where the objective is to identify the minimum number of changes that should be performed to one sequence so as to convert it to the other, by forcing the alignment to span the whole sequence. In contrast, the objective of local alignment is to identify local regions within the compared sequences that are highly similar to each other, while they could be globally divergent. Hence, the result of local alignment is to match several subsequences within the two sequences, while allowing for gaps in the alignment.

Starting with methods for local alignment, BLAST (Altschul et al 1990) is a well-known tool used to compute local alignments for long biological sequences. Since it is based on a heuristic search technique, BLAST is not guaranteed to always find the optimal local alignment. An improved version of BLAST, known as BLAST2 (Altschul et al 1997), achieves better accuracy by allowing a limited number of insertions and deletions during the alignment formation and improves search speed by imposing more stringent criteria when performing a local alignment. Other improvements of BLAST include MegaBLAST (Zhang et al 2000), MPBLAST (Korf and Gish 2000) and miBLAST (Kim et al 2005b). MegaBLAST is a greedy algorithm for detecting sequences that differ slightly as a result of sequencing. MPBLAST and miBLAST are different versions of BLAST used for parallel queries. BLAT (Kent 2002) builds an index of the database and then, given a query, it linearly scans the query searching for matches in the index. Apart from using an inverse index, BLAT differs from BLAST and BLAST2 in that it triggers extensions on any number of perfect hits whereas in BLAST extensions are triggered when one or two hits occur in proximity to each other. Several hash-based approaches (Kalafus et al 2004; Ning et al 2001)

have been developed for further speed up. A key limitation of all the above-mentioned variants of BLAST is that their accuracy and retrieval cost deteriorate as the query size increases. Also, as the volume of biological sequence databases increases, all the aforementioned exhaustive systems become prohibitively expensive. Another widely applied local alignment similarity measure is Smith-Waterman (Smith and Waterman 1981), on which OASIS is based (Meek et al 2003). The Shift-Or algorithm (Baeza-Yates and Gonnet 1992) and its extended variant (Wu and Manber 1992) are bit manipulation methods that also search for a query in a target sequence.

An embedding-based method for subsequence matching under local alignment is RBSA (Papapetrou et al 2009). The method employs an alphabet reduction technique, where, essentially, groups of letters are collapsed to one symbol (for example all the odd letters of the alphabet are replaced by 1 and all the even letters are replaced by 0). Employing the above technique in conjunction with reference-based embeddings, RBSA achieves very promising results. In particular, RBSA performs well for large queries and provides very accurate results, while it requires very expensive preprocessing and training. More importantly, it is designed for solving the problem of subsequence matching (under both edit distance and Smith-Waterman), which is orthogonal to the whole-sequence matching problem studied in this paper. In addition, the alphabet reduction technique used by RBSA does not reduce the length of the sequences, and as a result the time to compute the distance between two sequences with and without alphabet reduction is the same. In contrast, the method proposed here drastically reduces the length of the sequences, and thus it dramatically speeds up the computation of approximate distances.

Another family of methods, known as *short-read sequencing methods*, are widely used for aligning biological sequences. The key objective of short-read sequencing is to perform *near-exact subsequence matching*, which is fairly similar to local alignment, since a local segment of the target sequence is matched with the query. More specifically, in this setting, queries are substantially shorter than the target sequences, they are expected to match locally, with or without gaps in the alignment, and more importantly they assume the presence of a subsequence match in the target database sequence that is highly similar to the query. Methods of this family include SOAP (Li et al 2008c), Maq (Li et al 2008b), and Bowtie (Langmead et al 2009). The latter is essentially based on the properties of the Burrows-Wheeler Transforms (BWT) (Burrows and Wheeler 1994) to index the target sequence(s). The advantage of the index built, which can be seen as a suffix tree variant, is the small amount of memory used. WHAM (Li et al 2012) is another very recently proposed short-read sequencing method that employs the edit distance to align two sequences, while allowing for an arbitrary number of mismatches and gaps. It builds hash indexes on subsequences of the target sequence, and uses subsequences as seeds to find valid matching sequences. In addition, the fragments on which it is based on are non-overlapping subsequences, while it takes advantage of current memory sizes in order to provide fast alignments. The target queries are rather short, i.e., of size up to 100 base pairs. The PartEnum algorithm (Arasu et al 2006) is related to WHAM as it exploits similar ideas and computes the Hamming distance between binary vectors representing sets, which is not related to the problem we deal with in this paper. Also, in all the above studies, query lengths are very short, i.e., 58 base pairs on average, while the database se-

quence size is much longer (up to the size of a genome, i.e., 3.5 billion base pairs). A general framework for sequence matching in both time series and sequence databases has also been proposed (Zhu et al 2012). The framework reports matching results as pairs of query and database subsequences. However, this method is also restricted to short queries of size up to 20 base pairs. Additional methods on short-read sequencing include ARSM (Vergoulis et al 2012) and RCSI (Wandelt et al 2013). All the aforementioned methods are designed for solving the problem of short-read sequencing. Clearly, this problem is completely orthogonal to the problem we are studying in this paper, since our objective is full-sequence matching and not subsequence matching, while no assumptions are being made regarding the expected matching ranges.

Regarding global alignment methods, the Needleman-Wunch method (Needleman and Wunsch 1970) is a brute-force solution to our problem employing the edit distance. In addition, there have been several q-gram based methods proposed in the literature for approximate sequence matching in large sequence databases (Kim et al 2005a; Li et al 2007; Litwin et al 2007; Yang et al 2008; Li et al 2008a), where a q-gram is defined as a subsequence of length q. Q-gram methods are based on the pigeon-hole principle, according to which if two sequences have a certain degree of similarity they must share a minimum number of subsequences (q-grams). In the offline step the occurrence of each q-gram in each of the database objects is recorded in an inverted list. Given a query Q, all the q-grams of the query are extracted and their corresponding inverted lists are used to identify the database objects that share a minimum number of q-grams with the query. Next, a refine step is performed to identify which of the candidates match the search requirements. This method is guaranteed to always return all the objects that match the search criteria. One of the differences between this method and ours is that we do not use all of the possible subsequences of the query; instead, we map it to a reduced alphabet space based on the most frequent subsequences. The actual implementation that we have used for q-grams is from the publicly available Flamingo Package (Behm et al 2010). That package uses a smart implementation of the edit distance that stops the building of the matrix if it can be easily determined that the distance will be larger than the required threshold. Such quick determinations can be obtained using letter counting, and also using partial distance estimates computed by building the lower and upper half of the distance matrix separately.

Another indexing method that seems promising for the problem examined in this paper, and thus has been also implemented, uses a set of reference objects and the triangle inequality (Venkateswaran et al 2006). The distances from the reference objects to the query and to the database objects are used in conjunction with the triangle inequality to determine if a database object is certainly within the required similarity range (and as a result it should be part of the search result), certainly outside of it (excluded from the search result) or a possible candidate. The resulting candidates are then evaluated using the edit distance and the result set is updated. This method is also exact, rendering all the objects that match the search criteria. Its efficiency depends on the pruning power of the distances to the reference objects, and is thus hindered by the low variance in the distribution of pairwise distances.

In a similar flavour, methods in the area of graph indexing have been developed for subgraph discovery using discriminant patterns. For example, Yan et al. (Yan et al

2005) proposed an indexing model based on discriminative frequent subgraph components that are identified through a graph mining process, and used it for building a compact graph index for speeding up subgraph search. In addition, an approximate graph matching technique has been proposed (Tian et al 2007) for similarity search in biological graphs, which allows for node gaps, node mismatches and graph structural differences. Both methods are not directly applicable to our problem setting since we are focusing on sequences and not graphs.

To recap, there exists a plethora of methods in the literature for aligning biological sequences and sequences. Nonetheless, in this paper we focus on the problem of *full-sequence matching*, or global alignment, under the edit distance. Hence, the only applicable competitor methods are q-grams, embedding-based full-sequence matching (Venkateswaran et al 2006) (which also uses the edit distance), and the brute-force edit distance (which is equivalent to the Needleman-Wunch approach).

3 Background

In this section, we provide the basic notation used throughout the paper and formulate our problem. Note that, for the remainder of the paper, we shall use the terms "string" and "sequence" interchangeably.

Consider $\mathbb{X}$ to be a space of strings defined over an alphabet $\mathscr{A}$. A string $X \in \mathbb{X}$ of size $|X|$ is defined as $X = (X_1, \ldots, X_{|X|})$, where $X_i \in \mathscr{A}$ for $i = 1, \ldots, |X|$. A subsequence of X including its first i elements (for any $i \in [1, |X|]$) is denoted as $X^{1:i}$. A collection of strings defines a *string database* and is denoted as $\mathbb{S}$. We will use X to denote a database string and Q to denote a query string. Given two strings Q and X, their edit distance is computed by function $\mathscr{D}(Q,X)$.

3.1 Edit Distance

The edit distance $\mathscr{D}(Q,X)$ is a function measuring how *dissimilar* the strings Q and X are. This is achieved by computing the minimum cost of editing operations (insertion, deletion, and substitution) needed to convert one string to the other. Note that a cost should be specified for each editing operation. These costs are denoted as follows:

- C_{ins}: denotes the cost of the edit operation that inserts a letter to string Q.
- C_{del}: denotes the cost of the edit operation that deletes a letter from string Q.
- $C_{\text{sub}}(Q_j, X_i)$: denotes the cost of the edit operation that replaces letter Q_j with some letter $X_i \neq Q_j$, where $i \in [1, |X|], j \in [1, |Q|]$.

The most commonly used version of edit distance uses $C_{\text{ins}} = C_{\text{del}} = C_{\text{sub}} = 1$, and in that case $\mathscr{D}(Q,X)$ is the smallest total number of insertions, deletions, and substitutions that can convert Q to X. For simplicity, in the remainder of this paper, we assume the above setting of costs.

In order to compute $\mathscr{D}(Q,X)$, we will use an auxiliary matrix a, such that $a^{j,i}$ corresponds to the smallest possible distance between $Q^{1:j}$ and $X^{1:i}$. We also define

an auxiliary function $C(Q_j, X_i)$ that denotes the cost of matching letter Q_j with letter X_i:

$$C(Q_j, X_i) = \begin{cases} C_{\text{sub}} \text{ if } Q_j \neq X_i \\ 0 \text{ if } Q_j = X_i \end{cases} \quad (1)$$

Hence, $\mathscr{D}(Q,X)$ and the corresponding best alignment of Q and X can be found using dynamic programming, by computing $a^{j,i}(Q,X)$ for $j = 1,\ldots,|Q|$ and $i = 1,\ldots,|X|$, as follows:

initialization:

$$a^{0,0} = 0, a^{j,0} = a^{0,i} = \infty \,. \quad (2)$$

loop:

$$a^{j,i}(Q,X) = \min \begin{cases} a^{j,i-1}(Q,X) + C_{\text{ins}} \\ a^{j-1,i}(Q,X) + C_{\text{del}} \\ a^{j-1,i-1}(Q,X) + C(Q_j, X_i) \end{cases} \quad (3)$$

$$(j = 1,\ldots,|Q|; i = 1,\ldots,|X|) \,.$$

termination:

$$\mathscr{D}(Q,X) = a^{|Q|,|X|}(Q,X) \,. \quad (4)$$

Complexity: The evaluation of the edit distance $\mathscr{D}(Q,X)$ requires time $O(|Q||X|)$. We should also note that the alignment path can be found by keeping track, in each application of Equation 3, of the predecessor selected for each (j,i), and by backtracking at termination from position $(|Q|,|X|)$.

3.2 Problem Statement

Given a database $\mathbb{S}$, a query Q, and a similarity range $r = \lfloor \delta |Q| \rfloor$, $\delta \in \mathbb{R}^+$, we want to retrieve all database strings X, such that:

$$\mathscr{D}(Q,X) \leq r.$$

In other words, we are interested in performing a range query in $\mathbb{S}$ to identify all strings that are within edit distance $\delta|Q|$ from Q. Note that Q and r are provided by the user.

4 DRESS: Dimensionality Reduction for Efficient Sequence Search

DRESS works in a filter-and-refine manner. Its key component is the dimensionality reduction technique that it uses to transform the strings from their original representation to a more compact one. Effectively, this results in a *mapping* of the original string space to a new string space of reduced dimensionality. Note that in our case dimensionality corresponds to the length of the strings. At query time, both query and

database strings are mapped to the new string space based on a query-sensitive technique described next. Similarity search under the edit distance is then performed in the new space, where candidate matches are identified. These candidates are passed over to the refine step, where the database strings satisfying the desirable range are identified and returned as the result set. Next, we describe these steps in more detail.

4.1 Filter-and-Refine Framework

4.1.1 Mapping step

DRESS essentially performs dimensionality reduction from the original string space to a new space. This step is called the *mapping step*. For each query Q that belongs to the set of strings $\mathbb{X}$, we identify a set $\mathbb{E} = \{E_1, \ldots, E_t\}$ of codewords consisting of the t most frequent substrings of length l. The set of codewords has the property that there is no pair of strings that have overlapping suffix-prefix. This is provided more formally in the following definition.

Definition 1 (**codewords**) A set of codewords $\mathbb{E}$ is a set of strings such that, for any strings $E_1, E_2 \in \mathbb{E}$, no prefix of E_1 is a suffix of E_2, and no prefix of E_2 is a suffix of E_1.

In our implementation, the set of codewords $\mathbb{E}$ is not fixed. Instead, the number t of codewords and their length l are fixed (and are identified after appropriate training), and the actual codewords are chosen individually for each query. This query-sensitive selection of $\mathbb{E}$ can improve performance in practice, in datasets where the t most frequent codewords of Q typically appear with high frequency in the most similar database matches for Q, but not in database sequences unrelated to Q.

To find the t most frequent codewords of Q, we first count, for each possible codeword of length l, the number of times it occurs in Q. This is done by simply creating an array of size equal to the number of codewords, initializing the contents of that array to 0, and then walking through Q from left to right, examining each substring of length l, and incrementing the corresponding counter in the array. The index of the counter corresponding to a substring can be identified in constant time using a hash function.

After we find the t most frequent codewords of Q, we perform these steps:

1. Set $\mathbb{E} = \emptyset$.
2. Find codeword E with highest count, such that:
   - Codeword E is still not an element of $\mathbb{E}$.
   - No suffix of any codeword of $\mathbb{E}$ is a prefix of E.
   - No prefix of any codeword of $\mathbb{E}$ is a suffix of E.
3. Insert E to $\mathbb{E}$.
4. If $|\mathbb{E}| \neq t$, go to step 2, else we are done.

Next, once we have selected the set $\mathbb{E}$ of codewords, we map the database and the query to a new space according to the occurrence of these codewords. To do so, we first need to provide the definition of the $\mathbb{E}$-mapping.

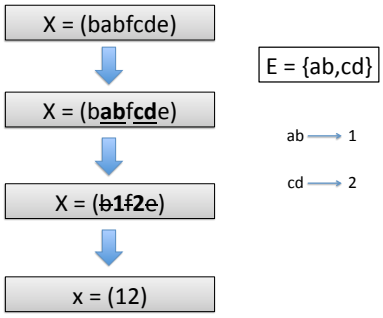

Fig. 1 An example of the dimensionality reduction performed during the mapping step of DRESS.

Definition 2 ($\mathbb{E}$**-mapping**) Given a set of codewords $\mathbb{E}$ and a string X, the $\mathbb{E}$-mapping of X is a new string where all instances of each codeword $E_i \in \mathbb{E}$ are replaced by a new character e_i while the remaining parts of X are removed.

A simple way to obtain the $\mathbb{E}$-mappings of all database sequences would be to simply scan each database sequence and produce its embedding by replacing occurrences of each codeword with the corresponding symbol and ignoring all other letters of the sequence. More specifically, this could be done as follows:

- Each string $X \in \mathbb{S}$ is parsed from left to right.
- When a codeword, e.g., E_i, is found in X, it is assigned with a new letter, e_i, where $i \in [1,t]$. Note that each E_i corresponds to one e_i.
- All the remaining parts of the original string X are deleted. The resulting mapped version of X is denoted as x.

Similarly, the $\mathbb{E}$-mapping of Q is a new string denoted as q. Consider the example shown in Figure 1. The $\mathbb{E}$-mapping of string $(babfcde)$ according to the codeword set $\mathbb{E} = \{ab, cd\}$ is (12), where ab is mapped to 1 and cd is mapped to 2.

The time it takes to create the $\mathbb{E}$-mappings of the database sequences with the procedure described above is linear to the size of the database. However, we can do better than that, by precomputing an inverted index IND of the database, where we specify, for each possible codeword (i.e., each possible substring of length l), all the sequences where it occurs, and the positions at which it occurs at each sequence. In other words, for substring E of length l, IND$[E][i]$ is a (possibly empty) list of all positions where E occurs in the i-th database sequence.

We note that E is a string, whereas in notation IND$[E][i]$ we treat E as an integer index to array IND. We can easily map E to an integer in constant time using a hash function. We should also note that building the inverted index incurs a one-time preprocessing cost. The time it takes to build this index is linear to the size of the database.

At runtime, when we want to compute the $\mathbb{E}$-embedding of the i-th database sequence X_i, we follow these steps:

1. Set Pairs $= \emptyset$

2. For each $E \in \mathbb{E}$:
   - Set e to the letter assigned to keyword E.
   - For each position pos in IND$[E][i]$:
     - Insert pair (e, pos) to Pairs.
3. Sort Pairs in ascending order of the positions (i.e., the second elements of the pairs).
4. Set x_i to be the empty string.
5. For each pair (e, pos) in Pairs, considered in sorted order:
   - Insert letter e to the end of x_i.

At the end of these steps, x_i is the $\mathbb{E}$-embedding of database sequence X_i. The time it takes to compute x_i is $O(|x_i|log|x_i|)$. Typically the length of x_i is significantly smaller than the length of the original database sequence X_i, and thus the method we use for computing the embedding of X_i is faster than simply scanning X_i.

According to the mapping step, the resulting strings are significantly shorter than their original counterparts. For example, the alphabet $\mathscr{A}$ for protein sequences includes 20 letters, which give rise to 400 possible codewords with length $l = 2$. If we only select $t = 4$ codewords, and the selection is made randomly, then we expect that the proposed mapping will reduce the length of a sequence by (on average) a factor of 100. In that case, we expect that computing the edit distance between two dimensionality reduced strings will be faster by a factor of 10,000. Note that, since the selection of codewords is not random, for our method we anticipate that the length of a sequence will be reduced by less than 100 times.

4.1.2 Filter and Refine steps

Next, we perform brute force search between each mapped database string x and the mapped query q under the edit distance. The database strings that are within a certain percentage δ' of the length of the mapped query are considered as candidate matches to the query, in other words they pass the filter step. More information about δ' is given in the following section.

These candidates are further evaluated in the refine step by applying brute force search in the original space, i.e., the edit distance is computed between each candidate database string X and the query Q. Finally, the database strings that are within a δ percentage of the length of the query belong to the result set of matches to Q. The pseudo-code of our method is given by Algorithm 1, and in Figure 2 we show the main steps of DRESS.

4.1.3 Search Range

The percentage δ' that we use in the target space depends on δ, i.e., the percentage that the user has specified in the original space, and is equal to δf, where f is a scaling factor greater than one. For example, if $\delta = 15\%$ and $f = 2$, then $\delta' = 30\%$. δ' is higher than δ to account for the cases where the distance in the new string space is a higher percentage of the mapped query length. This can easily happen because the mapped query and database strings are drastically shorter than the original strings.

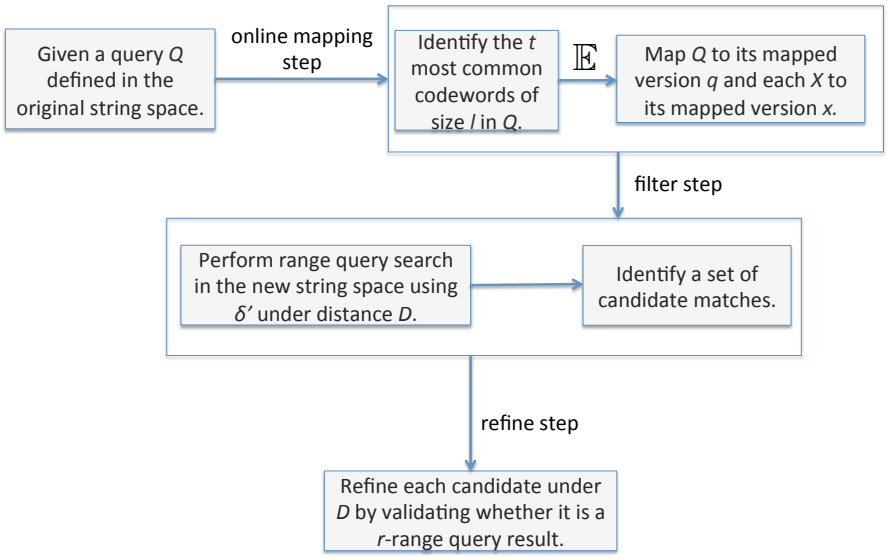

Fig. 2 An overview of the main steps of DRESS.

While we expect similar strings to map to similar mapped strings, the loss of information incurred by the mapping can lead to higher distances as percentages of query length. Good values for f are estimated in a straight-forward manner, using a training set of queries. This training set can be chosen as a subset from the database, or alternatively we can use randomly generated strings. More details on how we set the value of f can be found in Section 5.1.4.

4.2 Theoretical Analysis

Here, we provide a theoretical analysis of DRESS. We mainly show that the mapping is contractive and prove that the definition requirements of a codeword are necessary to guarantee the contractiveness of the mapping. A mapping $\mathbb{E}$ is contractive if for any X, Y, $\mathscr{D}(X,Y) \geq \mathscr{D}(\mathbb{E}(X),\mathbb{E}(Y))$.

We first start by showing that the conditions for a legal set of codewords $\mathbb{E}$ from Definition 1 are necessary for proving that the mapping is contractive. We can show this with a simple example. Let $\mathbb{E} = \{ca, ac\}$. We note that, in $\mathbb{E}$, codeword ac has a suffix (its last letter c) that is a prefix of another codeword (ca), and thus $\mathbb{E}$ is not a legal set of codewords. Suppose that we map ca to new letter 1 and we map ac to new letter 2. Let $X = (eaca)$, and $Y = (caca)$. The $\mathbb{E}$-mapping of X is $x = (2)$ (ac and ca both occur, but their occurrences overlap and ac takes precedence), and the

Algorithm 1: DRESS

Input: Query sequence $Q \in \mathbb{X}$, database $\mathbb{S} \subset \mathbb{X}$, percentage δ, number of codewords t, adjustment scaling factor for the target space f.
Output: Result set of database objects $\subset \mathbb{S}$.

```
1  begin
2      Find the top t most frequent codewords of Q, 𝔼.
3      Map Q according to 𝔼, yielding q.
4      r = ⌊δ|Q|⌋.
5      δ' = δf.
6      for each X ∈ 𝕊 do
7          if abs(|X| − |Q|) > r then
8              //(Length Filter)
9              Reject X.
10         end
11         else
12             Map X according to 𝔼: x = E(X).
13             if 𝒟(x,q) ≤ δ'|q| then
14                 //X is a candidate - refine the search
15                 if 𝒟(X,Q) ≤ r then
16                     Add X to the result set.
17                 end
18             end
19         end
20     end
21 end
```

$\mathbb{E}$-mapping of Y is $y = (11)$. The lower bound property is not satisfied: $\mathscr{D}(X,Y) = 1 \not\geq \mathscr{D}(x,y) = 2$, and thus the mapping is not contractive.

The reason why this mapping is not contractive is that making a single edit operation on X, namely replacing the initial e with a c, results to two edit operations on the $\mathbb{E}$-mapping of X, converting (2) to (11). This situation arises because, in $(eaca)$, there is an occurrence of codeword *ac overlapping* with an occurrence of codeword *ca*. In computing the $\mathbb{E}$-mapping of $(eaca)$, *ac* takes precedence because it is to the left of *ca*. By replacing e with c, we create a new occurrence of *ca*, but also by removing the occurrence of *ac* we allow the other occurrence of *ca* to be reflected on the $\mathbb{E}$-mapping. If the set of codewords is legal, and thus no suffix of a codeword is a prefix of another codeword, then it is not possible for occurrences of codewords to be overlapping,

Proposition 1 *If the set of codewords $\mathbb{E}$ is legal according to Definition 1, then a single edit operation (i.e., a single insertion, deletion, or substitution) on a string Y can only lead to zero or one edit operation on the $\mathbb{E}$-mapping of Y.*

Proof Let y be the $\mathbb{E}$-mapping of Y. An edit operation can happen either within the occurrence of a codeword (destroying that codeword), or outside of the occurrence of a codeword.

– **Case 1: the edit operation occurs within the occurrence of a codeword E_1, thus destroying that occurrence.** Suppose that e_1 is the letter that the $\mathbb{E}$-mapping replaces E_1 with. We have two subcases:

  - **Subcase 1a:** another occurrence of a codeword E_2 is generated, in which case the edit operation causes a substitution of letter e_1 at the appropriate position of y. It is also possible that $E_2 = E_1$. For example, if ca is a codeword, $Y = (caaaa)$, and we delete the second letter, essentially we destroy an occurrence of keyword ca but at the same time we create another occurrence of the same keyword. If $E_2 = E_1$, then the edit operation on Y leads to no edit operation on y, otherwise it leads to a single edit operation (a substition) on y.
  - **Subcase 1b:** no occurrence of another codeword is generated, which causes a deletion of letter e_1 at the appropriate position of y.

  Since codeword occurrences do not overlap, all other existing codeword occurrences in Y are not affected by the edit operation in Case 1.
- **Case 2: the edit operation is outside any existing occurrence of any codeword in Y.** Again, we have two subcases:
  - **Subcase 2a:** a new occurrence of a codeword E_2 is generated, in which case the edit operation causes an insertion of the letter corresponding to E_2 at the appropriate position of y.
  - **Subcase 2b:** no occurrence of another codeword is generated, in which case no changes to y are made

  We note that, if a new occurrence of some codeword E_2 has been generated in case 2, this new occurrence cannot overlap with any pre-existing codeword occurrences in Y, because it is impossible for occurrences of codewords to overlap as long as the set of codewords is legal according to Definition 1. Therefore, any pre-existing occurrences of codewords in Y are not affected by the edit operation in case 2.

In summary, under subcase 2b, and sometimes under subcase 1a, y is not changed. In all other cases y is changed by a single edit operation. □

We are now ready to prove that the proposed mapping is contractive.

Theorem 1 (**Contractiveness**) *Let X, Y be two strings, and $\mathbb{E}$ be a set of codewords that is legal according to Definition 1. Let x and y be the $\mathbb{E}$-mappings of X and Y. Then $\mathscr{D}(X,Y) \geq \mathscr{D}(x,y)$.*

Proof: Let $D = \mathscr{D}(X,Y)$. Then, D edit operations suffice to convert X to Y. Each of those edit operations causes zero or one edit operation to x, so in total the D edit operations on X cause at most D edit operations on x. The result of these at most D edit operations on x converts x into y, so $\mathscr{D}(x,y) \leq D = \mathscr{D}(X,Y)$. □

Contractiveness is important theoretically; when a mapping is contractive, then filter-and-refine retrieval can be performed so as to guarantee retrieving all results within a desired range (Hjaltason and Samet 2003). However, in the filter-and-refine method described in Sections 4.1.2 and 4.1.3, we actually do not use the contractiveness property. The method described in those sections obtains significantly more attractive trade-offs between accuracy and efficiency, by filtering more aggressively and missing a small percentage of matches within the desired range. Using contractiveness as described by Hjaltason and Samet (Hjaltason and Samet 2003) would lead to not missing any matches, but at the cost of much higher running time.

5 Experiments

In this section, we benchmark the performance of DRESS on the biology domain, and demonstrate its superiority in terms of retrieval cost and recall against two state-of-the-art methods for full-sequence matching under the edit distance.

5.1 Experimental Setup

5.1.1 Datasets

For our experiments we used two datasets from the biology domain, a protein dataset and a DNA dataset.

Proteins. The UniProt dataset [1] is a commonly used freely available dataset that consists of protein sequences. Specifically, it includes 530,264 strings defined over an alphabet of 25 letters. The protein strings have variable length ranging from 2 to 35213 (amino-acids). To study the performance of DRESS against the competitor methods in terms of different sequence sizes, we created three datasets. These datasets are disjoint subsets of UniProt and are referred to as $Dataset_{401}^{800}$, $Dataset_{801}^{1600}$, and $Dataset_{1601}^{max}$. Each dataset consists of three subsets: the validation set, the test set, and the database (Table 1). $Dataset_{401}^{800}$ contains 130,962 sequences in total (100 for validation, 500 for testing, and the rest for the database), and the length of each sequence is between 401 and 800. $Dataset_{801}^{1600}$ consists of 28,155 sequences (100 for validation, 500 for testing, and 27,555 for the database), and the lengths range from 801 to 1600. The third dataset, $Dataset_{1601}^{max}$, includes all sequences of UniProt with lengths greater than 1600, which are in total 3,879 (100 for validation, 100 for testing, and 3,679 for the database). For each dataset all 3 subsets (validation, test, and the database) are disjoint, and membership in each subset was randomly assigned.

DNA. We have created two datasets of DNA sequences taken from Human Chromosome 1 freely available at the NCBI repository [2]. The full size of this chromosome is $249,250,621$ bases. The first dataset contains $500,500$ DNA sequences randomly selected from the full chromosome (without duplicates) of lengths between 100 and 500 nucleotides. The second dataset contains $500,500$ sequences, selected again in a similar manner, of lengths between 500 and $1,000$ bases. Similar to UniProt, for each dataset we created a test set, a validation set, and a database. First, for each dataset we randomly selected 100 sequences for the test set and 100 for the validation set. Next, we added different levels of noise (from 5% to 30% in steps of 5%) to each sequence. Specifically, for each noise level $z\%$, we performed edit operations (insertions, deletions, or substitutions) on $z\%$ of the sequence length. This resulted in a set of 600 new test sequences (100 per noise level) and 600 validation sequences. The remaining $500,500$ sequences, including the original (without noise) test and validation set sequences, were used as the database. The two datasets are referred to as $Dataset_{100}^{500}$ and $Dataset_{500}^{1000}$ (Table 1).

[1] http://www.ebi.ac.uk/uniprot/

[2] ftp.ncbi.nlm.nih.gov

Table 1 Summary of datasets used in the experimental evaluation.

	UniProt			DNA	
	$Dataset_{401}^{800}$	$Dataset_{801}^{1600}$	$Dataset_{1601}^{max}$	$Dataset_{100}^{500}$	$Dataset_{500}^{1000}$
sequence length	$[401, 800]$	$[801, 1600]$	$[1601, 35213]$	$[100, 500]$	$[500, 1000]$
total # of sequences	130,962	28,155	3,879	500,500	500,500
# of seq. in validation set	100	100	100	600	600
# of seq. in database	130,362	27,555	3,679	500,500	500,500
# of seq. in test set	500	500	100	600	600

The full UniProt dataset, its three subsets, and the two DNA datasets we experimented with can be found at `http://vlm1.uta.edu/~akotsif/dress`.

5.1.2 Methods

We compare DRESS with two competitor methods that have been proposed for speeding up whole sequence matching and can be used in biological databases:

- Reference-based embeddings (RBE): the method proposed by Venkateswaran et al (2006) that uses distances to reference objects to define an embedding space and the triangle inequality to quickly identify a small set of candidate matches. For this method we have built a new index for each search range.
- q-grams: the method described by Li et al (2008a) that uses inverted indexes of q-gram occurrences to quickly identify candidate matches. For our experiments we used the publicly available Flamingo Package code (Behm et al 2010). Note that for simplicity we have used exact q-grams. As it has also been shown by Papapetrou et al (2009) the performance of near-exact q-grams deteriorates in a similar manner since they are designed to tolerate only low values of δ, i.e., less than 15%.

Note that the brute force approach is the edit distance, as described in Section 3.1, that compares each query with all database sequences in the original space.

5.1.3 Evaluation Measures

The measures that we used for the comparative evaluation of DRESS with RBE and q-grams are provided next.

Recall: We measure the percentage of database objects that are within the desired search range and are successfully retrieved by the system. When we report cumulative results on a set of queries, we sum up the total number of correct results that the system retrieves and we divide it by the total number of correct results that should be retrieved. RBE and q-grams are exact methods and they guarantee a recall of 100%, while DRESS does not, since it is an approximate method. Hence, in the experimental results we only report the accuracy of DRESS. It is important to note that the refine step of our method assures no false matches.

Runtime: We compute the average runtime per query that is needed by each method to retrieve the final matches.

Retrieval cost: One limitation of measuring efficiency using running times is that those times can depend significantly on particular aspects of the hardware, such as memory, cache size, bus speed, and so on. Running times also depend on the efficiency of the implementation, compiler optimizations, and choice of programming language. As an alternative platform-independent measure we employ a more theoretical estimate, where we try to use upper-bound (and thus less favorable) estimates for DRESS and lower-bound (and thus more favorable) estimates for the competitors. We believe that these numbers help in obtaining a clearer picture of the efficiency that our method achieves compared to the competitors. We note that one-time preprocessing costs, like building the inverted index of Section 4.1.1, are considered neither for our method nor for the competitors, in calculating retrieval cost.

Our measure of retrieval efficiency is reported as a percentage of brute force search. Brute force has to compute entries on dynamic programming tables. If $|Q|$ denotes the length of the query and $|\mathbb{S}|$ denotes the sum of lengths of database strings, then the number of entries that must be computed in these dynamic programming tables is $|Q||\mathbb{S}|$. Since all methods we evaluate have a refine step, we measure, at each experiment, a quantity that we denote as x'. This quantity corresponds to the sum of lengths of all database strings that are considered at the refine step. Thus, fraction $x'/|\mathbb{S}|$ is a lower bound of the computational cost of a method, since it does not take into account the cost of any processing outside of the refine step (such as the cost of the filter step).

For q-grams and RBE we report $x'/|\mathbb{S}|$ as our estimate of retrieval efficiency. As noted above, this is a favorable approach for those methods as it ignores all costs outside of the refine step.

For our method we add to $x'/|\mathbb{S}|$ two additional quantities: an estimate of the online mapping cost and an estimate of the filter cost. Taking advantage of the inverted index of the database, for the mapping cost we consider every letter of the mapped database strings to cost as much as computing an entry in a dynamic programming table. For the filter cost, the computation is more straightforward as the filter step measures edit distances in the new string space. Hence, we count the total number of entries in the dynamic programming tables computed during this step. So, overall, the efficiency of our method is measured in units of dynamic programming table entries, and we divide that cost by the cost of brute force search.

To report the retrieval efficiency over a set of queries, we provide the average of the retrieval costs, $x'/|\mathbb{S}|$ (plus the two additional quantities for DRESS), attained by the queries.

5.1.4 Implementation Choices

DRESS. To implement DRESS we need to make certain choices. Here, we document the choices we have made and the process for making those choices.

- **Length of codewords (l):** Performing training on the validation set, we obtained satisfactory results in terms of recall for codewords with $l = 2$ for all datasets.

However, for completeness, we also experimented with $l = 3$ and $l = 4$ for the three UniProt datasets. In the experimental results, unless otherwise stated, we present the results for $l = 2$.

- **Number of codewords (t):** We have experimented on the validation set with using 2, 3, and 4 codewords. For all datasets, using 4 codewords worked better in terms of the tradeoff between recall and cost, so that is the setting we used for the test sets.
- **Scaling factor (f):** As a reminder, the scaling factor f is used in equation $\delta' = f\delta$, where δ is the user-specified percentage, and δ' is the percentage that our system uses in the new space. Larger values of f bring recall up to 100%, as they eventually lead to the filter step not pruning any items and passing the entire database to the refine step, but also yield a search cost closer to the cost of brute force search. To choose f for each experiment, we exploited the validation set and we identified the smallest value of f that produced over 99% recall. The reason for choosing 99% was that we wanted to be close to 100% recall, but at the same time to prevent f from being determined by a few query outliers. Note that, while the chosen value of f produced over 99% recall on the validation set, we still needed to measure the actual recall obtained on the test set, which is shown in our experimental results.

q-grams. For q-grams, we made the following choices in our experiments:

- We used the DivideSkip merging algorithm, because it was reported as being the most efficient one by Li et al (2008a), and it also performed better in a few initial experiments with our datasets.
- For all UniProt datasets we used a filter tree with a length filter and fan of 10 for 4-grams. We ran experiments for $\delta = 5\% - 40\%$ with 5% step and q-gram lengths between 2 and 6, and we found 4 to be the optimal value. We also experimented with different fanout values, 5, 20, and 30, for all search ranges, but they did not make a difference. Since the retrieval cost of q-grams, especially for large δ values, was approaching that of brute force, we did not consider them further for the DNA datasets.

RBE. In order to construct the reference set of UniProt $Dataset_{401}^{800}$ and $Dataset_{801}^{1600}$ we used 2,000 random sample objects from each database, while for $Dataset_{1601}^{max}$ we used 500 reference objects due to the total size of this dataset. For the DNA $Dataset_{100}^{500}$ and $Dataset_{500}^{1000}$ 500 reference objects were randomly selected from each database. Thus, when a query is presented, each database object is evaluated based on the $2,000$ triangle inequalities for the first two datasets, and the 500 inequalities for the other three datasets.

Recall that, for both q-grams and RBE, in the reported cost we have excluded all costs outside of the refine step.

All methods were implemented in Java, and experiments were ran on a 2GHz Intel Xeon (QuadCore, but we used a single core) with 4GB of RAM, under Windows 7.

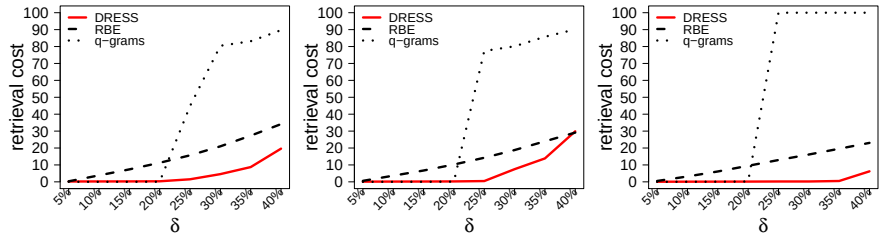

Fig. 3 Retrieval cost for all methods for UniProt $Dataset_{401}^{800}$ (left), $Dataset_{801}^{1600}$ (middle), and $Dataset_{1601}^{max}$ (right), respectively. Parameter δ is the percentage of the query length within which we want to retrieve database matches.

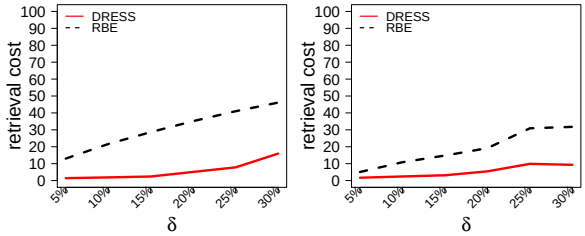

Fig. 4 Retrieval cost for DRESS and RBE methods for the DNA datasets $Dataset_{100}^{500}$ (left) and $Dataset_{501}^{1000}$ (right). Parameter δ is the percentage of the query length within which we want to retrieve database matches.

5.2 Experimental Results

5.2.1 Retrieval Cost and Recall

The retrieval cost (as defined in Section 5.1.3) of all methods for searches in the three UniProt datasets within ranges $5\% - 40\%$ (with a step of 5%) of the query length are shown in Tables 2, 3, 4, and Figure 3. Tables 5 and 6 and Figure 4 show the retrieval cost for the two DNA datasets, respectively, DRESS and RBE methods for searches within ranges $5\% - 30\%$ (with 5% step) of the query length. We note that our method achieves significantly lower costs than the competitors, especially for large range values.

More specifically, as an example of the UniProt datasets, the retrieval cost of DRESS with $l = 2$ for the three datasets is just 1.51%, 0.46%, and 0.21% for $\delta =$ 25%, and 8.73%, 13.85%, and 0.5% for $\delta = 35\%$, respectively. These costs are significantly lower than those of the competitor methods. DRESS beats RBE for all datasets and search ranges. For $Dataset_{401}^{800}$ the retrieval cost of DRESS is lower by about 1.5, 16, 29, 35, 10, 4.6, 3, and 1.7 times for the eight search ranges, for $Dataset_{801}^{1600}$ by 5, 23, 35, 41, 31, 2.5, and 1.7 times for $\delta = 5\% - 35\%$, and for $Dataset_{1601}^{max}$ by 7.5, 31, 50, 63, 61, 75, 39, and 3.7 times for the eight ranges, respectively. Based on these results, we observe that the longer the test and database sequences, the better our method performs.

Regarding q-grams, their retrieval cost is much worse than that of RBE for $\delta \geq 25\%$ for all datasets. For the first dataset the cost is close to 45%, 80%, 83%, and 90%, while for $Dataset_{801}^{1600}$ the cost is 77%, 80%, 86%, and 90% for $\delta = 25\%, 30\%, 35\%, 40\%$, respectively. For $Dataset_{1601}^{max}$ the cost of q-grams is more than 99% for $\delta \geq 25\%$. As a result, our method achieves approximately 30, 18, 10, and 5 times lower cost than q-grams for the first dataset and $\delta = 25\%, 30\%, 35\%, 40\%$, respectively. For $Dataset_{801}^{1600}$ DRESS achieves more than 167, 10, 6, and 3 times lower cost than q-grams for the same four ranges. For $Dataset_{1601}^{max}$ and $\delta = 25\%, 30\%$ DRESS has more than 461 times lower cost than q-grams, for $\delta = 35\%$ more than 199 times, and for $\delta = 40\%$ about 16 times.

For the DNA datasets, the retrieval cost for all δ values is less than 16%, and it is much lower than that of RBE. Specifically, for $Dataset_{100}^{500}$, the cost of DRESS is lower by about 9, 11.5, 12, 6.9, 5.2, and 2.9 times for $\delta = 5\% - 30\%$, and for $Dataset_{500}^{1000}$, the cost of DRESS is lower by about 3, 4.5, 5, 3.5, 3, and 3.4 times for $\delta = 5\% - 30\%$, respectively.

We would like to underline that these comparisons are unfair for our method, since the retrieval costs for the two competitor methods do *not* include any costs outside of the refine step as opposed to DRESS, which includes *all* costs. Thus, the gain of using DRESS is even larger than each factor reported above. Another observation is that DRESS benefits significantly from very large sequences. We also note that, while our method does not guarantee 100% recall, the actual recall obtained does not fall below 98% and is actually measured above 99% for most cases, and even 100% (for UniProt $Dataset_{1601}^{max}$ and $\delta = 40\%$, and most δ values for the DNA datasets).

Table 2 Retrieval cost for all methods for UniProt $Dataset_{401}^{800}$. Parameter δ is the percentage of the query length within which we want to retrieve database matches. Note that in the reported cost for RBE and q-grams all costs outside of the refine step have been excluded.

δ	DRESS($l = 2$)		RBE	q-grams
	recall (%)	Cost (%)	Cost (%)	Cost (%)
5%	98.20	0.2127	0.3172	0.0047
10%	98.97	0.2327	3.8266	0.0098
15%	99.22	0.2566	7.3660	0.0165
20%	98.89	0.3253	11.2688	0.0541
25%	99.44	1.5121	15.6807	44.8125
30%	98.95	4.5712	20.9975	80.4539
35%	98.57	8.7319	27.3194	83.1394
40%	98.96	19.6296	34.1353	89.6902

5.2.2 Runtime

The average brute force runtime per query for UniProt $Dataset_{401}^{800}$, $Dataset_{801}^{1600}$, and $Dataset_{1601}^{max}$ is 256.80, 176.93, and 178.29 seconds, for DNA $Dataset_{100}^{500}$ it is 279.16, and for $Dataset_{500}^{1000}$ it is 1,646.22 seconds, respectively.

In Table 7 and Figure 5 we show the runtimes for the three UniProt datasets and the three methods for all search ranges. The runtimes suggest that DRESS with

Table 3 Retrieval cost for all methods for UniProt $Dataset_{801}^{1600}$. Parameter δ is the percentage of the query length within which we want to retrieve database matches. Note that in the reported cost for RBE and q-grams all costs outside of the refine step have been excluded.

δ	DRESS ($l=2$)		RBE	q-grams
	recall (%)	Cost (%)	Cost (%)	Cost (%)
5%	99.12	0.1277	0.5850	0.009
10%	99.94	0.1601	3.6452	0.0149
15%	99.14	0.1929	6.6974	0.0306
20%	98.83	0.2483	10.2436	0.1238
25%	98.82	0.4629	14.3579	77.3736
30%	99.86	7.4873	18.7941	80.1438
35%	99.72	13.8502	23.9686	85.7722
40%	99.77	29.8850	29.1247	90.1262

Table 4 Retrieval cost for all methods for UniProt $Dataset_{1601}^{max}$. Parameter δ is the percentage of the query length within which we want to retrieve database matches. Note that in the reported cost for RBE and q-grams all costs outside of the refine step have been excluded.

δ	DRESS ($l=2$)		RBE	q-grams
	recall (%)	Cost (%)	Cost (%)	Cost (%)
5%	98.00	0.0686	0.5133	0.0183
10%	98.94	0.1039	3.1776	0.0500
15%	99.47	0.1226	6.1311	0.0769
20%	99.54	0.1505	9.4873	0.1951
25%	99.63	0.2100	12.8964	99.9937
30%	99.72	0.2168	16.1533	100
35%	99.51	0.5014	19.5718	100
40%	100	6.2159	23.0382	100

Table 5 Retrieval cost for DRESS and RBE for the DNA $Dataset_{100}^{500}$. Parameter δ is the percentage of the query length within which we want to retrieve database matches. Note that in the reported cost for RBE all costs outside of the refine step have been excluded.

δ	DRESS ($l=2$)		RBE
	recall (%)	Cost (%)	Cost (%)
5%	100	1.4304	13.0093
10%	100	1.8792	21.6184
15%	100	2.4021	28.6427
20%	98	5.1251	35.1539
25%	99	7.8182	41.0073
30%	100	15.9593	46.1054

Table 6 Retrieval cost for DRESS and RBE for the DNA $Dataset_{500}^{1000}$. Parameter δ is the percentage of the query length within which we want to retrieve database matches. Note that in the reported cost for RBE all costs outside of the refine step have been excluded.

δ	DRESS ($l=2$)		RBE
	recall (%)	Cost (%)	Cost (%)
5%	100	1.6818	5.0706
10%	99	2.4278	10.8592
15%	100	3.1381	14.7995
20%	100	5.4510	19.1777
25%	100	9.9005	30.9222
30%	100	9.3096	31.7492

Table 7 Runtimes for all methods and UniProt datasets. Parameter δ is the percentage of the query length within which we want to retrieve database matches. Note that the runtimes for RBE include only the refine step, and the results reported for DRESS are for $l = 2$. All runtimes are reported in seconds.

	$Dataset_{401}^{800}$			$Dataset_{801}^{1600}$			$Dataset_{1601}^{max}$		
δ	DRESS	RBE	q-grams	DRESS	RBE	q-grams	DRESS	RBE	q-grams
5%	0.45	2.77	0.02	0.11	1.43	0.04	0.08	0.62	0.11
10%	0.35	13.19	0.04	0.21	7.03	0.06	0.15	3.75	0.23
15%	0.50	23.25	0.08	0.31	12.68	0.13	0.19	7.39	0.34
20%	0.75	34.22	0.22	0.44	19.24	0.51	0.24	11.74	0.62
25%	5.51	40.50	24.09	0.85	26.81	139.24	0.35	16.30	75.31
30%	10.58	60.88	78.75	13.48	35.05	186.52	0.37	20.90	96.07
35%	20.12	78.36	148.18	24.90	44.56	219.80	0.89	25.90	122.66
40%	45.00	96.59	208.94	53.68	54.05	244.18	11.17	31.24	133.70

Table 8 Runtimes for DRESS and RBE methods and DNA datasets. Parameter δ is the percentage of the query length within which we want to retrieve database matches. Note that the runtimes for RBE include only the refine step, and the results reported for DRESS are for $l = 2$. All runtimes are reported in seconds.

	$Dataset_{100}^{500}$		$Dataset_{500}^{1000}$	
δ	DRESS	RBE	DRESS	RBE
5%	4.4	47.35	48.18	325.65
10%	5.39	76.08	54.68	499.42
15%	7.11	100.53	73.22	636.37
20%	14.7	121.28	115.72	772.42
25%	22.25	140.91	192.35	941.61
30%	44.56	158.01	187.33	1021.47

$l = 2$ is much faster than RBE for all datasets and search ranges. Specifically, for $Dataset_{401}^{800}$ DRESS is faster by more than 6, 37, 46, 45, 7, 5.7, 3.8, and 2.1 times for $\delta = 5\% - 40\%$ (with 5% step), for $Dataset_{801}^{1600}$ by more than 13, 33, 40, 43, 31, 2.5, and 1.7 times for $\delta = 5\% - 35\%$, and for the third dataset by about 7.8, 25, 39, 49, 47, 56, 29, and 3 times for the eight search ranges. Notice that the runtimes of RBE include *only* the refine step. Hence, DRESS is even more efficient than the aforementioned factors.

With regard to q-grams competitor method, for the first UniProt dataset DRESS (with $l = 2$) is slower for $\delta = 5\%, 10\%, 15\%, 20\%$, but faster by more than 4.3, 7.4, 7.3, and 4.6 times for $\delta = 25\%, 30\%, 35\%, 40\%$, respectively. For $Dataset_{801}^{1600}$ DRESS is faster than q-grams for $\delta = 20\%, 25\%, 30\%, 35\%, 40\%$ by more than 1.15, 163, 13.5, 8.5, and 4.5 times, respectively. For the last UniProt dataset q-grams are slower than DRESS by approximately 1.4, 1.5, 1.8, 2.6, 215, 260, 138, and 12 times for the eight search ranges. It should be mentioned that for $Dataset_{801}^{1600}$ the runtimes of q-grams for $\delta \geq 30\%$ are higher than the runtime of the brute force approach, making the benefit obtained by our method much more obvious. In Figure 5 we demonstrate the runtime of all methods for the last dataset and all search ranges.

In Table 8 and Figure 6 we show the runtimes for the DNA datasets, for DRESS (with $l = 2$) and RBE and all search ranges. For $Dataset_{100}^{500}$ DRESS is faster than RBE for more than 10.7, 14.1, 14.1, 8.2, 6.3, and 3.5 times, and for $Dataset_{500}^{1000}$ for more than 6.7, 9.1, 8.6, 6.6, 4.8, and 5.4 times for the six search ranges $\delta = 5\% - 30\%$, respectively, showing again the efficiency of our method.

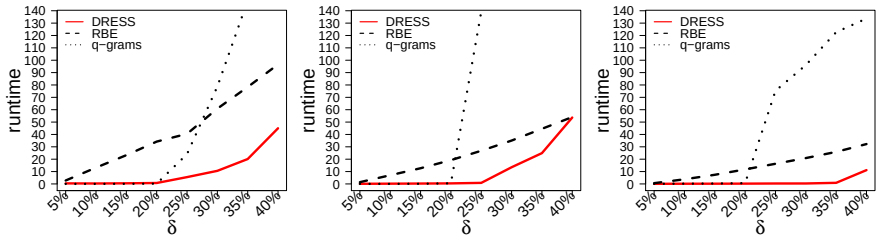

Fig. 5 Runtime in seconds for all methods for UniProt $Dataset_{401}^{800}$ (left), $Dataset_{801}^{1600}$ (middle), and $Dataset_{1601}^{max}$ (right), respectively. Parameter δ is the percentage of the query length within which we want to retrieve database matches.

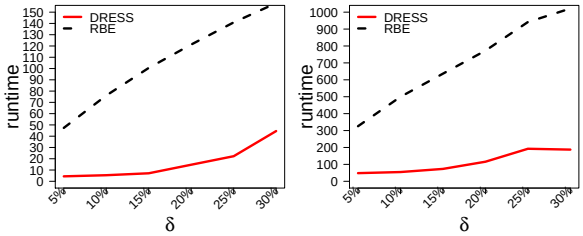

Fig. 6 Runtime in seconds for DRESS and RBE for DNA $Dataset_{100}^{500}$ (left) and $Dataset_{500}^{1000}$ (right). Parameter δ is the percentage of the query length within which we want to retrieve database matches.

5.2.3 DRESS

In Tables 9, 10, 11, 14 and 15 we see the performance of DRESS with $l = 2$ for different search ranges in the five datasets. For each search range we show the recall, retrieval cost, runtime, and also the scale factor f and resulting δ' percentage in the new space. As discussed in Sections 5.2.1 and 5.2.2, DRESS achieves significantly lower cost and runtimes compared to the competitor methods. In addition, it attains very high recall, between 98% and 100%, in all cases. We also observe that for the UniProt datasets f ranges between 1.4 and 4.3, depending on δ, while for the DNA datasets f ranges from 1.3 to 2.

For completeness, in Tables 12 and 13 we also present the recall, retrieval cost, and runtime of DRESS with $l = 3$ and $l = 4$, respectively, for all UniProt datasets and search ranges. For each of the three datasets, the scaling factor used for each δ is the one presented in Tables 9, 10, and 11, respectively. Based on these results, we observe that the highest recall is 97.87% (for $Dataset_{1601}^{max}$ and $\delta = 10\%$), and that for all datasets and search ranges the recall values are much lower than the respective ones for DRESS with $l = 2$; in some cases the recall is even less than 70%. In addition, DRESS with $l = 3$ achieves much better recall than DRESS with $l = 4$ for all UniProt datasets and search ranges, except for $Dataset_{401}^{800}$ where it is worse for $\delta = 5\%, 10\%, 20\%$. The reason for obtaining lower recall values for DRESS with $l = 3$ and $l = 4$ than DRESS with $l = 2$ is because there are not many codewords with such lengths and high frequency. As a result, the length of the mapped strings is very small, and useful information is lost. Referring to the retrieval cost, for all cases it

is less than 1% showing that the number of database strings evaluated is very small. However, this is reflected in the recall attained, since oftentimes the database strings used in the refine step do not include the correct ones (of which the distance from the queries is within the search range). Note that the runtimes are also very small and, in general, lower than the respective ones for DRESS with $l = 2$ (especially for large δ values).

Table 9 The performance of DRESS with $l = 2$ on UniProt $Dataset_{401}^{800}$. The number of codewords is 4, and δ' is the percentage used in the new string space: $\delta' = \delta f$.

δ	recall (%)	Cost (%)	Runtime (seconds)	f	δ' (%)
5%	98.20	0.2127	0.45	4.3	21.50
10%	98.97	0.2327	0.35	2.9	29.00
15%	99.22	0.2566	0.5	2.5	37.50
20%	98.89	0.3253	0.75	2.2	44.00
25%	99.44	1.5121	5.51	2.1	52.50
30%	98.95	4.5712	10.58	1.9	57.00
35%	98.57	8.7319	20.12	1.7	59.50
40%	98.96	19.6296	45.00	1.6	64.00

Table 10 The performance of DRESS with $l = 2$ on UniProt $Dataset_{801}^{1600}$. The number of codewords is 4, and δ' is the percentage used in the new string space: $\delta' = \delta f$.

δ	recall (%)	Cost (%)	Runtime (seconds)	f	δ' (%)
5%	99.12	0.1277	0.11	3.6	18.00
10%	99.94	0.1601	0.21	3.5	35.00
15%	99.14	0.1929	0.31	2.7	40.50
20%	98.83	0.2483	0.44	2.2	44.00
25%	98.82	0.4629	0.85	1.9	47.50
30%	99.86	7.4873	13.48	1.9	57.00
35%	99.72	13.8502	24.9	1.7	59.50
40%	99.77	29.8850	53.68	1.6	64.00

Table 11 The performance of DRESS with $l = 2$ on UniProt $Dataset_{1601}^{max}$. The number of codewords is 4, and δ' is the percentage used in the new string space: $\delta' = \delta f$.

δ	recall (%)	Cost (%)	Runtime (seconds)	f	δ' (%)
5%	98.00	0.0686	0.08	2.9	14.50
10%	98.94	0.1039	0.15	3	30.00
15%	99.47	0.1226	0.19	2.3	34.50
20%	99.54	0.1505	0.24	2	40.00
25%	99.63	0.2100	0.35	1.8	45.00
30%	99.72	0.2168	0.37	1.5	45.00
35%	99.51	0.5014	0.89	1.4	49.00
40%	100	6.2159	11.17	1.4	56.00

Table 12 The performance of DRESS with $l = 3$ on all UniProt datasets. The scaling factor used for each δ value is the one reported in Tables 9, 10 and 11 for each of the three datasets, respectively. The number of codewords is 4, and the runtimes are reported in seconds.

	$Dataset_{401}^{800}$			$Dataset_{801}^{1600}$			$Dataset_{1601}^{max}$		
δ	Recall (%)	Cost (%)	Runtime	Recall (%)	Cost (%)	Runtime	Recall (%)	Cost (%)	Runtime
5%	86.94	0.1935	0.28	95.22	0.1082	0.04	92.00	0.0567	0.05
10%	80.76	0.1968	0.26	88.79	0.1117	0.09	97.87	0.0782	0.09
15%	88.31	0.2020	0.31	96.57	0.1246	0.13	82.91	0.0894	0.11
20%	78.94	0.2059	0.49	89.38	0.1350	0.19	89.40	0.1063	0.15
25%	81.44	0.2421	1.42	84.64	0.1476	0.26	89.38	0.1353	0.21
30%	80.11	0.3189	1.07	80.67	0.2689	0.52	78.03	0.1391	0.23
35%	64.41	0.3458	1.64	72.80	0.3971	0.80	80.78	0.1820	0.30
40%	68.58	0.7694	2.30	69.54	0.8321	1.53	85.29	0.3846	0.68

Table 13 The performance of DRESS with $l = 4$ on all UniProt datasets. The scaling factor used for each δ value is the one reported in Tables 9, 10 and 11 for each of the three datasets, respectively. The number of codewords is 4, and the runtimes are reported in seconds.

	$Dataset_{401}^{800}$			$Dataset_{801}^{1600}$			$Dataset_{1601}^{max}$		
δ	Recall (%)	Cost (%)	Runtime	Recall (%)	Cost (%)	Runtime	Recall (%)	Cost (%)	Runtime
5%	92.57	0.1952	0.59	90.09	0.1076	0.09	86.00	0.0565	0.16
10%	87.70	0.1969	0.50	86.33	0.1119	0.18	85.11	0.0786	0.17
15%	82.83	0.2006	0.60	88.11	0.1214	0.24	79.37	0.0866	0.19
20%	80.90	0.2048	0.93	75.64	0.1250	0.34	79.26	0.1053	0.24
25%	79.92	0.2179	2.00	64.41	0.1290	0.46	75.82	0.1221	0.27
30%	69.16	0.2194	1.46	64.02	0.1583	0.61	62.25	0.1267	0.31
35%	58.50	0.2235	1.61	49.85	0.1698	0.73	58.39	0.1437	0.32
40%	59.80	0.2587	1.64	52.61	0.2169	0.83	72.07	0.1922	0.40

Table 14 The performance of DRESS with $l = 2$ on DNA $Dataset_{100}^{500}$. The number of codewords is 4, and δ' is the percentage used in the new string space: $\delta' = \delta f$.

δ	recall (%)	Cost (%)	Runtime (seconds)	f	δ' (%)
5%	100	1.4304	4.4	2	10.00
10%	100	1.8792	5.39	1.8	18.00
15%	100	2.4021	7.11	1.5	22.50
20%	98	5.1251	14.7	1.6	32.00
25%	99	7.8182	22.25	1.5	37.50
30%	100	15.9593	44.56	1.4	42.00

Table 15 The performance of DRESS with $l = 2$ on DNA $Dataset_{500}^{1000}$. The number of codewords is 4, and δ' is the percentage used in the new string space: $\delta' = \delta f$.

δ	recall (%)	Cost (%)	Runtime (seconds)	f	δ' (%)
5%	100	1.6818	48.18	1.5	7.50
10%	99	2.4278	54.68	1.5	15.00
15%	100	3.1381	73.22	1.5	22.50
20%	100	5.4510	115.72	1.7	34.00
25%	100	9.9005	192.35	1.5	37.50
30%	100	9.3096	187.33	1.3	39.00

In Tables 16, 19, 20, and 23 we show the difference in average length between the original database strings (Tables 16 and 19) and query strings (Tables 20 and 23) and their mapped versions for DRESS with $l = 2$ after applying the length filter. We also present the respective differences in average length for DRESS with $l = 3$ and $l = 4$ for the database strings (Tables 17 and 18) and query strings (Tables 21 and 22). For a specific δ value in the new space, for each query, the sum of the lengths of the mapped database strings that pass the length filter is found, along

with the average mapped database string length. Then, the average mapped database string length over all queries is computed, which is the value reported in Tables 16-19 for each search range. According to the results, we note that, on average, for DRESS with $l = 2$ the mapped queries were about 18-20 times shorter than the original queries, and that the mapped database strings were about 50 times shorter than the original database strings for UniProt $Dataset_{401}^{800}$ and $Dataset_{801}^{1600}$, and about 35 times shorter for $Dataset_{1601}^{max}$. For DRESS with $l = 3$, the mapped queries were about 50-70 shorter than the original queries, and the mapped database strings about 700 times shorter than the original database strings for $Dataset_{401}^{800}$ and $Dataset_{801}^{1600}$, and 400 shorter for $Dataset_{1601}^{max}$. For DRESS with $l = 4$, the mapped queries were shorter than the original ones by 80-130 times, and the mapped database strings by more than 8,500 times for UniProt $Dataset_{401}^{800}$ and $Dataset_{801}^{1600}$, and about 2,700 times for the last UniProt datasets. Referring to the DNA datasets and DRESS with $l = 2$, on average, mapped queries were about 4.1 times shorter than the original queries, and the mapped database strings were about 4.6 times shorter than the original database strings for $Dataset_{100}^{500}$ and 4.5 times shorter for $Dataset_{500}^{1000}$.

The fact that the mapped queries are not shortened by as large a factor as the database strings is expected; the mapping used for every query (and that is applied to the database, so as to process that query) is query-specific, and identifies the most frequently occurring codewords in the query. This conclusion is more obvious in the UniProt datasets due to the fact that they have a much larger alphabet size than the DNA datasets. Moreover, by looking carefully at column "DRESS-4%" for all UniProt datasets, we conclude that the ratio of the average mapped database string length over the original average database string length decreases as the δ value increases. This is reasonable since, when the search range increases, more and longer database strings pass the length filter, resulting in more deleted elements while the top t most frequent codewords are the same.

Table 16 The average length of the database sequences in the original space and in the new string space produced by DRESS using the top 4 codewords (DRESS-4) and $l = 2$ for all UniProt datasets. The % columns show the ratio between the new size and the original size (e.g., in DRESS-4% for $\delta = 5\%$ and the database of $Dataset_{401}^{800}$, 2.0381% = 10.85/532.36 * 100).

	$Dataset_{401}^{800}$			$Dataset_{801}^{1600}$			$Dataset_{1601}^{max}$		
δ	original	DRESS-4	DRESS-4%	original	DRESS-4	DRESS-4%	original	DRESS-4	DRESS-4%
5%	532.36	10.85	2.0381	1,036.16	21.01	2.0277	2,423.41	72.98	3.0115
10%	532.36	10.77	2.0231	1,036.16	20.85	2.0122	2,423.41	71.09	2.9335
15%	532.36	10.77	2.0231	1,036.16	20.69	1.9968	2,423.41	69.17	2.8542
20%	532.36	10.55	1.9817	1,036.16	20.54	1.9823	2,423.41	67.51	2.7857
25%	532.36	10.55	1.9817	1,036.16	20.40	1.9688	2,423.41	65.27	2.6933
30%	532.36	10.49	1.9705	1,036.16	20.30	1.9592	2,423.41	63.52	2.6211
35%	532.36	10.45	1.9630	1,036.16	20.25	1.9543	2,423.41	60.67	2.5035
40%	532.36	10.44	1.9611	1,036.16	20.26	1.9553	2,423.41	58.63	2.4193

5.2.4 RBE

In Tables 24 and 25 we show the performance of RBE for the three UniProt and the two DNA datasets, respectively. In particular, for different search ranges, we show the retrieval cost, the percentage of database objects that were pruned using the embedding, and the runtime. For completeness, we mention that the embedding cost for

Table 17 The average length of the database sequences in the original space and in the new string space produced by DRESS using the top 4 codewords (DRESS-4) and $l = 3$ for all UniProt datasets. The % columns show the ratio between the new size and the original size.

	$Dataset_{401}^{800}$			$Dataset_{801}^{1600}$			$Dataset_{1601}^{max}$		
δ	original	DRESS-4	DRESS-4%	original	DRESS-4	DRESS-4%	original	DRESS-4	DRESS-4%
5%	532.36	0.77	0.1416	1,036.16	1.51	0.1457	2,423.41	6.99	0.2884
10%	532.36	0.76	0.1428	1,036.16	1.48	0.1428	2,423.41	6.36	0.2624
15%	532.36	0.75	0.1409	1,036.16	1.47	0.1419	2,423.41	6.01	0.2480
20%	532.36	0.74	0.1390	1,036.16	1.45	0.1399	2,423.41	5.80	0.2393
25%	532.36	0.73	0.1371	1,036.16	1.44	0.1390	2,423.41	5.51	0.2274
30%	532.36	0.73	0.1371	1,036.16	1.43	0.1380	2,423.41	5.27	0.2175
35%	532.36	0.73	0.1371	1,036.16	1.43	0.1380	2,423.41	4.96	0.2047
40%	532.36	0.73	0.1371	1,036.16	1.43	0.1380	2,423.41	4/77	0.1968

Table 18 The average length of the database sequences in the original space and in the new string space produced by DRESS using the top 4 codewords (DRESS-4) and $l = 4$ for all UniProt datasets. The % columns show the ratio between the new size and the original size.

	$Dataset_{401}^{800}$			$Dataset_{801}^{1600}$			$Dataset_{1601}^{max}$		
δ	original	DRESS-4	DRESS-4%	original	DRESS-4	DRESS-4%	original	DRESS-4	DRESS-4%
5%	532.36	0.07	0.0131	1,036.16	0.14	0.0135	2,423.41	1.33	0.0549
10%	532.36	0.06	0.0113	1,036.16	0.13	0.0125	2,423.41	1.06	0.0437
15%	532.36	0.06	0.0113	1,036.16	0.12	0.0116	2,423.41	0.95	0.0392
20%	532.36	0.06	0.0113	1,036.16	0.12	0.0116	2,423.41	0.90	0.0371
25%	532.36	0.06	0.0113	1,036.16	0.12	0.0116	2,423.41	0.82	0.0338
30%	532.36	0.06	0.0113	1,036.16	0.12	0.0116	2,423.41	0.74	0.0305
35%	532.36	0.06	0.0113	1,036.16	0.11	0.0106	2,423.41	0.67	0.0276
40%	532.36	0.06	0.0113	1,036.16	0.11	0.0106	2,423.41	0.64	0.0264

Table 19 The average length of the database sequences in the original space and in the new string space produced by DRESS using the top 4 codewords (DRESS-4) and $l = 2$ for the DNA datasets. The % columns show the ratio between the new size and the original size (e.g., in DRESS-4% for $\delta = 5\%$ and the database of $Dataset_{500}^{1000}$, 21.8069% = 163.58 / 750.13 * 100).

	$Dataset_{100}^{500}$			$Dataset_{500}^{1000}$		
δ	original	DRESS-4	DRESS-4%	original	DRESS-4	DRESS-4%
5%	300.21	67.01	22.3210	750.13	163.58	21.8069
10%	300.21	65.07	21.6748	750.13	164.20	21.8895
15%	300.21	66.52	22.1578	750.13	166.83	22.2401
20%	300.21	64.21	21.3884	750.13	166.63	22.0802
25%	300.21	62.87	20.9420	750.13	165.54	22.0682
30%	300.21	62.52	20.8254	750.13	168.58	22.4734

Table 20 The average length of the test sequences in the original space and in the new string space produced by DRESS using the top 4 codewords (DRESS-4) and $l = 2$ for all UniProt datasets. The % columns show the ratio between the new size and the original size (e.g., in DRESS-4% for $Dataset_{401}^{800}$, 5.6711% = 30.18/532.17 * 100). Note that the sizes are independent of δ for the test sequences.

$Dataset_{401}^{800}$			$Dataset_{801}^{1600}$			$Dataset_{1601}^{max}$		
original	DRESS-4	DRESS-4%	original	DRESS-4	DRESS-4%	original	DRESS-4	DRESS-4%
532.17	30.18	5.6711	1,019.56	52.42	5.1414	3,263.87	165.34	5.0658

this method reflects computing the distances between the query and the reference objects, and it is 1.53%, 7.25%, and 18.93% of the brute force cost for the three UniProt datasets, and less than 0.1% for the DNA datasets, respectively. Note that, as mentioned in Section 5.1.3, for all costs reported for this method we have excluded the above embedding cost. The same holds for the runtimes, where only the runtime of the refine step is reported. Finally, the percentage of database objects that were pruned drops from approximately 99% ($\delta = 5\%$) to around (on average) 69% ($\delta = 40\%$) for

Table 21 The average length of the test sequences in the original space and in the new string space produced by DRESS using the top 4 codewords (DRESS-4) and $l = 3$ for all UniProt datasets. The % columns show the ratio between the new size and the original size. Note that the sizes are independent of δ for the test sequences.

$Dataset_{401}^{800}$			$Dataset_{801}^{1600}$			$Dataset_{1601}^{max}$		
original	DRESS-4	DRESS-4%	original	DRESS-4	DRESS-4%	original	DRESS-4	DRESS-4%
532.17	10.73	2.0163	1,019.56	16.38	1.6066	3,263.87	47.90	1.4676

Table 22 The average length of the test sequences in the original space and in the new string space produced by DRESS using the top 4 codewords (DRESS-4) and $l = 4$ for all UniProt datasets. The % columns show the ratio between the new size and the original size. Note that the sizes are independent of δ for the test sequences.

$Dataset_{401}^{800}$			$Dataset_{801}^{1600}$			$Dataset_{1601}^{max}$		
original	DRESS-4	DRESS-4%	original	DRESS-4	DRESS-4%	original	DRESS-4	DRESS-4%
532.17	6.49	1.2195	1,019.56	9.77	0.9583	3,263.87	25.89	0.7932

Table 23 The average length of the test sequences in the original space and in the new string space produced by DRESS using the top 4 codewords (DRESS-4) for the DNA datasets. The % columns show the ratio between the new size and the original size (e.g., in DRESS-4% for $\delta = 5\%$ and the database of $Dataset_{500}^{1000}$, 24.1956% = 171.07/707.03 * 100).

	$Dataset_{100}^{500}$			$Dataset_{500}^{1000}$		
δ	original	DRESS-4	DRESS-4%	original	DRESS-4	DRESS-4%
5%	289.71	72.08	24.8801	707.03	171.07	24.1956
10%	289.61	70.68	24.4052	707.09	171.57	24.2642
15%	289.71	72.08	24.8801	705.93	173.52	24.5803
20%	289.86	70.62	24.3635	706.27	173.43	24.5558
25%	289.56	70.29	24.2748	705.97	172.55	24.4415
30%	289.80	69.64	24.0304	706.75	174.46	24.6848

the UniProt datasets, from 88% ($\delta = 5\%$) to 55% ($\delta = 30\%$) for the DNA $Dataset_{100}^{500}$, and from 93% ($\delta = 5\%$) to 61% ($\delta = 30\%$) for DNA $Dataset_{401}^{800}$.

Table 24 The performance of RBE for all UniProt datasets. For $Dataset_{401}^{800}$ and $Dataset_{801}^{1600}$ 2,000 reference objects were randomly selected, and for $Dataset_{1601}^{max}$ 500 reference objects were used (again randomly chosen). The reported cost does not include any costs outside of the refine step.

	$Dataset_{401}^{800}$			$Dataset_{801}^{1600}$			$Dataset_{1601}^{max}$		
δ	Cost (%)	Pruned (%)	Runtime	Cost (%)	Pruned (%)	Runtime	Cost (%)	Pruned (%)	Runtime
5%	0.3172	99.71	2.77	0.585	99.38	1.43	0.5133	99.38	0.62
10%	3.8266	96.07	13.19	3.6452	96.13	7.03	3.1776	96.17	3.75
15%	7.3660	92.44	23.25	6.6974	92.95	12.68	6.1311	92.71	7.39
20%	11.2688	88.42	34.22	10.2436	89.24	19.24	9.4873	88.87	11.74
25%	15.6807	83.83	40.5	14.3579	84.92	26.81	12.8964	85.05	16.3
30%	20.9975	78.26	60.88	18.7941	80.28	35.05	16.1533	81.47	20.9
35%	27.3194	71.61	78.36	23.9686	74.86	44.56	19.5718	77.70	25.9
40%	34.1353	64.50	96.59	29.1247	69.54	54.05	23.0382	73.89	31.24

6 Conclusions and Future Work

We have proposed a novel method for whole sequence matching in large string databases. The main advantage over state-of-the-art competitors is that it does not require any cumbersome training step and it can handle large query sizes efficiently without loss

Table 25 The performance of RBE for the DNA datasets. For both datasets 500 reference objects were randomly selected. The reported cost does not include any costs outside of the refine step.

	$Dataset_{100}^{500}$			$Dataset_{500}^{1000}$		
δ	Cost (%)	Pruned (%)	Runtime	Cost (%)	Pruned (%)	Runtime
5%	13.0093	88.33	47.35	5.0706	93.06	325.65
10%	21.6184	79.89	76.08	10.8592	85.70	499.42
15%	28.6427	73.00	100.53	14.7995	80.83	636.37
20%	35.1539	66.53	121.28	19.1777	75.52	772.42
25%	41.0073	60.52	140.91	30.9222	62.29	941.61
30%	46.1054	54.91	158.01	31.7492	61.13	1021.47

in accuracy. The experimental results on three protein datasets and two DNA datasets demonstrate that for higher values of search range DRESS produces significantly lower costs and runtimes than the competitors. One price that we pay, compared to the competitors, is the loss of guarantee of 100% recall. At the same time, we believe that this price can be an acceptable trade-off in several domains, given the significant runtime savings that our method achieves.

It will be interesting to explore directions for further improving the performance of our method. One approach may be to implement multiple filter steps, in place of the single filter step that our method uses. These filter steps can be applied in sequence, so that each filter step is applied only on the candidates selected by the previous step, and each filter step does somewhat more work (e.g., by using more codewords) than the previous step, so as to prune some more candidates. Finally, we would like to study the performance of DRESS on sequences from other domains, such as text.

Acknowledgements This work was partially supported by National Science Foundation grants IIS-1055062, CNS-1059235, CNS-1035913, and CNS-1338118. The work of Gautam Das was partially supported by National Science Foundation under grants 0812601, 0915834, 1018865 and grants from Microsoft Research.

References

Altschul S, Madden T, Schffer R, JZhang, ZZhang, WMiller, Lipman D (1997) Gapped blast and psi-blast: a new generation of protein database search programs. Nucleic Acids Res 25:3389–3402

Altschul SF, Gish W, Miller W, Myers EW, Lipman DJ (1990) Basic local alignment search tool. Journal of molecular biology 215(3):403–410

Arasu A, Ganti V, Kaushik R (2006) Efficient exact set-similarity joins. In: Proceedings of Very Large Database Endowment (PVLDB), pp 918–929

Baeza-Yates R, Gonnet GH (1992) A new approach to text searching. Communications of the ACM 35(10):74–82

Behm A, Vernica R, Alsubaiee S, Ji S, Lu J, Jin L, Lu Y, Li C (2010) UCI Flamingo Package 4.0. `http://flamingo.ics.uci.edu/releases/4.0/`

Bhadra R, Sandhya S, Abhinandan KR, Chakrabarti S, Sowdhamini R, Srinivasan N (2006) Cascade psi-blast web server: a remote homology search tool for relating protein domains. Nucleic Acids Research 34(Web-Server-Issue):143–146

Burrows M, Wheeler DJ (1994) A block-sorting lossless data compression algorithm. Tech. Rep. 124, Systems Research Center, Palo Alto, URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.37.6774`

Hjaltason G, Samet H (2003) Properties of embedding methods for similarity searching in metric spaces. IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI) 25(5):530–549

Jongeneel CV (2000) Searching the expressed sequence tag (est) databases: Panning for genes. Bioinformatics 1:76–92

Kalafus KJ, Jackson AR, Milosavljevic A (2004) Pash: Efficient genome-scale sequence anchoring by positional hashing. Genome Resources 14(4):672678

Kent WJ (2002) Resource BLAT-The BLAST-Like Alignment Tool. Genome Research

Kim MS, Whang KY, Lee JG, Lee MJ (2005a) n-gram/2l: A space and time efficient two-level n-gram inverted index structure. In: Proceedings of the 31st international conference on Very large data bases, VLDB Endowment, pp 325–336

Kim YJ, Boyd A, Athey BD, Patel JM (2005b) miblast: scalable evaluation of a batch of nucleotide sequence queries with blast. Nucleic Acids Res 33:4335–4344

Korf I, Gish W (2000) Mpblast : improved blast performance with multiplexed queries. Bioinformatics 16:1052–1053

Langmead B, Trapnell C, Pop M, Salzberg SL, et al (2009) Ultrafast and memory-efficient alignment of short dna sequences to the human genome. Genome Biol 10(3):R25

Li C, Wang B, Yang X (2007) Vgram: Improving performance of approximate queries on string collections using variable-length grams. In: Proceedings of the 33rd international conference on Very large data bases, VLDB Endowment, pp 303–314

Li C, Lu J, Lu Y (2008a) Efficient merging and filtering algorithms for approximate string searches. International Conference on data Engineering (ICDE)

Li H, Ruan J, Durbin R (2008b) Mapping short dna sequencing reads and calling variants using mapping quality scores. Genome research 18(11):1851–1858

Li R, Li Y, Kristiansen K, Wang J (2008c) Soap: short oligonucleotide alignment program. Bioinformatics 24(5):713–714

Li Y, Patel JM, Terrell A (2012) Wham: a high-throughput sequence alignment method. ACM Transactions on Database Systems (TODS) 37(4):28

Litwin W, Mokadem R, Rigaux P, Schwarz T (2007) Fast ngram-based string search over data encoded using algebraic signatures. In: Proceedings of the Very Large Database Endowment (PVLDB), pp 207–218

Liu B, Wang X, Zou Q, Dong Q, Chen Q (2013) Protein remote homology detection by combining chous pseudo amino acid composition and profile-based protein representation. Molecular Informatics 32(9-10):775–782

Meek C, Patel JM, Kasetty S (2003) Oasis: An online and accurate technique for local-alignment searches on biological sequences. In: Proceedings of Very Large Database Endowment (PVLDB), vol 29, pp 910–921

Needleman SB, Wunsch CD (1970) A general method applicable to the search for similarities in the amino acid sequence of two proteins. Journal of molecular biology 48(3):443–453

Ning Z, Cox AJ, Mullikin JC (2001) SSAHA: A fast search method for large dna databases. Genome Resources 11(10):1725–1729

Papapetrou P, Athitsos V, Kollios G, Gunopulos D (2009) Reference-based alignment in large sequence databases. Proceedings of the Very Large Database Endowment (PVLDB) 2(1):205–216

Smith TF, Waterman MS (1981) Identification of common molecular subsequences. Journal of molecular biology 147(1):195–197

Tian Y, Mceachin RC, Santos C, States DJ, Patel JM (2007) Saga: A subgraph matching tool for biological graphs. Bioinformatics 23(2):232–239

Traina C, Traina AJM, Seeger B, Faloutsos C (2000) Slim-trees: High performance metric trees minimizing overlap between nodes. International Conference on Extending Database Technology (EDBT) pp 51–65

Venkateswaran J, Lachwani D, Kahveci T, Jermaine C (2006) Reference-based indexing of sequence databases. In: International Conference on Very Large Databases (VLDB), pp 906–917

Vergoulis T, Dalamagas T, Sacharidis D, Sellis TK (2012) Approximate regional sequence matching for genomic databases. VLDB J 21(6):779–795

Vieira MR, Traina C, Chino FJT, Traina AJM (2004) Dbm-tree: A dynamic metric access method sensitive to local density data. Brazilian Symposium on Databases (SBBD) pp 163–177

Wandelt S, Starlinger J, Bux M, Leser U (2013) Rcsi: Scalable similarity search in thousand(s) of genomes. Proceedings of the VLDB Endowment (PVLDB) p To appear

Wu S, Manber U (1992) Fast text searching: allowing errors. Communications of the ACM 35(10):83–91

Yan X, Yu PS, Han J (2005) Graph indexing based on discriminative frequent structure analysis. ACM Trans Database Syst 30(4):960–993

Yang X, Wang B, Li C (2008) Cost-based variable-length-gram selection for string collections to support approximate queries efficiently. In: Proceedings of the 2008 ACM SIGMOD international conference on Management of data, ACM, pp 353–364

Zhang Z, Schwartz S, Wagner L, Miller W (2000) A greedy algorithm for aligning dna sequences. Journal of Computational Biology 7:203–214

Zhu H, Kollios G, Athitsos V (2012) A generic framework for efficient and effective subsequence retrieval. Proceedings of the VLDB Endowment (PVLDB) 5(11):1579–1590