Better Greedy Sequence Clustering with Fast Banded Alignment*†

Brian Brubach[1], Jay Ghurye[2], Mihai Pop[3], and Aravind Srinivasan[4]

1 **Department of Computer Science, University of Maryland, College Park, USA**
bbrubach@cs.umd.edu
2 **Department of Computer Science, University of Maryland, College Park, USA**
jayg@cs.umd.edu
3 **Department of Computer Science, University of Maryland, College Park, USA**
mpop@umiacs.umd.edu
4 **Department of Computer Science, University of Maryland, College Park, USA**
srin@cs.umd.edu

Abstract

Comparing a string to a large set of sequences is a key subroutine in greedy heuristics for clustering genomic data. Clustering 16S rRNA gene sequences into operational taxonomic units (OTUs) is a common method used in studying microbial communities. We present a new approach to greedy clustering using a trie-like data structure and Four Russians speedup. We evaluate the running time of our method in terms of the number of comparisons it makes during clustering and show in experimental results that the number of comparisons grows linearly with the size of the dataset as opposed to the quadratic running time of other methods. We compare the clusters output by our method to the popular greedy clustering tool UCLUST. We show that the clusters we generate can be both tighter and larger.

1998 ACM Subject Classification B.2.4 Algorithms

Keywords and phrases Sequence Clustering, Metagenomics, String Algorithms

Digital Object Identifier 10.4230/LIPIcs.WABI.2017.3

1 Introduction

The problem of comparing a string against a large set of sequences is of central importance in domains such as computational biology, information retrieval, and databases. Solving this problem is a key subroutine in many *greedy* clustering heuristics, wherein we iteratively choose a cluster center and form a cluster by recruiting all strings which are similar to the center. In computational biology, sequence similarity search is used to group biological sequences that are closely related. We will use this domain as a motivating example throughout the paper.

Traditionally, clustering 16S rRNA gene [11] sequences involved building a multiple sequence alignment of all sequences, computing a pairwise distance matrix of sequences based on the multiple sequence alignment, and clustering this matrix [17]. However, finding the best multiple sequence alignment is computationally intractable and belongs to the class of

* AS and BB were supported in part by NSF Awards CNS 1010789 and CCF 1422569. MP and BB were supported in part by the NIH, grant R01-AI-100947 to MP. MP and JG were supported in part by the Bill and Melinda Gates Foundation (PI Jim Nataro, subcontract to MP). AS was supported in part by a research award from Adobe, Inc.
† The authors wish to thank the anonymous reviewers for their helpful comments.

NP-hard problems[16]. Another naive way of clustering sequences is to perform all-versus-all comparison to compute a similarity metric such as edit distance and perform hierarchical clustering to merge closely related sequences together. However, the resulting running time is at least quadratic in the total number of sequences. With the development of faster and cheaper DNA sequencing technologies, metagenomic sequencing datasets can contain over 1 billion short reads [2]. At this scale, both strategies can prove to be very expensive and take months to generate clusters. To counter this, heuristic-based methods like greedy clustering are used. While these methods can have worst case quadratic running time, they can run faster in practice[8, 3, 5].

Here, we show a new method for reducing that worst case quadratic running time in practice when the distance metric is the Levenshtein distance [7] and similarity is determined by a maximum distance of d. Our algorithm improves the speed of the recruitment step wherein we seek all strings within d distance of a chosen center. In addition to promising experimental results, we give slightly weaker, but provable, guarantees for our techniques while many existing methods do not. Finally, we analyze the quality of the clusters output by our method in comparison to the popular greedy clustering tool UCLUST. We show that the clusters we generate can be both tighter and larger.

1.1 Related Work

The problem of comparing a query string against a large string database has been widely studied for at least the past twenty years. For similarity metrics like the edit distance, a dynamic programming algorithm [14] can be used to compare two sequences in $O(m^2)$ time, where m is the length of the sequences. When we only wish to identify strings which are at most edit distance d apart, the running time for each comparison can be reduced to $O(md)$ [12] using a modified version of the standard dynamic programming algorithm. This type of sequence alignment is referred to as *banded alignment* in the literature since we only need to consider a diagonal "band" through the dynamic programming table. The simple dynamic programming approach can also be sped up by using the Four Russians method [10, 9], which divides the alignment matrix into small square blocks and uses a lookup table to perform the alignment quickly within each block. This brings the running time down to $O(m^2 \log(\log(m))/\log(m))$ and $O(m^2/\log m)$ for arbitrary and finite alphabets, respectively. Myers [13] considered the similar problem of finding all locations at which a query string of length m matches a substring of a text of length M with at most d differences. They used the bit vector parallelism in hardware to achieve a running time of $O(mM/w)$ where w is the machine word size. However, when used for clustering sequences, these methods need to perform pairwise comparison of all sequences, thereby incurring the high computational cost of $O(n^2)$ comparisons where n is the total number of sequences.

Sequence search against a database is a crucial subroutine in sequence clustering in general and greedy clustering in particular. In greedy approaches, we choose some sequences to be cluster centers and clusters are formed by recruiting other sequences which are similar to the centers. Depending on the approach, we may compare a sequence to recruit against a set of cluster centers or compare a single cluster center against all other sequences, recruiting those within some specified distance. The comparison can be done using any of the methods mentioned above, but in the worst case, existing algorithms may need to perform all pairs banded alignment resulting in $O(n^2md)$ running time on arbitrary input data. However, the interesting property of sequencing data is that most of the sequences generated by the experiments are highly similar to each other. To exploit sequence similarity and reduce the computation performed in dynamic programming, the DNACLUST [5] algorithm

lexicographically sorts the sequences and compares sequences against the center sequence in sorted order. Since the adjacent sequences in sorted order share a long prefix, the part of the dynamic programming table corresponding to their longest common prefix remains unchanged, allowing the "free" reuse of that part of the table for further alignments. This method fails when two sequences differ at the start but are otherwise similar. In this case, the entire dynamic programming table needs to be recomputed. The UCLUST [3] algorithm uses the USEARCH [3] algorithm to compare a query sequence against a database of cluster centers. However, the algorithm used by UCLUST makes several heuristic choices in order to speed up the calculation of clusters and thus, the resulting clusters are not guaranteed to satisfy any specific requirement. For example, the distances between sequences assigned to the same cluster should ideally satisfy triangle inequality (ensuring that the cluster diameter is at most twice the radius) and the cluster diameters should be both fairly uniform and within the bounds specified by the user.

1.2 Preliminaries

Let the multiset $\mathcal{S}$ be the set of n sequences to be clustered. Let m be the maximum length of any sequence in $\mathcal{S}$. For simplicity of exposition and analysis, we will assume throughout most of this paper that all sequences have length exactly m. We also assume m is much smaller than n.

1.2.1 Distance metric

We use the same edit distance-based similarity metric as DNACLUST[5], namely

$$\text{similarity} = 1 - \frac{\text{edit distance}}{\text{length of the shorter sequence}}$$

Here, we define edit distance to be Levenshtein distance with uniform penalties for insertions, deletions, and substitutions. The "length of the shorter sequence" refers to the original sequences' lengths without considering gaps inserted by the alignment. We say that two sequences are *similar* if their alignment meets or exceeds a given similarity threshold. Let d be the maximum edit distance between two sequences aligned to the same cluster. This distance is usually computed from a similarity threshold provided by the user, e.g., 97%. Both of our algorithms will be performing *banded alignment* with d as the maximum allowable distance. In this case, if we determine that two sequences have distance greater than d, we need not report their actual distance.

1.2.2 Intervals

Our algorithm involves dividing each sequence into overlapping substrings of length k at regular *intervals*. We formalize the definition of an interval as follows. Given a *period length* p such that $k = p + d + 1$, we divide each sequence into $\lfloor m/p \rfloor$ intervals of length k. For $i \in \{0, 1, \ldots, \lfloor m/p \rfloor - 1\}$, the i^{th} interval starts at index ip inclusive and extends to index $ip + k$ exclusive. We will see in Section 2.1.2 that we must choose p to be at least d. However, choosing a larger p may give a better speedup when dealing with highly similar sequences. Further, for an interval i, we define b_i to be the number of *distinct* substrings for interval i over all sequences in $\mathcal{S}$ and we define $b = \max_i b_i$. We will show in Section 2.1.3 that when b is much smaller than n we get some theoretical improvement on the running time. Figure 1 shows an example of how a sequence is partitioned into a set of overlapping substrings. We

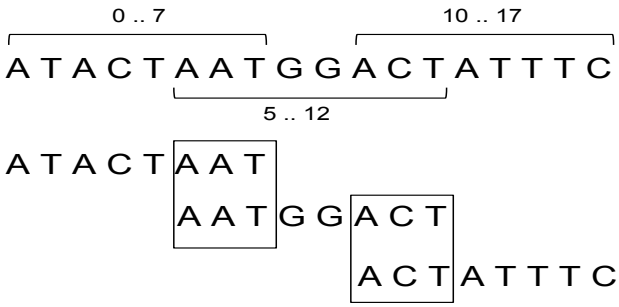

Figure 1 An example of how a string is divided in overlapping substrings called intervals. In this case, the length of each substring (k) is 8. Since the substrings must overlap by $d + 1$ characters, which in this case is 3, the period length (p) is 5.

store these intervals in a data structure we call an *Edit Distance Interval Trie (EDIT)* which is described in detail in Section 2.2.

1.2.3 Greedy clustering

The greedy clustering approach (similar to CD-HIT[8], UCLUST, and DNACLUST) can be described at a high level as follows. De-replicate the multiset $\mathcal{S}$ to get the set $\mathcal{U}$ of distinct sequences. Optionally, impose some ordering on $\mathcal{U}$. Then, iteratively remove the top sequence from $\mathcal{U}$ to form a new cluster center s_c. Recruit all sequences $s \in \mathcal{U}$ that are within d distance from s_c. When we recruit a sequence s, we remove it from $\mathcal{U}$ and add it to the cluster centered at s_c. If s_c does not recruit any sequences, we call it a singleton and add it to a list of singletons, rather than clusters. We continue this process until $\mathcal{U}$ is empty.

We order the sequences of $\mathcal{U}$ in decreasing order of their abundance/multiplicity in $\mathcal{S}$. This is also the default ordering used by UCLUST. Alternatively, DNACLUST uses decreasing order of sequence length. The reason for ordering by abundance is that assuming a random error model, the abundant sequences should be more likely to be "true" centers of a cluster. The reason for DNACLUST ordering by length is to preserve triangle inequality among sequences in a cluster when performing semi-global alignment allowing gaps at the end with no penalty. Semi-global alignment is necessary for comparing reads generated by specific sequencing technologies such as 454. However, since we perform global alignment, triangle inequality is guaranteed regardless of the ordering and thus, ordering by abundance is preferred.

1.3 Our Contributions

We developed a method for recruiting in exact greedy clustering inspired by the classical Four Russians speedup. In Section 2, we describe our algorithm and prove that the worst case theoretical running time is better than naive all-versus-all banded alignment under realistic assumptions on the sequencing data used for clustering. In section 3, we present experimental results from using our method to cluster a real 16S rRNA gene dataset containing about 2 million distinct sequences. We show that on real data the asymptotic running time of the algorithm grows linearly with the size of the input data. We also evaluated the quality of the clusters generated by our method and compared it with UCLUST, which is one of the widely used methods. We show that our method generates tighter and larger clusters at 99% similarity both when considering edit distance and evolutionary distance. At 97%

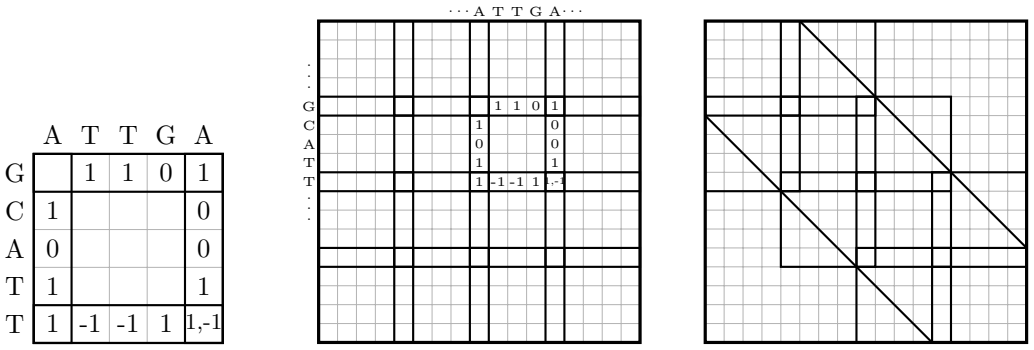

Figure 2 Example of classic Four Russians. **Left:** a single block. Notice that for any input in the upper left corner, we can sum that value with one path along the edges of the block to recover the value in the lower right corner. Note that the offset value in the lower right corner may be different for the row and column vectors overlapping at that cell. In this case, the lower right cell is one more than its left neighbor and one less than its above neighbor. **Center:** the full dynamic programming table divided into nine 5×5 blocks. Note that the offset values in the example block may not correspond to the optimal alignment of the two substrings shown since they depend on the global alignment between the two full length strings. **Right:** blocks covering only the diagonal band in the context of banded alignment.

similarity, we show that the our method produces clusters with a much tighter edit distance diameter compared to UCLUST. While UCLUST runs faster at similarities 97% and less, our approach is faster at higher similarities. In particular, we highlight that UCLUST does not scale linearly at the 99% similarity threshold while our approach does.

2 Recruiting algorithm

We show two ways in which the classical Four Russians speedup can be adapted to banded alignment. Then, we describe a trie-like data structure for storing sequences. Finally, we use this data structure to recruit similar sequences to a given center sequence using our Four Russians method.

2.1 Banded Four Russians approach

We present two ways to extend the Four Russians speedup of edit distance computation to banded alignment. The first is a very natural extension of the classical Four Russians speedup. The second is useful for tailoring our algorithm to meet the needs of 16S rRNA gene clustering. Specifically, we exploit the fact that the strings are similar and the maximum edit distance is small.

2.1.1 Warm-up: classic Four Russians speedup

In the classical Four Russians speedup of edit distance computation due to [10, 9], the dynamic programming table is broken up into square *blocks* as shown in the center of Fig. 2. These blocks are tiled such that they overlap by one column/row on each side (for a thorough description of this technique see [6]). When computing banded alignment, we only need to tile the area within the band as in the righthand of Fig. 2. Let the maximum edit distance be d and the string lengths be m. Then our block size k can be as small as $d + 1$ and we require roughly $2m/k$ blocks in total.

The high level idea of the Four Russians speedup is to precompute all possible solutions to a *block function* and store them in a lookup table (In our implementation we use lazy computation and store the lookups in a hash table instead of precomputing for all inputs). The block function takes as input the two substrings to be compared in that block and the first row and column of the block itself in the dynamic programming table. It outputs the last row and column of the block. We can see in the Fig. 2 that given the two strings and the first row and column of the table, such a function could be applied repeatedly to compute the lower right cell of the table and therefore, the edit distance. Note that cells outside the band will not be used since any alignment visiting those cells must have distance larger than d.

There are several tricks that reduce the number of inputs to the block function to bound the time and space requirements of the lookup table. For example, the input row and column for each block can be reduced to vectors in $\{-1, 0, 1\}^d$. These *offset vectors* encode only the difference between one cell and the next (see Fig. 2) which is known to be at most 1 in the edit distance table. It has also been shown that the upper left corner does not need to be included in the offset vectors. This bounds the number of possible row and column inputs at 3^d each [10].

Notice that for the banded alignment problem, this may not provide any speedup for comparing just two strings of length m. Indeed, building and querying the lookup table may take more time than simply running the classical dynamic programming algorithm restricted to the band of width $2d + 1$. However, our final algorithm will do many such comparisons between different pairs of strings using the same lookup table. In practice, we also populate the lookup table as needed rather than pre-computing it. This technique, known as *lazy computation*, allows us to avoid adding unnecessary entries for comparisons that don't appear in our dataset. Additionally, decomposing sequences into blocks will be a crucial step in building the data structure in Section 2.2.

2.1.2 Our approach to the Four Russians speedup

Notice that the previous approach will not offer much benefit in practice when d is small (e.g. $d = 2$). The overhead of looking up block functions and stitching them together may even be slower than simply running dynamic programming on a block. Further, our dataset may not require us to build a lookup table comparing all possible strings of length k.

Here we consider a different block function. This function is designed for situations in which we wish to use a block size k that is larger than $d + 1$. The blocks now overlap on a square of size $d + 1$ at the upper left and lower right corners. We will call these overlapping regions *overlap squares*. Our block function now takes as input the two substrings to be compared and the first row and column of the the upper left overlap square. It outputs the first row and column of the lower right overlap square as well as the difference between the upper left corners of the two overlap squares.

Thus, we can move directly from one block to the next, storing a sum of the differences between the upper left corners. In this case, reaching the final lower right cell of the table requires an additional $O(d^2)$ operation to fill in the last overlap square, but this adds only a negligible factor to the running time.

This approach succeeds when the number of possible substring inputs to the block function is limited by some properties of the dataset as opposed to an absolute theoretical upper bound such as $O(|\sigma|^k)$ based on the number of possible strings of length k for an alphabet σ. Rather than computing and storing all possible inputs, we simply store the inputs encountered by our algorithm. The advantage is that a larger block size reduces the number of lookups needed to compare two strings which is $m/(k - d - 1)$ for this approach. Naturally, the same

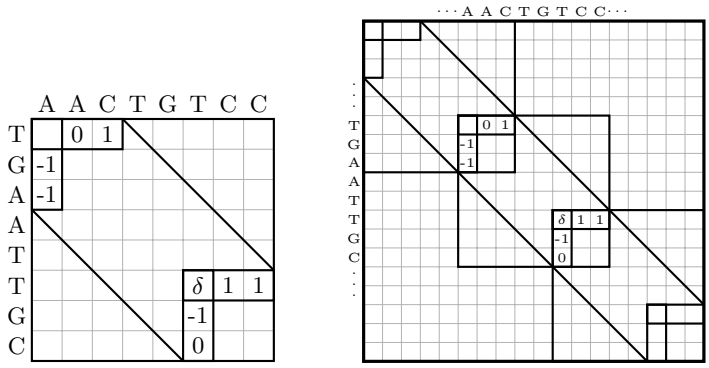

Figure 3 Example of our approach to the Four Russians speedup. **Left:** a block for maximum edit distance $d = 2$. The output δ represents the offset from the upper left corner of the current block to the upper left corner of the next block. Note that we only need to consider a diagonal band of the block itself. **Right:** using these blocks to cover the diagonal band of the dynamic programming table in the context of banded alignment.

tricks such as offset encoding of the input rows and columns as some vector in $\{-1, 0, 1\}^d$ can be applied in this case.

Another benefit of this approach is that it is more straightforward to implement in practice. Each block depends on the full output of one previous block. In contrast, the classical approach requires combining partial input from two previous blocks and also sending output to two separate blocks.

2.1.3 Theoretical bound on the running time of our approach

To give some intuition, we prove a theoretical bound on the running time under the assumption of at most b distinct substrings per interval in the dataset. This is a reasonable assumption for certain application in computational biology. For example, the 16s rRNA gene is highly conserved and thus b is much smaller than n for such datasets. While standard banded alignment takes $O(n^2md)$, we show that for small enough b this can be reduced to $O(n^2m)$. We prove this bound for our approach to using the Four Russians speedup for banded alignment, but it extends to the classical approach as well.

▶ **Theorem 1.** *If $b \leq \frac{n}{3^d\sqrt{d}}$, we can find all pairs of distance at most d in $O(n^2m)$ time.*

Proof. To simplify, we will assume the lookup table is pre-computed. Then, we must show that if $b \leq \frac{n}{3^d\sqrt{d}}$, then building the lookup table and doing the actual string comparisons can each be done in $O(n^2m)$ time. We further assume $k \approx 2d$ (in practice we choose a larger k).

First, we show that there are at most $\frac{m}{k-d}b^2 3^{2d}$ entries in the lookup table. There are at most $\frac{m}{k-d}$ intervals and since each interval has at most b distinct strings, there are at most b^2 relevant string comparisons. Each distinct string comparison must be computed for all 3^{2d} offset vector inputs. The cost of generating each lookup entry is simply the cost of computing banded alignment on a block, kd. Thus, the lookup table can be built in time $\frac{m}{k-d}b^2 3^{2d}kd$. Keeping our goal in mind we see that

$$\frac{m}{k-d}b^2 3^{2d}kd \leq n^2m \qquad \text{is true when} \qquad b \leq \frac{n}{3^d\sqrt{d}} \qquad \text{since } k \approx 2d$$

To bound the running time of the string comparisons, notice that comparing two strings requires computing $\frac{m}{k-d}$ block functions. The time spent at each block will be $O(k + d)$ to

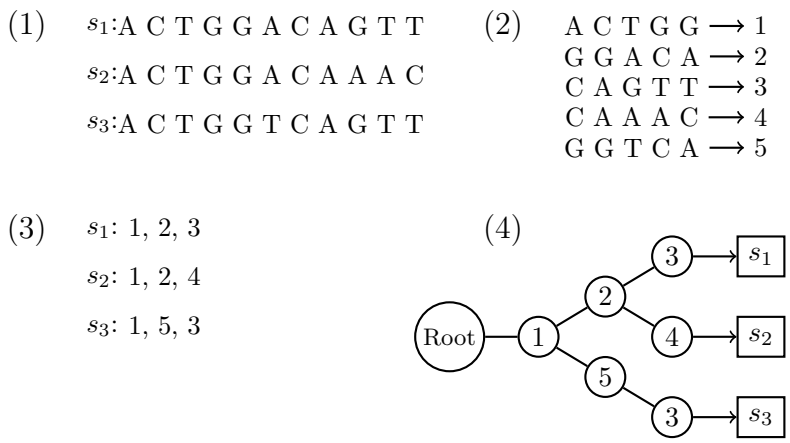

Figure 4 Example illustrating the steps of Algorithm 1 with $d = 1$ and $k = 5$.

look up the output of the block function and update our sum for the next corner. Thus, building the lookup table and computing the edit distance between all pairs using the lookup table each take $O(n^2m)$ time. ◀

2.2 The Edit Distance Interval Trie (EDIT)

To facilitate the crucial step of identifying all strings within edit distance d of a chosen cluster center, we construct a trie-like data structure on the intervals. This structure will be built during a pre-processing stage. Then, during recruitment, any recruited sequences will be deleted from the structure. The procedure for building this structure is summarized in Algorithm 1 and illustrated in Figure 4. The main benefit of this data structure, like any trie, is that it exploits prefix similarity to avoid duplicating work.

The mapping in step 2 of Algorithm 1 is a one-to-one mapping to integers from one to the number of distinct substrings. Here, it serves to reduce the size of the data structure since the number of distinct substrings will typically be much less than all possible length k strings on the given alphabet. This mapping also speeds up calls to the lookup table during the recruitment subroutine summarized in the next section.

Algorithm 1: BUILD-EDIT

1. Partition each sequence into overlapping intervals of length k, such that each interval overlaps on exactly $d + 1$ characters.
2. Map each distinct substring of length k appearing in our list of interval strings to an integer.
3. Assign an integer vector *signature* to each sequence by replacing each block with its corresponding integer value.
4. Insert these signatures into a trie with the leaves being pointers to the original sequences.

2.3 Recruiting to a center

Given a center sequence s_c, we can recruit all sequences of distance at most d from s_c by simply traversing the trie in depth first search order and querying the block function of each

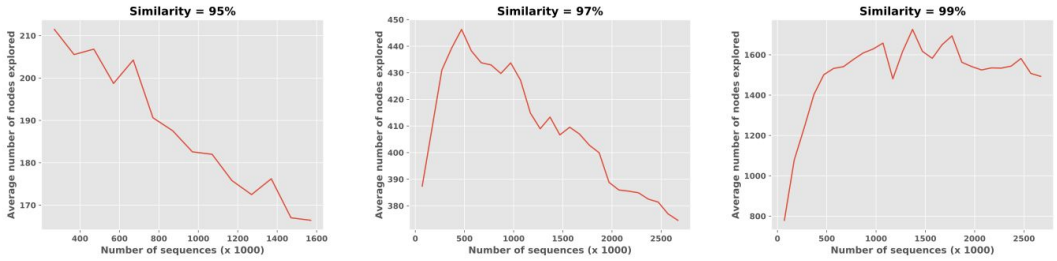

Figure 5 Plots for the average number of nodes explored in the tree while recruiting sequences to a cluster center.

node we encounter. The input to each block function is the substring of that node in the trie, the substring at the same depth within the signature of s_c, and the offset vectors output by the previous block function. As we traverse a path from the root toward a leaf, we store a sum of the edit distance as provided by the output of each block function. If this sum ever exceeds the maximum distance d, we stop exploring that path and backtrack. Whenever we reach a leaf ℓ, we retrieve its corresponding sequence s_ℓ. Then, we align the remaining suffixes and compute the true similarity threshold $d' \leq d$ based on the length of the shorter sequence. If the final edit distance is less than d', we add s_ℓ to the cluster centered at s_c and prune/remove any nodes in the the trie corresponding only to s_ℓ.

3 Experimental results

3.1 Properties of our recruitment algorithm and data structure

In this section, we highlight some key features of our recruitment algorithm and the EDIT data structure. To evaluate our method, we used a dataset consisting of about 57 million 16S rRNA amplicon sequencing reads with 2.7 million distinct sequences. To understand the impact of the number of input sequences to cluster on the average number of comparisons in each recruitment step, we ran our algorithm on different input sizes at different similarity thresholds. We counted the average number of tree nodes explored while recruiting a particular center sequence and used it as a quantitative representation of the amount of comparisons made since all nodes represent a substring of fixed length k. Figure 5 shows the plots for the average number of tree nodes explored for different similarity thresholds. For the 95% and 97% similarity thresholds, the average number nodes explored decreases as more sequences are clustered. This happens because of the fact that although more sequences are clustered, due to the lower similarity threshold a large number of sequences get clustered in each traversal of the tree. For 99% similarity threshold, the average number of nodes explored increases initially with the number of sequences, but becomes uniform after about $100,000$ sequences. The strict increase in the number of nodes can be explained by the high similarity threshold. However, in all cases, the number of nodes explored by each center does not increase linearly with the number of input sequences. Thus the total number of comparisons made for given dataset is observed to be increasing as function of n rather than the worst case n^2.

To understand the likelihood of backtracking at each level of the tree, we clustered a sample of 1.07 million distinct sequences at three different similarity thresholds (95%, 97%, and 99%). The backtracking probability for a given node was calculated as the ratio of the number of times we stopped exploring a path at that node to the total number of times

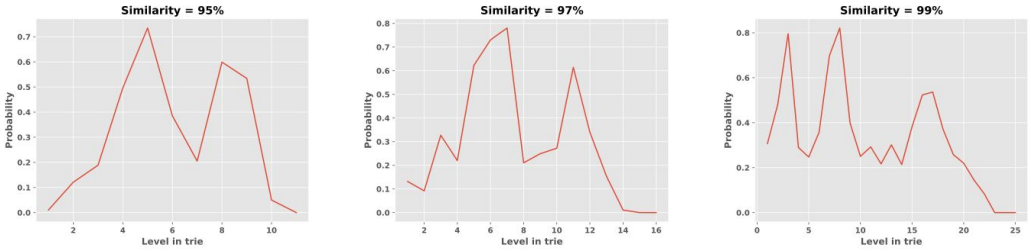

Figure 6 Plots for the probability of backtracking at a particular level in the tree. Note that the number of levels is different for different similarity thresholds since our substring length k is dependent on the maximum distance d.

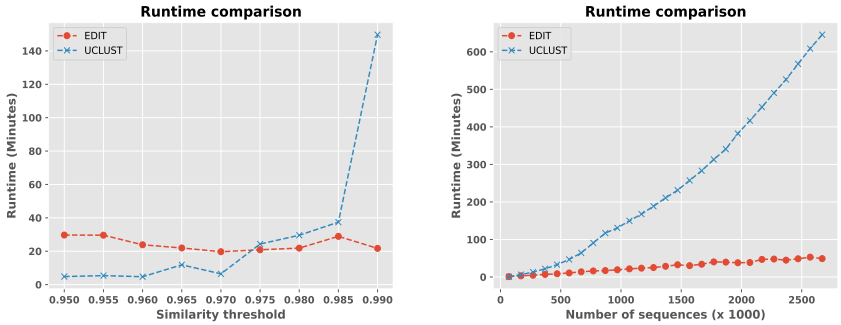

Figure 7 Running time comparison of EDIT and UCLUST as a function of similarity threshold and number of sequences.

that node was explored. We aggregated this probability for all of the nodes belonging to the same level of the tree. Figure 6 shows this likelihood for all levels of the tree at the different similarity thresholds. As we define block size based on the similarity, there are a different number of levels in the tree corresponding to different similarity thresholds. For all three similarity thresholds, the probability of backtracking decreases as we go deeper into the tree except a couple of sharp peaks at intermediate levels. These peaks can be attributed to the sequencing artifacts. The 16S rRNA gene reads are sequenced using the Illumina paired-end sequencing protocol. These paired reads are then merged to make a single read which we use for clustering. The reads contain sequencing errors concentrated near their ends. Due to such sudden errors along the sequence, while recruiting, the edit distance can easily go above the threshold and backtracking needs to be performed. Since we're using lazy computation, our first encounter with a particular input to the block function requires us to explicitly perform the dynamic programming for that block and store it in the lookup table. However, we observed that this explicit dynamic programming computation is rare and most block functions can be computed by simply querying the lookup table (data not shown).

3.2 Comparison with UCLUST

Here, we evaluate EDIT against UCLUST, a highly used tool for analyzing 16S rRNA gene datasets.

3.2.1 Running time analysis

We compared the running time of EDIT and UCLUST on a subsample 1.07 million distinct sequences at different similarity thresholds. Figure 7 shows the plot for running time at

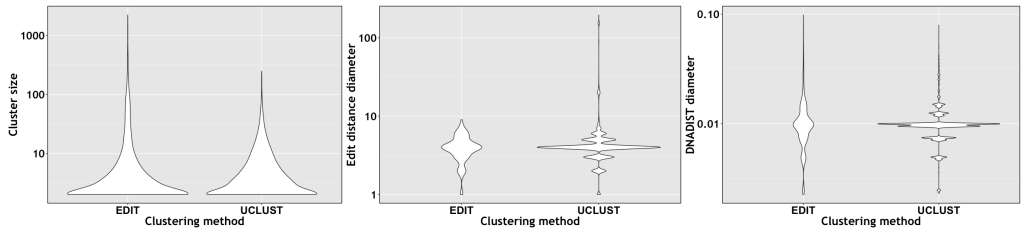

Figure 8 Evaluation at similarity threshold of 99%. All of the plots are log scaled.

Figure 9 Evaluation at similarity threshold of 97%. All of the plots are log scaled.

different similarity thresholds. We observed that the running time of EDIT stays fairly constant at different similarity thresholds whereas the running time of UCLUST was very low for lower similarity thresholds, but increased non-linearly at higher similarity thresholds. Especially, between 98.5% to 99%, the running time of UCLUST grows 5 folds. We did further analysis of running time at 99% similarity threshold using different sample sizes as input. Figure 7 shows the running time comparison of UCLUST and EDIT. It can be observed that, the running time of UCLUST on large sample sizes ($>$ 1 million) grows much faster that the running time of EDIT, which scales almost linearly. For the largest sample of 2.7 million sequences, UCLUST running time was ten times greater than EDIT running time. This evaluation implies that higher similarity thresholds ($>$ 98%), EDIT was faster compared to UCLUST. Also, the running time of EDIT showed low variance compared to UCLUST for different similarity thresholds.

3.2.2 Evaluation of clusters

We subsampled 135,880 distinct sequences from the entire dataset and ran both methods at the 97% and 99% similarity thresholds. We then compared the outputs of both methods using three metrics: the cluster size, the cluster diameter based on the sequence similarity, and the cluster diameter based on the evolutionary distance. To compute the cluster diameter based on sequence similarity, we computed the maximum edit distance between any two sequences in each cluster. To compute the cluster diameter based on evolutionary distance, we first performed a multiple sequence alignment of the sequences in each cluster using clustalW [15]. Once the multiple sequence alignment was computed, we used the DNADIST program from the phylip [4] package to compute a pairwise evolutionary distance matrix. The maximum distance between any pair of sequences is defined as the DNADIST diameter. Using two orthogonal notions of cluster diameter helps to define the "tightness" of clusters. Figures 8 and 9 show violin plots for different comparison metrics at the 99% and 97% similarity thresholds, respectively. At the 99% similarity threshold, EDIT is able to produce larger clusters compared to UCLUST. The edit distance diameters for the clusters generated by EDIT is fairly well constrained. However, the edit distance diameters for the clusters

generated by UCLUST had a large variance, implying that several dissimilar sequences may be getting clustered together. The DNADIST diameter for both methods was comparable. At the 97% similarity threshold, both EDIT and UCLUST generated similar sized clusters. Even in this case, the edit distance diameter for UCLUST clusters showed a larger variance compared to the edit distance diameter for EDIT clusters. The DNADIST diameter for UCLUST has slightly more variance compared to that of EDIT clusters, implying some of the clusters generated by UCLUST had sequences with a large evolutionary distance between them. This validation confirms that the sequences in the clusters produced by EDIT at different similarity thresholds are highly similar to each other.

At the 99% similarity threshold, we observed a stark difference between the cluster sizes of EDIT and UCLUST. For example, the two largest clusters produced by EDIT had sizes 7,978 and 3,383 respectively whereas the two largest clusters produced by UCLUST were of sizes 249 and 233, which is almost 30 times smaller than the largest EDIT cluster. To investigate this further, we used BLAST[1] to align all clusters of UCLUST against the top two largest clusters of EDIT. We only considered the alignments with 100% alignment identity and alignment coverage. We observed that 765 distinct UCLUST clusters had all of their sequences aligned to the largest EDIT cluster and 837 distinct clusters had at least 80% of their sequences aligned to the largest EDIT cluster. Only 82 UCLUST clusters out of 16,968 total (not including singletons) had less than 80% of their sequences mapped to the largest EDIT cluster. Those 82 clusters accounted for only 255 sequences, roughly 30 times fewer than the number of the sequences in the largest EDIT cluster alone. As far as singletons (the clusters with only one sequence) are concerned, EDIT generated 22,318 singleton clusters whereas UCLUST generated 33,519 singleton clusters. For the size of the sample considered in this analysis, this difference is very significant. This evaluation implies that at a high similarity threshold, heuristic based methods like UCLUST tend to produce fragmented clusters whereas EDIT was able to capture a higher number of similar sequences in a single cluster.

4 Conclusion and future directions

The datasets analyzed by biologists are rapidly increasing in size due to the advancements in sequencing technologies and efficient clustering are needed to analyze these datasets in reasonable memory and running time. In this paper, we proposed a first step towards this goal by designing a novel data structure to perform banded sequence alignment. We extended the traditional Four Russian's method to perform banded alignment of highly similar sequences and use that to perform greedy clustering of 16S rRNA amplicon sequencing reads. We compared our method to UCLUST and showed that our method generates tight clusters at different similarity thresholds when both string similarity and evolutionary distance are considered. We focused our discussion of results around high similarity clustering ($> 97\%$) because there is no fixed threshold that can create biologically meaningful clusters. Our method can generate mathematically well-defined and tight clusters, which can serve as representative clusters from the original data and thus can be used to perform downstream computationally intensive analysis.

Although we use clustering as a motivating example throughout the paper, our algorithm could be used in a variety of different contexts where highly similar sequences need to be identified from the data. We plan to extend our algorithm to make it parallelized by performing the traversal of each tree branch in parallel. Most the the tree is explored by sequences which end up becoming singletons and this dominated the running time. We

plan to explore different methods such as k-mer filters and locality sensitive hashing to flag singletons and exclude them from the recruiting process.

References

1. Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. Basic local alignment search tool. *Journal of molecular biology*, 215(3):403–410, 1990.
2. J. Gregory Caporaso, Christian L. Lauber, William A Walters, Donna Berg-Lyons, James Huntley, Noah Fierer, Sarah M. Owens, Jason Betley, Louise Fraser, Markus Bauer, et al. Ultra-high-throughput microbial community analysis on the Illumina HiSeq and MiSeq platforms. *The ISME journal*, 6(8):1621–1624, 2012.
3. Robert C. Edgar. Search and clustering orders of magnitude faster than BLAST. *Bioinformatics*, 26(19):2460–2461, 2010.
4. J. Felsenstein. PHYLIP-phylogeny inference package (version 3.2). *cladistics*, 5:164–166, 1989.
5. Mohammadreza Ghodsi, Bo Liu, and Mihai Pop. DNACLUST: accurate and efficient clustering of phylogenetic marker genes. *BMC bioinformatics*, 12(1):271, 2011.
6. Dan Gusfield. *Algorithms on strings, trees and sequences: computer science and computational biology*. Cambridge university press, 1997.
7. Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710, 1966.
8. Weizhong Li and Adam Godzik. Cd-hit: a fast program for clustering and comparing large sets of protein or nucleotide sequences. *Bioinformatics*, 22(13):1658–1659, 2006.
9. William J. Masek and Michael S. Paterson. How to compute string-edit distances quickly. In D. Sankoff and J. B. Kruskal, editors, *Time Warps, String Edits, and Macromolecules: the Theory and Practice of Sequence Comparison*, pages 337–349. Addison-Wesley Publ. Co., Mass., 1983.
10. William J. Masek and Mike Paterson. A faster algorithm computing string edit distances. *J. Comput. Syst. Sci.*, 20(1):18–31, 1980. doi:10.1016/0022-0000(80)90002-1.
11. Gerard Muyzer, Ellen C. De Waal, and Andre G. Uitterlinden. Profiling of complex microbial populations by denaturing gradient gel electrophoresis analysis of polymerase chain reaction-amplified genes coding for 16S rRNA. *Applied and environmental microbiology*, 59(3):695–700, 1993.
12. Eugene W. Myers. An $O(ND)$ difference algorithm and its variations. *Algorithmica*, 1(1):251–266, 1986.
13. Gene Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM (JACM)*, 46(3):395–415, 1999.
14. Temple F. Smith and Michael S. Waterman. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, 1981.
15. Julie D. Thompson, Toby Gibson, Des G. Higgins, et al. Multiple sequence alignment using ClustalW and ClustalX. *Current protocols in bioinformatics*, pages 2–3, 2002.
16. Lusheng Wang and Tao Jiang. On the complexity of multiple sequence alignment. *Journal of computational biology*, 1(4):337–348, 1994.
17. James R. White, Saket Navlakha, Niranjan Nagarajan, Mohammad-Reza Ghodsi, Carl Kingsford, and Mihai Pop. Alignment and clustering of phylogenetic markers-implications for microbial diversity studies. *BMC bioinformatics*, 11(1):152, 2010.