2021

SPARSE FORWARD-BACKWARD ALIGNMENT FOR SENSITIVE DATABASE SEARCH WITH SMALL MEMORY AND TIME REQUIREMENTS

David H. Rich

THESIS: SPARSE FORWARD-BACKWARD ALIGNMENT FOR SENSITIVE DATABASE SEARCH WITH SMALL MEMORY AND TIME REQUIREMENTS

By

David Hunter Rich

Bachelor of Science, The University of Montana, Missoula, MT, 2012

Thesis

presented in partial fulfillment of the requirements
for the degree of

Master of Science
in Computer Science

The University of Montana
Missoula, MT

Spring 2021

Approved by:

Ashby Kinch Ph.D., Dean
Graduate School

Travis Wheeler Ph.D., Chair
Computer Science

Jesse Johnson Ph.D.
Computer Science

Mark Grimes Ph.D.
Biology

Rich, David Hunter, M.S., June 2021 Computer Science

Thesis: Sparse Forward-Backward alignment for sensitive database search with small memory and time requirements

Chairperson: Travis Wheeler

Sequence annotation is typically performed by aligning an unlabeled sequence to a collection of known sequences, with the aim of identifying non-random similarities. Given the broad diversity of new sequences and the considerable scale of modern sequence databases, there is significant tension between the competing needs for sensitivity and speed, with multiple tools displacing the venerable BLAST software suite on one axis or another. In recent years, alignment based on profile hidden Markov models (pHMMs) and associated probabilistic inference methods have demonstrated increased sensitivity due in part to consideration of the ensemble of all possible alignments between a query and target using the Forward/Backward algorithm, rather than simply relying on the single highest-probability (Viterbi) alignment. Modern implementations of pHMM search achieve their speed by avoiding computation of the expensive Forward/Backward algorithm for most (HMMER3) or all (MMseqs2) candidate sequence alignments. Here, we describe a heuristic Forward/Backward algorithm that avoids filling in the entire quadratic dynamic programming (DP) matrix, by identifying a sparse cloud of DP cells containing most of the probability mass. The method produces an accurate approximation of the Forward/Backward alignment with high speed and small memory requirements. We demonstrate the utility of this sparse Forward/Backward approach in a tool that we call MMOREseqs; the name is a reference to the fact that our tool utilizes the MMseqs2 software suite to rapidly identify promising seed alignments to serve as a basis for sparse Forward/Backward.

MMOREseqs demonstrates improved annotation sensitivity with modest increase in run time over MMseqs2 and is released under the open BSD-3-clause license. Source code and Docker image are available for download at `https://github.com/TravisWheelerLab/MMOREseqs`.

ACKNOWLEDGMENTS

I would like to thank my parents, Jack and Belinda, for their love and support, without which none of this would be possible. Additionally to my sisters Kelly, Shannon, and the rest of my friends and family, for all dinners, drinks, and phone calls over the duration of this research and beyond.

I would also like the thank the members of my lab. First and foremost to my research advisor, Travis Wheeler, for this great opportunity and his guidance and contributions every step along the way of my research; George Lesica, for his assistance in developing my software, code reviews, and preparing MMOREseqs for release; Tim Anderson, for lending his eyes during code review; and to the rest of my lab for their invaluable input during our weekly meetings.

I would also like to express my appreciation for the University of Montana's Griz Shared Computing Cluster (GSCC) for access to their resources, and particularly Zachary Rossmiller, Allen Warren, and Shayne Johsson for their expertise in getting the cluster up and running, and for help with troubleshooting when things went sideways.

This work was supported by NIH grant P20 GM103546 (NIGMS) and DOE grant DE-SC0021216.

TABLE OF CONTENTS

COPYRIGHT ii

ABSTRACT iii

ACKNOWLEDGMENTS iv

LIST OF FIGURES vi

LIST OF TABLES vii

CHAPTER 1 INTRODUCTION 1

CHAPTER 2 RESULTS 4

CHAPTER 3 METHODS 14

CHAPTER 4 DISCUSSION 30

BIBLIOGRAPHY 32

LIST OF FIGURES

Figure 2.1 Efficacy and impact of sparsifying Forward/Backward matrix. 8

Figure 2.2 Efficacy and impact of sparsifying Forward/Backward matrix. 9

Figure 2.3 Recall as a function of false annotation rate 11

Figure 2.4 Run time vs. recall. 13

Figure 3.1 Cloud Search 20

Figure 3.2 Example anti-diagonal access pattern 21

Figure 3.3 Mapping anti-diagonal ranges to row-based ranges 23

Figure 3.4 Sparse Matrix 24

Figure 3.5 MMOREseqs Pipeline 26

LIST OF TABLES

2.1 **Long sequence pairs.** 6

CHAPTER 1 INTRODUCTION

The dominant method for accurate annotation of biological sequences is sequence database search, in which each unknown sequences is classified by aligning it to entities in a database of known sequences. This alignment-based approach of annotating sequences has historically been associated with the Smith Waterman algorithm [1] and fast heuristics such as BLAST [2]. In the years since the introduction of BLAST, profile hidden Markov models (pHMMs [3, 4, 5, 6]) have been shown to improve sequence search sensitivity. This sensitivity was initially offset by a significant run time penalty, but recent advances have produced pHMM software with competitive speed.

Much of the sensitivity of pHMMs is due to their natural representation of profiles [7] - when a collection of sequence family members is used to train the model, a pHMM captures the position-specific letter and gap frequencies inherent to the family. The value of position-specific scores has driven the development and use of databases of sequence alignments and accompanying pHMMs all across bioinformatics (e.g. [8, 9, 10, 11, 12]).

Perhaps less appreciated is the fact that pHMM-based software, such as HMMER, is typically more sensitive than BLAST even when aligning to a database of individual sequences rather than profiles [13, 14]. Unlike other alignment methods that compute just a single highest-scoring alignment (akin to a maximum probability Viterbi alignment in pHMM terminology), pHHMs can compute support for homology based on the sum of the probabilities of *all* alignments (treating the choice of specific alignment as a nuisance variable), via the Forward/Backward algorithm. Posterior probabilities resulting from the Forward/Backward algorithm also enable greater alignment accuracy [15, 16] as well as improved mechanisms for addressing composition bias and determining alignment boundaries.

Computing Forward/Backward is computationally expensive. Modern fast pHMM implemen-

tations achieve their speed by avoiding this computation for most (or all) candidate sequence relationships. HMMER3 [13] introduced a pipeline in which most candidates are never subjected to the most computationally expensive analysis, thanks to a series of filter stages. In the first stage (called MSV), all but an expected 2% of non-homologous sequences are removed from further analysis based on a test for high-scoring ungapped alignment. This is followed by computation of an optimal (Viterbi) gapped alignment score that typically allows passage of only 0.1% of non-homologous sequence. Thus, only reasonably-likely matches survive long enough to be subjected to the slow Forward algorithm and possible downstream analysis. In HMMER3, the result is pHMM alignment that is typically somewhat faster than blastp for protein database search, with essentially no loss in sensitivity over unaccelerated pHMM alignment. In common use cases, the first filter of HMMER3 (MSV) consumes ∼70% of HMMER's run time, and the final stage (Forward) consumes ∼20%. In cases of queries with extremely high length or large numbers of true matches, Forward dominates run time.

A newer acceleration approach introduced in MMseqs2 [17, 18] achieves even greater speed with some sensitivity loss. This speed is primarily due to two adjustments to the analysis pipeline. First, a lookup table is used to restrict further computation to only involve matches with two very short high scoring seeds; these seeds are extended to compute an ungapped alignment filter like that used in HMMER3. Next, MMseqs2 avoids the Forward/Backward step entirely, simply computing and reporting the Viterbi alignment. This produces impressive speed gains, and benefits from the advantages of position-specific scores in pHMMs, but misses out on the benefits of the more robust Forward/Backward algorithm.

Here, we describe a heuristic approach that constrains search space in the Forward/Backward dynamic programming (DP) matrix to a high-probability cloud. We show that our sparse Forward/Backward approach closely approximates the results of the full Forward/Backward algorithm, while providing a substantial reduction in space requirements and run time. We have implemented this sparse Forward/Backward approach in C, and demonstrate its utility in the context of our new software tool called MMOREseqs, which uses the MMseqs2 software suite as a filter and source of

candidate alignment seeds.

CHAPTER 2 RESULTS

The aspects of pHMM-based annotation that most leverage the available probabilistic underpinnings involve (i) computing the sum of probabilities of all alignments with the Forward algorithm, and (ii) downstream analyses (including creation of an alignment) based on posterior probabilities computed with Forward/Backward. The primary contribution of this work is development of a method to reduce the time and memory required for these stages. In other words, we sought a mechanism for approximating the results of filling in the quadratic-sized Forward/Backward dynamic programming matrices, without needing to actually fill in that entire matrix. Our approach is a close cousin to the X-drop heuristic used in BLAST: starting with a seed that establishes a region of interest within the dynamic programming (DP) matrix, and expanding DP calculations out in both directions until pruning conditions are met (details are found in the Methods section). Fig. 2.1 presents a single example of the reduced computation required by our sparse Forward/Backward for a relatively short alignment of one Pfam-based pHMM against a sequence belonging to the matching family.

We begin by describing the data used for evaluation, then demonstrate the space-pruning efficacy of our Cloud Search approach. We then show that sparse Forward/Backward analysis significantly improves accuracy over Viterbi-only analysis, at the cost of a moderate increase in run time. We close by evaluating a few basic pipeline hyperparameters, and describing the released software implementation.

Benchmarks

Pfam domain benchmark

Assessment was performed primarily using a benchmark created with software (*create-profmark*) that has previously been used to evaluate efficacy of the acceleration in HMMER [13]. The benchmark consists of 3,003 families from Pfam-A (v33.1) [8] that could be split into a training and test set such that no training-test pair of sequences shares greater that 25% identity. The training set defines a multiple sequence alignment for the family, which we refer to as the query. Sequences from the other group were down-sampled such that no two sequences are $> 50\%$ identical, leaving 35,456 in total; these were used as the basis of the test set. Each true test sequence was embedded in a larger unrelated sequence, to simulate the sub-sequence nature of the protein domains in Pfam; specifically, sequences were sampled from uniprot_sprot (2020_04), and shuffled prior to embedding. This set of sequences containing true positives was supplemented with 2 million additional sequences sampled and shuffled as above, but with no embedded matches. By construction, this benchmark contains cases that are highly difficult (and usually not impossible), for sequence alignment tools to recognize, in order to better emphasize differences in sensitivity. For more details on benchmark construction method and philosophy, see [13]. Note that our benchmark does not include reversed sequences, as these are prone to producing an excess of unexpected positives due to the surprising distribution high scores when aligning sequences to their reversals.

Long protein data set

Alignment with Pfam models represents a common use case for sequence alignment, but one that involves relative short sequences - the median Pfam domain length is just over 300. The purpose of our sparse Forward/Backward implementation is to avoid calculation over a full quadratic-sized dynamic programming matrix, and longer sequences are the ones that suffer most from this quadratic scaling; we therefore performed some tests using sequences on the longer end of the protein sequence length distribution. Specifically, we captured 6 pairs of long sequences from Uniprot (Table 2.1), and performed experiments to assess time and space efficiency along with

approximation accuracy. For each pair, one sequence was designated the *query* and the other the *target*.

Table 2.1: **Long sequence pairs.**

Query		Target	
Name	Length	Name	Length
TITIN_HUMAN	34,350	TITIN_MOUSE	35,213
EBH_STAAC	10,498	EBH_STAEQ	9,439
VLMS_LECSP	8,903	W4932_FUSPC	8,892
R1AB_CVH22	6,758	R1AB_BC512	6,793
HMCN1_HUMAN	5,635	HMCN1_MOUSE	5,634
RYR1_HUMAN	5,038	RYR1_PIG	5,035

Analysis pipeline - a sketch

To demonstrate the value of our sparse Forward/Backward algorithm in sequence annotation, we have incorporated it into a tool, MMOREseqs. The MMOREseqs pipeline uses MMseqs2 search as a subroutine, so that a candidate query/target pair are subjected to a sequence of filters and processing operations. In MMseqs2, (i) a k-mer match stage identifies candidate matches based on the presence of two co-diagonal k-position matches with score above a dynamically-selected threshold (e.g. 120 when used on our benchmark; MMOREseqs adjusts this to 80 by default), with a maximum of 1000 sequences allowed per query (MMseqs2 default: 300); (ii) above-threshold k-mer matches are extended to capture only those with good-scoring ungapped alignments (MMOREseqs retains the MMseqs2 default), then (iii) surviving candidates are subjected to a full Viterbi algorithm, which seeks the single highest-scoring (gapped) alignment for each query/target pair. MMOREseqs alters the final stage to report all results with P-value of 0.01 (MMseqs2 default: E-value=0.001); this filter is akin to the Viterbi filter of HMMER3 [13]). Alignments surviving these filter stages serve as input to our Cloud Search and sparse Forward/Backward implementation. Using the first and last positions of the MMseqs2 alignment as *begin* and *end* positions, our method identifies a cloud of dynamic programming matrix cells with non-negligible probability.

With this constrained space, our method then completes a sparse variant of Forward/Backward, which yields an overall alignment score along with position-specific posterior probabilities that positions are aligned; these posterior probabilities are used to compute a composition bias score adjustment along with the final sequence alignment. Because the pipeline relies heavily on MMseqs2 as a preprocessing stage, and identifies *more* good matches, we call the resulting pipeline MMOREseqs. See Methods for more details.

Sparse Forward/Backward reduces computation, is a good approximation

To evaluate our sparse Forward/Backward method, we tested the extent to which it reduces the number of computed cells, as this directly impacts time and space utilization. We also measured how well the sparse analysis approximates important byproducts computed using full Forward/Backward.

To analyze search space reduction, we computed the percentage of the full quadratic search space that is explored by the sparse approach (Fig. 2.2(A)). Reduction in search space is modest for shorter sequences; this is not surprising, as the total size of the dynamic programming matrix is not particularly large, so that a band around the maximum-scoring alignment will consume much of the analysis space. For longer sequences (on the order of 1,000 amino acids or longer), our sparse method often restricts the total number of computed cells to 1% or less of the full size of the matrix. For the longest protein sequences, a full quadratic-sized dynamic programming matrix consumes many gigabytes of memory, challenging the capacity of most modern hardware; a 1000-fold reduction in computed cells by our sparse method brings memory requirements into the scale of megabytes. The reduction in the number of computed cells means that the largest dynamic programming matrices can be computed in a small fraction of typical time, though the time reduction relative to optimized Forward/Backward (as in HMMER) does not match the space reduction due to additional computational overhead: (i) the value of each cell is computed more often - once or twice during the cloud search, then twice again during the sparse Forward/Backward step; and (ii) our approach does not leverage vector parallel instructions to speed alignment. Even

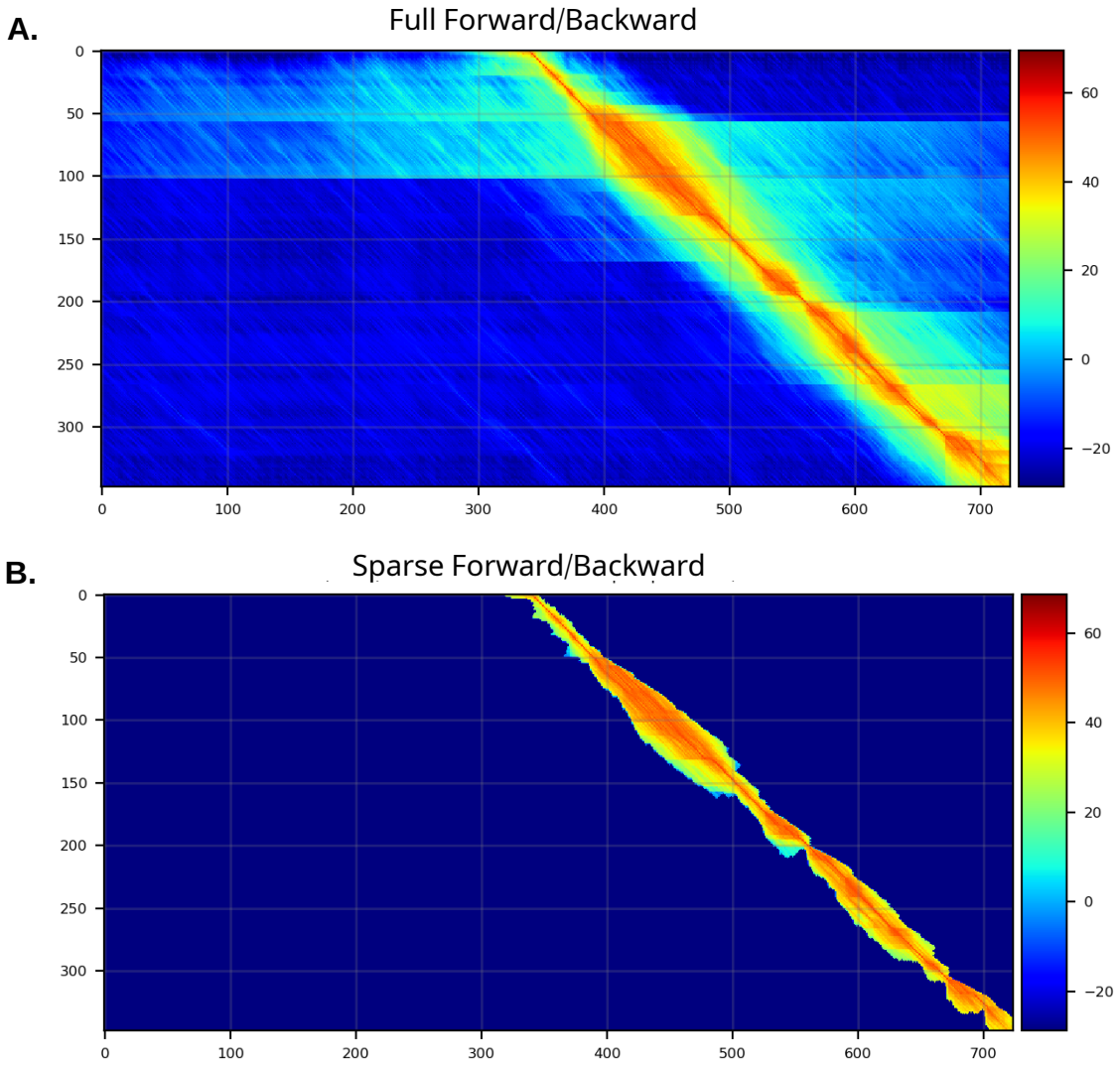

Figure 2.1: **Efficacy and impact of sparsifying Forward/Backward matrix.** Top panel (A) shows heatmap of scores per cell in the Match State matrix for the sequence Q01LW8_ORYSA aligned to the model for its matching family, DAO (FAD dependent oxidoreductase); bottom panel (B) shows the sparse set of (non-blue) cells that make up the cloud used for computing sparse Forward/Backward. The model positions are aligned along the y-axis and the sequence positions are aligned along the x-axis.

so, speed gains are significant (see below). Fig. 2.2(B) shows, for true positives from the domain benchmark, that the Forward score computed on the sparse matrix closely matches the score computed by Forward on the full matrix.

The dependency of MMOREseqs on MMseqs2 creates two common ways that a good alignment

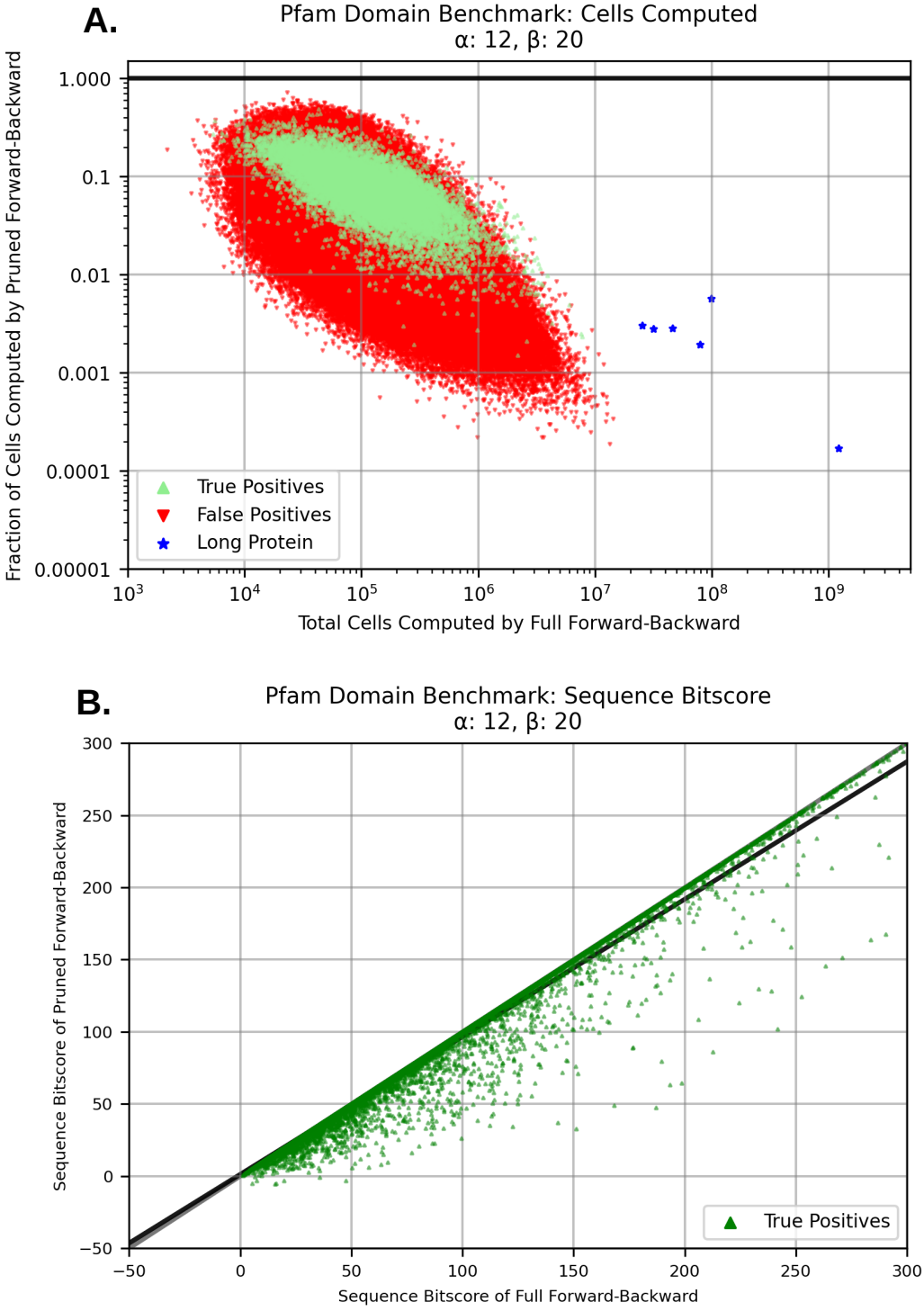

Figure 2.2: **Efficacy and impact of sparsifying Forward/Backward matrix.** (A) Green points show the fraction of the full matrix that is included in the sparse cloud, for all sparse alignments of true domain benchmark matches that survived the MMseqs2 Viterbi filter; red points show the same sparsification for false positives surviving the MMseqs2 Viterbi filter; blue dots (bottom right) show sparsification for long-sequence pairs. (B) Each point represents the relationship between sparse and full Forward scores on all domain benchmark true matches. Loss of score shows up as vertical depression below diagonal. 13,387 out of 16,299 true positive alignments lose less than 1% of score.

can be missed. In the most straightforward one, the MMseqs2 portion of the pipeline fails to find a good Viterbi alignment, so the sparse Forward/Backward stage is never given a chance to compute an alignment. The fast k-mer match stage of MMseqs2 is the common cause of such misses. These misses are responsible for most of the sensitivity difference between MMOREseqs and HMMER (see Fig 2.3). A more nuanced issue is that the Viterbi may identify one region as the highest-scoring alignment, though another region may produce a superior Forward/Backward score/alignment. Ideally, the sparse Forward/Backward stage would work with both such candidate regions, but our pipeline implementation only analyzes the region produced by MMseqs2 with the highest Viterbi score; this is left as future work.

Recall as a function of false annotation

We used the Pfam-based benchmark described above to assess the accuracy gains achieved with the Forward/Backward algorithm, and to measure the efficacy of our sparse implementation in retaining these gains. Each of the 3,003 query alignments was used to search for matching family members in the test database (containing 35,456 true sequences mixed with 2 million simulated sequences). An alignment was considered to be 'true positive' if at least 50% of the length of an embedded target sequence was covered by an alignment with the query from the same family. A hit that mostly covered simulated sequence was defined as a 'false positive'. An alignment between a query and target of differing families was treated as neutral (ignored) rather than being penalized, since it is not possible to ensure lack of homology across assigned family labels.

Fig. 2.3 presents a modified ROC plot. For each tested method, all resulting alignments were gathered together and sorted by increasing E-value. Each ROC curve shows how the fraction of planted positives that has been recovered (recall) grows as a function of the increasing number of false matches. MMseqs2 and HMMER3's hmmsearch were run with default settings to produce recovery results for comparison. On this benchmark, MMseqs2 labels no false matches, so that there is only a point on the plot.

Sensitivity gains for MMseqs2 were observed by requesting a reduced k-mer score threshold

(80), and increased E-value (E=1000) and maximum number of results per query (1000) lead to a mix of true and false matches. Inclusion of the full Forward/Backward calculation on all pairs passing the MMseqs2 Viterbi stage leads to a large increase in early sensitivity (MMORE-full). Note that the full set of aligned pairs is identical in the MMORE-full and MMseqs-k80 lines; significant gains in sensitivity are due to the superiority of Forward/Backward (supplemented by improved bias correction) in differentiating true matches from false matches. The full Forward/Backward establishes an upper bound on the sensitivity possible with sparse Forward/Backward.

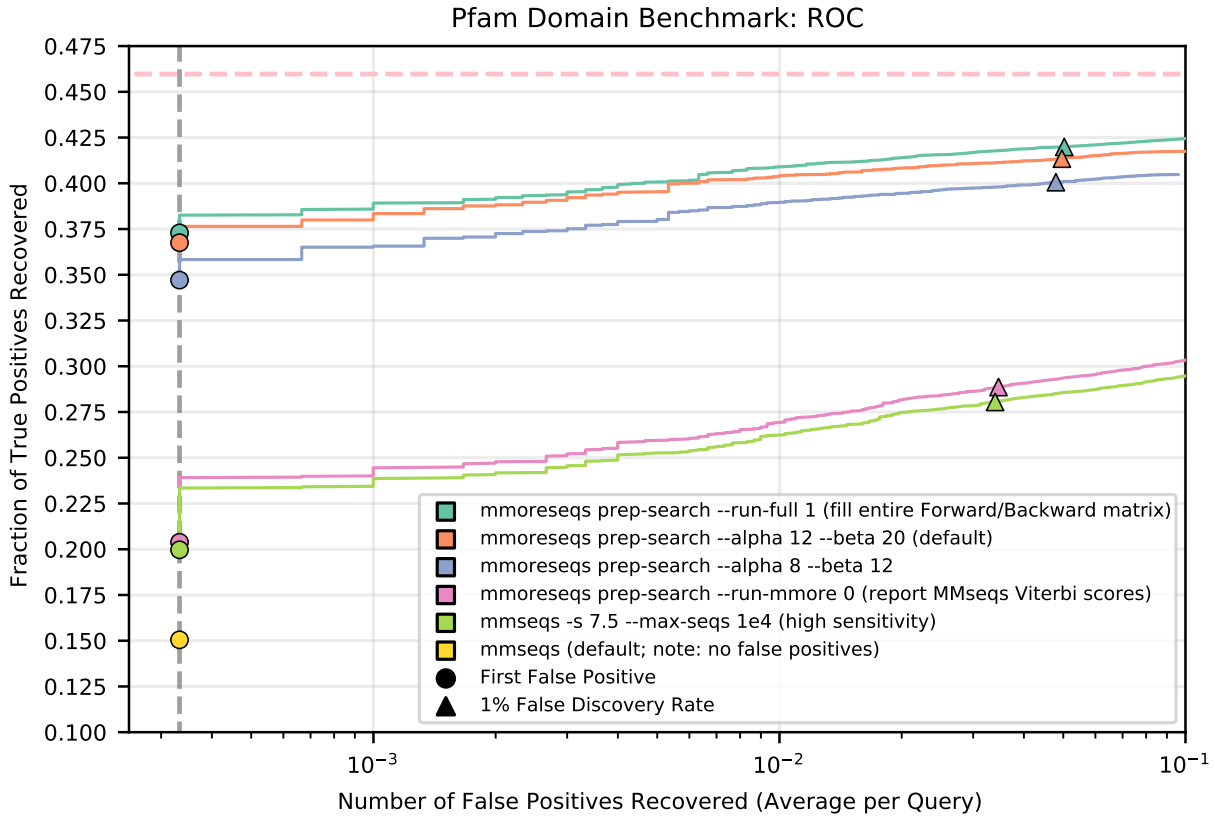

Figure 2.3: **Recall as a function of false annotation rate.** The protein domain benchmark consists of 35,456 true target sequences from 3,003 Pfam families, mixed with 2 million shuffled sequences from Uniprot. MMOREseqs and MMseqs2 were each tested with default parameters and a couple of exploratory parameters (listed in legend).

The MMORE-sparse curve in Fig. 2.3 shows the results of the full MMOREseqs pipeline, which filters Viterbi-aligned pairs with poor P-values, then uses surviving alignments as the basis of Cloud

Search and sparse Forward/Backward alignment and post-processing. The slight sensitivity gap between sparse and full Forward appears to be the result of the second source of error mentioned above: MMseqs2 sometimes produces a (best) Viterbi alignment that does not overlap the high-scoring alignment identified with a full Forward/Backward, so that the sparse Forward/Backward approach fails to find the true hit. Though no solution to this concern is provided here, in the future, this performance gap could be closed by exploring multiple good Viterbi matches in the Cloud Search stage.

Speed and Accuracy

Assessment of sequence annotation methods must consider the tradeoff between speed and sensitivity. In doing so, it is helpful to summarize the full sensitivity curves from Fig 2 with a simple statistic. Here, we use *recall prior to the first false positive* (fraction of planted positives assigned an E-value better than the best-scoring false positive). This summary statistic, which we call "recall-0", is easy to interpret, and generally agrees with other measures of relative performance. In Fig 2.4, we plot run time and recall-0 for annotation of the Pfam-based benchmark described above. These results show that inclusion of sparse Forward/Backward alignment in an MMseqs2-based annotation pipeline produces increased sensitivity at a modest run time cost. We view these results as a conservative estimate of the speed benefits of the sparse Forward/Backward approach, because the Pfam-based domain sequences are relatively short; the relative speed/recall tradeoff is expected to be increasingly in favor of sparse Forward/Backward for longer sequence elements.

For completeness, we include results of searching with HMMER3; it produces much higher sensitivity at larger run time costs. The significant drop-off in sensitivity between HMMER3 and MMOREseqs is caused by aggressive filtering of candidates by the k-mer match stage in MMseqs2. The plot also shows the effect of further reducing the k-mer score cutoff beyond the MMOREseqs default of 80 and alternatively increasing the post-Viterbi filter threshold to a more permissive level. In both cases, sensitivity shows a small increase, at a cost in run time.

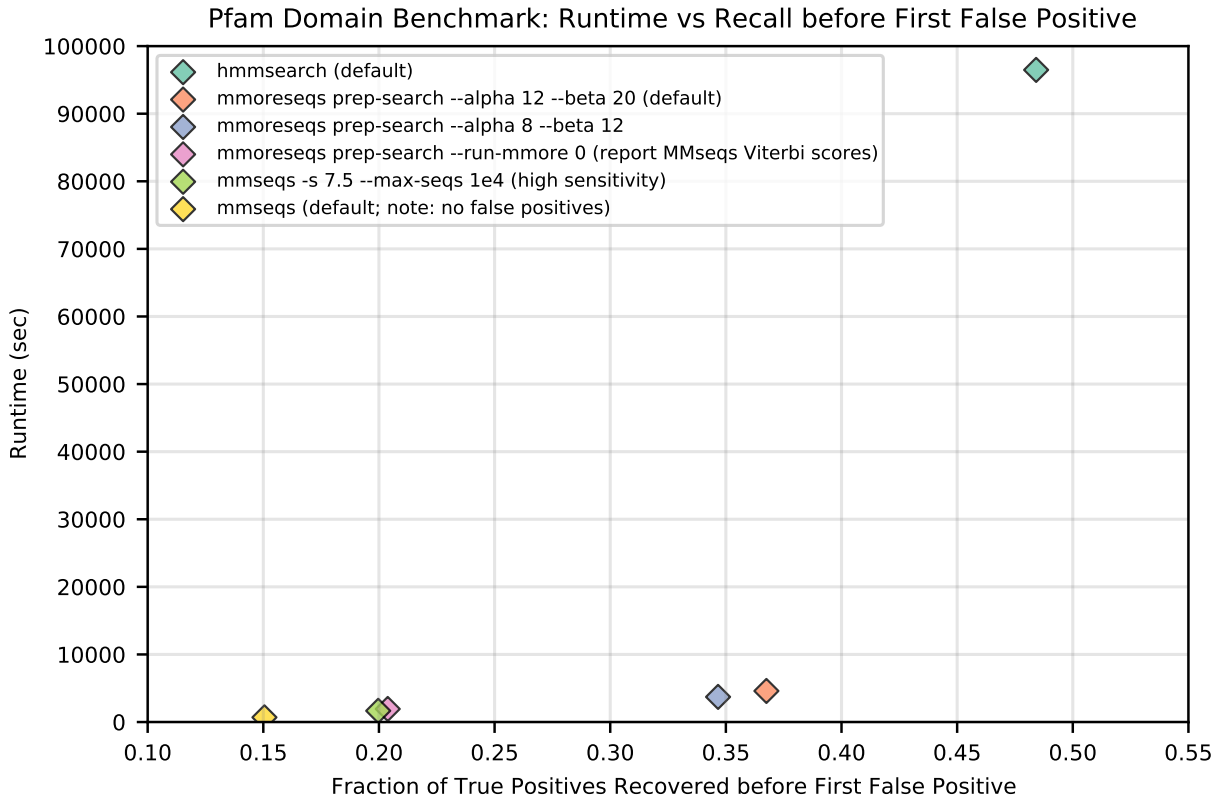

Figure 2.4: **Run time vs. recall.** Pfam-based benchmark was subjected to search with MMseqs2 variants, HMMER3's hmmsearch (default), and a sampling of MMOREseqs variants intended to demonstrate a performance-runtime tradeoffs. Parameters for each are provided in figure legend. The *mmoreseqs –run-mmore=0* flag causes MMOREseqs to run MMseqs2's prefilter with a k-score of 80, with a maximum of 1000 reported sequences per query, and simply reports MMseqs2's Viterbi-based profile-to-sequence search E-values with an effective P-value of 0.01.

CHAPTER 3 METHODS

This manuscript describes both (i) an efficient approach for identifying a sparse cloud of non-negligible-probability cells within the standard quadratic-sized Forward/Backward sequence alignment search space, and (ii) release of sequence database search software that applies this approach to annotation candidates produced by running MMseqs2 [17] with permissive parameters. We begin with a description of the standard Forward/Backward algorithm, then describe the procedure for identifying a sparse cloud within the corresponding alignment search space, then provide details for construction of the software pipeline.

Default implementation of the Forward/Backward algorithm

To prepare for discussion of a sparse alignment implementation, we first describe the standard implementation of the Forward/Backward algorithm for profile HMMs. Input to the algorithm consists of:

- An alphabet Σ of size k (e.g. $k = 20$ for the amino acid alphabet).
- A length-n target sequence $T = t_1, t_2, \ldots, t_n$, with all $t_j \in \Sigma$.
- A query model Q defined by a collection of values organized around three core states for each of m positions:
  - Match states (M) emit letters from Σ with a position-specific distribution, and during alignment are used to associate (match) a letter t_j from T to a position i in Q;
  - Insert states (I) emit letters in between match-state letters, and during alignment allow some letters in T to not correspond to positions in Q (i.e. to lie between matched letters).

In principle, position-specific insertion emission probabilities are legal, but we follow the common convention that these share a distribution with the random sequence, so that there are no position-specific insert state emission values;

- Delete states (D) are silent states (no emission) that, in alignment, allow consecutive letters in T to associate with non-consecutive positions in Q (i.e. some positions in Q are not represented, or are deleted, in T).

In support of these states, Q is described by two matrices:

1. For each position i, emissions of match state M_i are defined by a length-k vector $q_{i1}, q_{i2}, \ldots, q_{ik}$, where a value q_{ic} corresponds to the model's probability of observing letter c at position i (specifically, this is a succinct representation of the odds ratio for each (i, c) pair, capturing the ratio of the emission probability of letter c at position i over a background emission probability for c; see [5] for a more verbose form).
2. A transition matrix that captures the probability of transitioning from one state to another in sequential positions (following common convention, we do not include transitions between D and I states):
   $t(M_i, M_{i+1}), t(M_i, D_{i+1}), t(M_i, I_i), t(I_i, I_i), t(I_i, M_{i+1}), t(D_i, D_{i+1}), t(D_i, M_{i+1})$

With this input, the Forward algorithm fills in three $(m + 1)(n + 1)$ matrices, one for each state (F^M, F^I, and F^D); the value stored at a cell (i, j) in a state's matrix corresponds to all ways of aligning the first j letters of T with the first i model positions, ending in that state. After initializing $F^M_{0,0} = F^D_{0,0} = F^I_{0,0} = 0$, the remaining matrix cells are computed via the recurrence equations:

$$\mathrm{F}^{M}_{i,j} = q_{it_j} \cdot sum \begin{cases} F^{M}_{i-1,j-1} \cdot t(M_{i-1}, M_i) \\ F^{I}_{i-1,j-1} \cdot t(I_{i-1}, M_i) \\ F^{D}_{i-1,j-1} \cdot t(D_{i-1}, M_i) \end{cases}$$

$$\mathrm{F}^{I}_{i,j} = sum \begin{cases} F^{M}_{i,j-1} \cdot t(M_i, I_i) \\ F^{I}_{i,j-1} \cdot t(I_i, I_i) \end{cases}$$

$$\mathrm{F}^{D}_{i,j} = sum \begin{cases} F^{M}_{i-1,j} \cdot t(M_{i-1}, D_i) \\ F^{D}_{i-1,j} \cdot t(D_{i-1}, D_i) \end{cases}$$

Notes:

- The result of the Forward algorithm is a ratio of the sum, over all possible alignments, of the probability of observing T under the assumption of relationship to Q, divided by the probability of observing T under a random model. The log of this ratio is a score, and the E-value of an alignment can be computed based on how this score relates to the distribution of scores for random alignments (see [19]).

- This recurrence is similar to the Viterbi recurrence for finding the highest-probability path/alignment; it differs in that it sums the values of alternate paths, rather than selecting the max;

- This description addresses only the core model and assumes global alignment; local alignment, and additional states, require straightforward modifications to the recurrence, e.g. see [19]).

- The recurrence involves calculation of the products of probabilities, and can suffer from numerical underflow. The Viterbi (max) method avoids underflow by performing all computations

in log space. The is not possible for the Forward algorithm, due to the fact that it adds probabilities. This is often addressed by moving values in and out of log space (supported by fast approximation of $\log(p_1 + p_2)$); this is the method used in our implementation. Some implementations demonstrate further acceleration by scaling values directly in order to avoid conversion to log space entirely [13].

- Though the recurrence suggests recursive function calls, the matrix can be computed by filling a table in an ordered fashion, due to the ordered local dependencies of computations. This is usually performed in row-major order (filling from upper left to lower right, one row at a time), though dependencies allow for other orders, such as filling in sequential anti-diagonals. A striped pattern of vectorized data access has also been used to accelerate computation [20, 13].

The Forward algorithm computes a measure of support for the relationship between T and Q, but does not directly produce a specific alignment between the two. One important byproduct of the calculation is that each (i, j) cell in the *Forward* matrices represents the probability of all alignments *ending* in the corresponding state, having accounted for the first j letters of T and the first i positions of Q. A common followup to Forward is to perform the same sort of computation in reverse, filling in tables from lower-right to upper-left based on an inversion of the recurrence for Forward. This *Backward* algorithm computes, for each cell, the probability of all alignments *starting* at t_j, and model position i. The Forward and Backward matrices can be combined (see [5]) to produce, for each cell, a posterior probability of that cell being part of the true alignment. This posterior probability matrix can serve as the basis of an alignment with maximum expected accuracy [15, 5]. We omit details, as they are not required to understand the work here, but note that typical calculation of each of these matrices is performed across the full quadratic alignment space.

Efficient search for high-probability cloud in Forward/Backward matrices

The Forward/Backward computation described above captures the total probability of all possible alignments, and in doing so, fills in multiple matrices with quadratic size (the product of the lengths of T and Q). We improve computational efficiency with a heuristic that exploits the fact that this is usually overkill - most possible alignments have such low probability that excluding them from computation has no relevant impact on the overall sum of probabilities. Our approach aims to identify which matrix cells contain non-negligible probability, and limit calculations to touch only those cells. Doing so minimally impacts computed scores and resulting sequence alignments, while substantially reducing the total computation. In this section, we describe a heuristic approach for achieving this goal. The method, which we call *Cloud Search* resembles the well-known X-drop algorithm used in maximum-score alignment methods such as blast [cite]; it begins with a seed that provides guidance on where high-probability cells are likely to be found, then expands a search forward and backward across the matrices for a cloud of cells around this seed that appear to contain essentially all relevant probability mass. This constrained space is then used as the basis for all downstream analysis.

Cloud Search by pruned anti-diagonal completion

The method proceeds as follows

- Cloud Search is initiated with a pair of *begin* and *end* alignment matrix cells. As implemented here, this pair is taken from an MMseqs2 Viterbi alignment between Q and T (Fig 3.1: 'Viterbi Alignment') – the first and last positions of the alignment specify the pair of cells (i_b, j_b) and (i_e, j_e). In practice a cell pair could be produced by some other approach, such as the ungapped alignment phase in MMseqs2 or HMMER. Also, in practice, the Cloud Search stage could be initialized by more than one such pair of begin/end cells.

- Cloud Search flood-fills the matrices forward (down and right) from the *begin* cell, extending out until pruning conditions are reached (Fig 3.1: 'Cloud Forward'). After initializing $F^M_{i_b,j_b} = F^D_{i_b,j_b} = F^I_{i_b,j_b} = 0$ (red cell in upper left), neighboring cells down and right of (i_b, j_b) are

computed in anti-diagonal fashion, first filling the two cells (i_{b+1}, j_b) and (i_b, j_{b+1}), then the three cells below these, and so on. Cells on one anti-diagonal push values to cells in subsequent anti-diagonals, so that the only cells computed on on anti-diagonal are those that are reachable from some active cell on a previous anti-diagonal. Beginning from (i_b, j_b), all reachable anti-diagonal cells are computed and retained, until the anti-diagonal achieves length γ (default: 5). After this, two pruning conditions are applied, to constrain expansion of search space (pruning is performed based entirely on values stored in the Match state matrix F^M, and all scores are maintained in log space).

- Once all values in an anti-diagonal d have been computed, the maximum value for that diagonal is captured as $\max_d$. All cells with $F^M_{i,j} \geq \max_d - \alpha$ are retained, and others are pruned. Scores at this point are captured in *nats* (natural logarithms), with default $\alpha = 12$, so that this effectively prunes cells on an anti-diagonal that have probability that is ~ 6 orders of magnitude lower than the most-probable cell on that anti-diagonal.
- As flood fill continues, the overall best-seen score across all computed anti-diagonals is captured as $\max_o$. Any cell with score $F^M_{i,j} < \max_o - \beta$ is pruned. With a default $\beta = 20$, this prunes cells with ~ 9 orders of magnitude drop-off from the best seen value (this is analogous to X in the X-drop heuristic). When all cells in an anti-diagonal are pruned, the flood fill stops.

The result of this phase is a set of cells expanding down and right from (i_b, j_b), schematically represented as purple cells in in Fig 3.1: 'Cloud Forward'. This cloud of cells typically remains in a fairly tight band around the maximum probability Viterbi path, and rarely extends much beyond the end of the alignment computed using unbounded Forward/Backward. Importantly, this cloud search approach typically *does* extend well beyond the initial *end* cell (i_e, j_e), meaning that a conservative selection of initial points does not constrain the Forward cloud search.

- After the forward Cloud Search phase, a similar backward pass is performed, beginning at

(i_e, j_e), and flood filling as in the previous stage, up and to the left (Fig 3.1: 'Cloud Backward'; green cells)

- Cloud Search concludes by selecting the union of the forward and backward clouds (Fig 3.1: 'Cloud Union'), establishing a set of cells that hold a non-negligible expansion around the range bounded by the initiating cells (i_b, j_b) and (i_e, j_e).

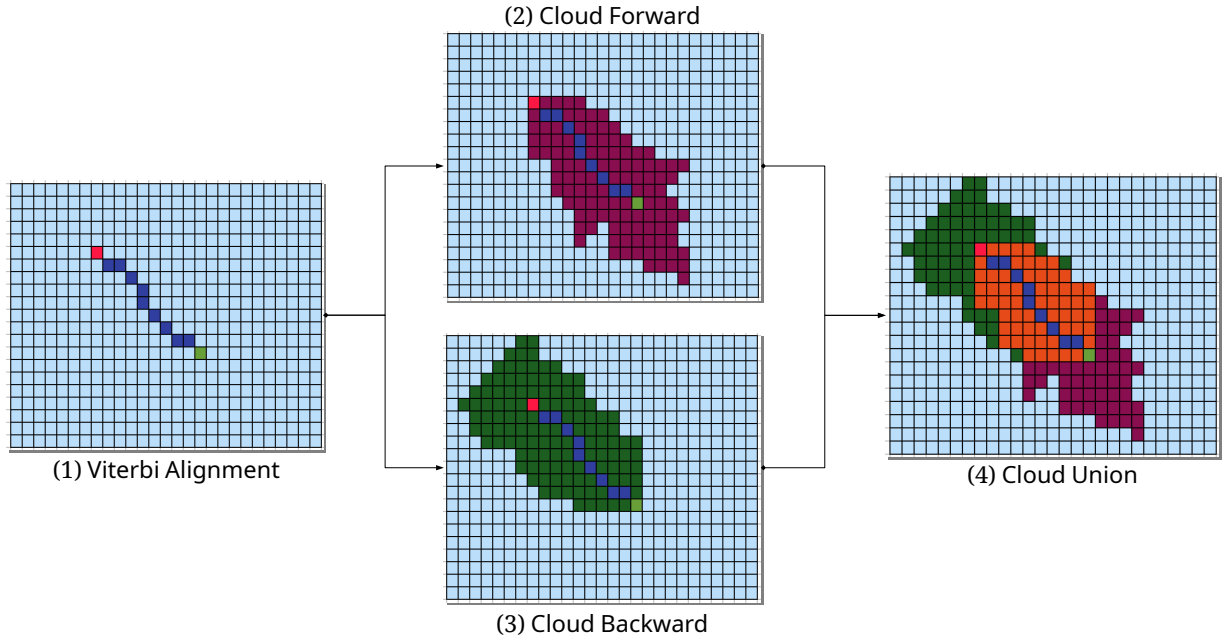

Figure 3.1: **Cloud Search.** In this schematic representation of Cloud Search, (1) a Viterbi alignment from MMseqs2 is used as the source of begin- and end-points (red and green; these could come from any source); (2) a scan is performed in the forward direction from the being point, pruning based on score-dropoff conditions, usually extending beyond the end point; (3) an similar scan is performed in the reverse direction starting from the provided end point; and (4) the union of the two resulting spaces is identified as the sparse cloud.

Linear space for anti-diagonal flood fill

The forward and backward cloud search stages can be computed in linear space, with three vectors each of length at most $m + n$. In Fig 3.2, the top-left matrix shows the three vectors as they correspond to the implicit complete dynamic programming matrix; in this example, when computing the dark green cell in anti-diagonal 13, the standard recurrence requires access to two

neighboring (orange) cells from anti-diagonal 12 and one (dark red) cell from anti-diagonal 11. The bottom left matrix shows how these three active anti-diagonals are laid out in a three-vector data structure during calculation of anti-diagonal 13. The right side of Fig 3.2 demonstrates how, when computing (blue) anti-diagonal 14, the vector holding the now retired values of anti-diagonal 11 is reused. In general, for a given matrix cell $F^{\cdot}_{i,j}$, its diagonal $d_0 = i + j$ is assigned to vector d_0 mod 3, with the cell located at offset i in the vector. Modifications to the recurrence equations follow naturally.

In our implementation, after each anti-diagonal is computed, it is trimmed from its outer edges, pruning inward until the first cell is reached that survives the threshold. Remaining cells are captured to an efficient list of (anti-diagonal index, left edge index, right edge index) tuples. We have developed an alternative implementation that conditionally prunes every cell. This approach provides no benefits over the edge-trimming approach for our pipeline, but will be the preferred option in a future implementation that supports multiple seeds (and downstream possibly-overlapping clouds).

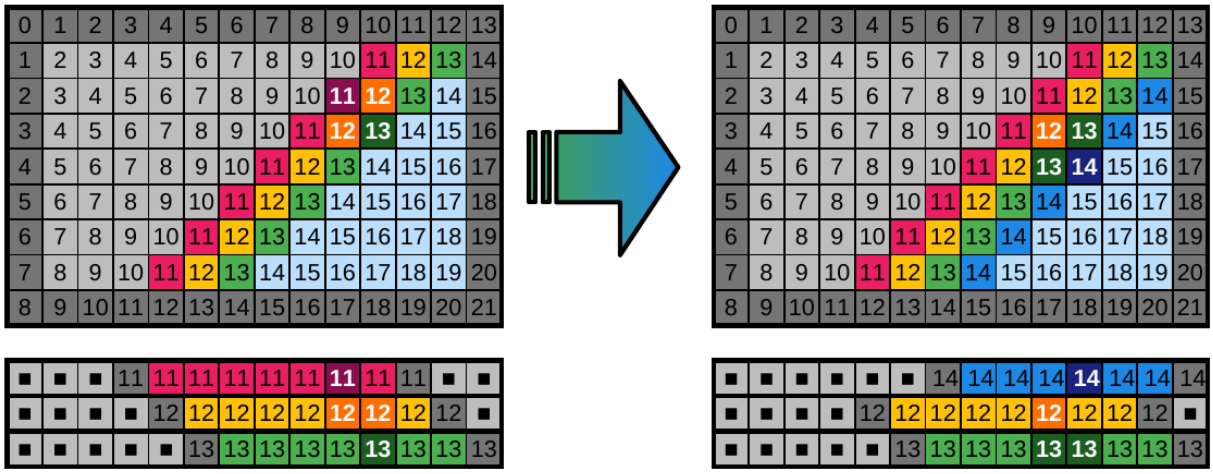

Figure 3.2: **Example anti-diagonal access pattern.** Only three arrays are used during anti-diagonal Cloud Search. The example shows the state in those three arrays when anti-diagonal 13 is being filled: the two previous anti-diagonals are maintained, due to data dependencies, and those dependencies are represented with darker-colored cells (with white letters). When the next anti-diagonal is computed, its calculations can be performed by reusing the oldest array from the previous phase.

Union and Reorientation:

The pruned anti-diagonal ranges from the Cloud Search are captured as a forward and backward list of the aforementioned tuples (anti-diagonal index, left edge index, right edge index). A union of these ranges is computed, yielding a new list that defines the unioned cloud as a list of anti-diagonal start/stop tuples.

Next, the list is reoriented to produce a list of ranges organized by row. Let R_i refer to the set of ranges for row i, with $R_i[k]$:= the k^{th} range (typically, there is only one range per row), $R_i[k]$.start := the beginning of the k^{th} range, $R_i[k]$.end := the end of the k^{th} range, and last_i := the index of the right-most range in R_i. Beginning with the left-most anti-diagonal range, the following actions are performed for each cell in each anti-diagonal range:

- compute the cell's row i and column j, based on anti-diagonal and offset within that anti-diagonal
- if R_i = null (no range yet exists for row i), create $R_i[0] = (j, j)$.
- else if $j == R_i[\text{last}_i].\text{end} + 1$, increment $R_i[\text{last}_i]$.end (extend the active range for that row)
- else consider that previous range closed, and begin a new range ($\text{last}_i += 1; R_i[\text{last}_i].\text{start} = R_i[\text{last}_i].\text{end} = j$)

The run time of this procedure is proportional to the number of cells in the cloud (which empirically grows roughly as a function of the length of the longer of the Query and Target).

Sparse Forward/Backward to recover score and alignment

With the cloud of non-negligible alignment matrix cells in hand, it is possible to compute an approximation of the full Forward/Backward alignment algorithm by filling in only cells in the cloud, implicitly treating all other cells as if they carry a probability of zero.

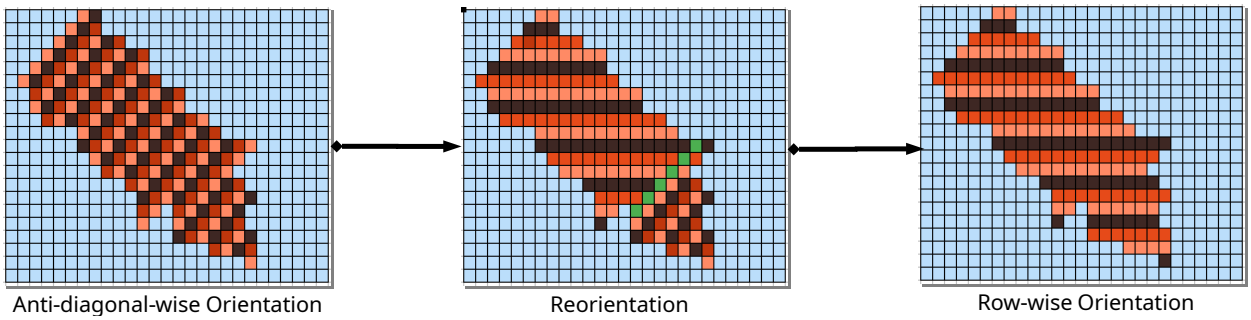

Figure 3.3: **Mapping anti-diagonal ranges to row-based ranges.** During the Cloud Search phase, data is arranged according to positions on successive anti-diagonals. After Cloud Search has completed, these ranges are reoriented into ranges based on rows, to simplify downstream sparse calculations.

Sparse matrix organization

To compute a Forward/Backward approximation, the ranges R are used as the basis for creating a sparse version of each of the matrices M, I , and D. Blocks of active cells in the cloud are considered in order, and padded with cells necessary to ensure safe performance of the Forward-Backward recursion without special cases for edge-checking. Once the padded search space list is created, it is used as the basis of a single array layout that will hold all required cells (Fig. 3.4A). In practice, the space required to hold active and buffer cells this is generally only slightly larger than the number of active cells. This layout is used to allocate a sparse M, I, and D matrix in the form of an array for computing sparse Forward, another three copies for computing Backward, and a single array for computing per-cell posterior probabilities in support of optimal accuracy alignment.

This flat array is supported by a table of complementary offsets that enable rapid identification of locations in the flat array corresponding to positions in the implicit matrix (Fig. 3.4B), with one tuple of offsets for each block of active cells.

Sparse Forward-Backward

Computing the sparse approximation of Forward-Backward is a simple matter of traversing the compressed arrays in increasing order for Forward (and decreasing order for Backward), in runs defined by blocks of active cells. When filling in the sparse DP matrix, pad cells are set to zero,

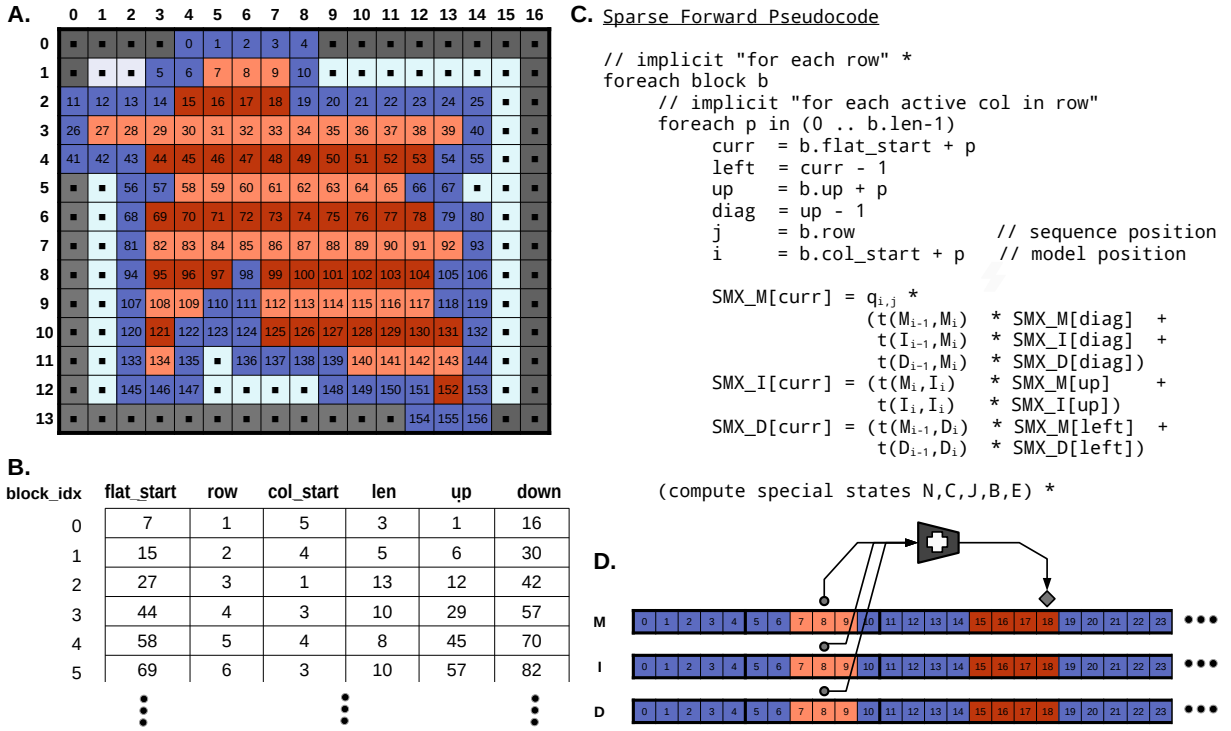

Figure 3.4: **Sparse Matrix.** Example organization of a sparse cloud into a flat array with supporting offset data, and demonstration of its use. (A) sparse cloud cells in red/pink are supplemented with the set of buffer cells (blue) that ensure that any Forward/Backward calculation dependencies will refer to either a cloud or buffer cell (to avoid conditionals in the DP inner loop). Numbers in the cloud/buffer cells indicate the corresponding index in the flat array. (B) Table of values required to compute offsets into flat array during DP recurrence computation. (C) Pseudocode for the sparse Forward implementation. The three F-matrices for default computation (F^M, F^I, F^D) are replaced with three flat (sparse) arrays (SMX_M, SMX_I, SMX_D). The sparse Backwards recurrence is similarly structured. (* In most cases, a single block in the flat array corresponds to a single row in the implicit full matrix, but when a row has multiple corresponding blocks (e.g. rows 8 through 11 in this example), special states are computed only after all blocks for the row. (D) Example of computing the DP recurrence for the implicit (2,7) cell, which requires identification of appropriate cells in the SMX_M, SMX_I, SMX_D arrays, using the offsets from (B).

and other cells are computed based on the standard recurrence equations, with offsets into the compressed arrays determined using the mappings calculated during the sparse vector layout stage (pseudocode: Fig. 3.4C). To compute cell-wise posterior probabilities, the product of the Forward and Backward matrices are computed in the usual fashion, and a Maximum Expected Accuracy alignment is identified based on these posterior probabilities [5].

MMOREseqs implementation

To demonstrate the efficacy of this Cloud Search and sparse Forward/Backward implementation, we have developed a tool that we call MMOREseqs, in reference to the fact that it runs the open source software *MM*seqs2 [17] to identify alignment begin/end points for Cloud Search, then proceeds with downstream sparse Forward/Backward and alignment to find *more* hits. MMseqs2 is a database search tool that performs very fast pHMM sequence database search with blast-like accuracy, achieving this speed by performing Viterbi (maximum-probability) alignment on candidates initially identified using a filter that requires two above-threshold k-mer matches on the same diagonal. This approach carries some inefficiencies at the interface, but clearly demonstrates an effective increase in alignment sensitivity with good speed.

The MMOREseqs analysis pipeline is divided into four major stages: (0) File Preparation, (1) MMseqs2 filter, (2) MMOREseqs filter, and (3) MMORE final processing. The default version of MMOREseqs accepts two input files: (i) a Query file consisting of one or more multiple sequence alignments (MSAs, in Stockholm format), and (ii) a Target file consisting of one or more sequences (in FASTA format).

(0) File Preparation Step:

The preparation phase produces all the necessary file formats required by the pipeline, based on the raw input files. In order to implement our pruned Forward/Backward, we opted to construct and use a pHMM with the HMMER3 format, using the tool *hmmbuild*. MMseqs2 constructs a pHMM with custom format from an input MSA. We are not aware of a documented method for converting between these two HMM formats. Importantly, MMseqs2 and HMMER use different criteria for selecting which MSA columns are represented by model states, so that the models may be of different length and can not necessarily be registered to each other. To overcome this, we also depend on consensus sequences derived from the models produced by MMseqs2 and HMMER (see below for details).

Therefore, six inputs are prepared for the primary pipeline. For the Query: an MMseqs2

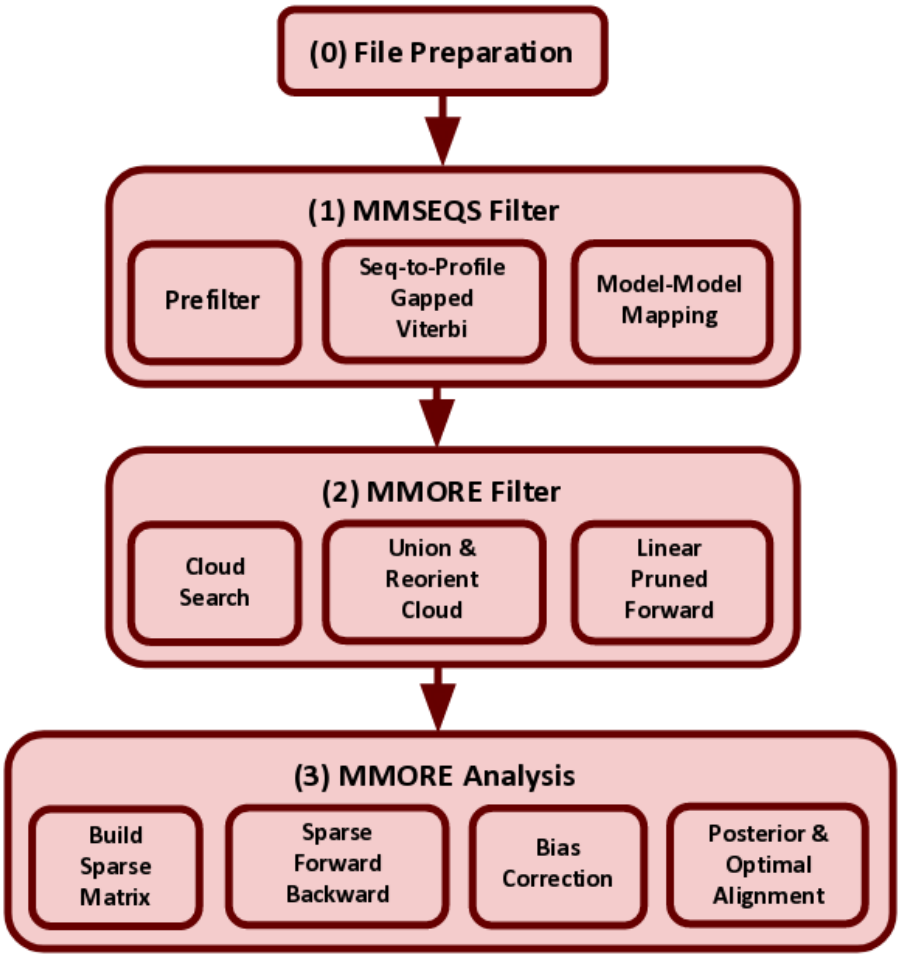

Figure 3.5: **MMOREseqs Pipeline.** This flowchart lays out the individual steps of the MMOREseqs pipeline, progressing from top to bottom, left to right.

pHMM file, a file containing one consensus sequence for each MMseqs2 pHMM (FASTA format), a HMMER pHMM file, and a file with consensus sequences for each HMMER3 pHMM (FASTA). For the Target: the original Target file input (FASTA) and an MMseqs2-formatted sequence database (consisting of multiple integrated files).

(1) MMseqs2 as a filter

MMOREseqs uses the MMseqs2 analysis pipeline as a filter in front of the sparse Forward/Backward analysis step, and to identify the Cloud Search begin/end seed positions. We describe the

relevant aspects of MMseqs2, because parameter selection in MMseqs stages impacts downstream annotation results.

MMseqs k-mer filter: A key driver behind the speed of MMseqs2 is its initial *two k-mer* search step, which quickly finds length-k alignments above a given score threshold τ, and identifies all candidate query/target pairs with at least two such seeds on the same matrix diagonal (i.e. no insertions or deletions separating high-scoring k-mer alignments). The value of τ is dynamically selected by MMseqs2 based on input conditions (particularly search space size); for our analysis benchmark, τ is dynamically set in MMseqs2 to 120. The MMOREseqs pipeline overrides this default, reducing τ in order to allow more sequences to pass to later analysis stages. By testing a small number of more lenient values, we have selected a default $\tau = 80$ for the MMOREseqs pipeline. No change was made to default seed length or spacing patterns in MMseqs2's k-mer filter. Overriding the default cap, the first 1000 k-mer filter-passing matches to a query are passed along to the next analysis stage.

MMseqs2 Profile-to-Sequence Viterbi filter: Candidate alignments passing the *two k-mer* filter are passed to the MMseqs2 implementation of Viterbi alignment between Target sequence T and the Query model Q (using the 'search' module). This stage produces a maximal-scoring alignment between Q and T, and computes significance (E-value) for each alignment. In the standard MMseqs2 pipeline, such alignments are sorted by E-value, and only those with a better-than-threshold value are reported. In the MMOREseqs pipeline, these significance values are treated as filtering criteria, such that the filter passes along all candidate alignments with a score corresponding to at most P-value of 0.01 (i.e. 1% of non-homologous Q-T pairs are expected to pass the filter).

MMseqs2 Model-to-Model Mapping: Ideally, the previous step would provide landmarks in the dynamic programming matrix (begin/end cells) for each candidate Q-T pair passing the filter. Because the MMOREseqs post-Viterbi stages utilize a HMMER3-style pHMM, while the Viterbi results

correspond to an MMseqs2-style pHMM, the pipeline is required to map the MMseqs2 pHMM position to the corresponding HMMER3 pHMM position. This is accomplished by performing a Viterbi alignment of each MMseqs pHMM against the a consensus sequence generated from the corresponding HMMER3 pHMM, using the MMseqs 'search' module. This produces an alignment where Insert states indicate an additional node in the MMseqs pHMM, Delete states indicate an additional node in the HMMER3 pHMM, and Match states indicates the nodes corresponding to a common position in the MSA. This alignment can be used to map between the two pHMMs through a linear scan.

(2) Sparse Forward/Backward module:

Each candidate alignment pair passing the previous filter is subject to Cloud Search, seeded by the begin/end positions from sequence-to-sequence alignment. For this stage the files used are (i) the FASTA-formatted target sequences, and (ii) the pHMM build by HMMER's hmmbuild from the Query MSAs.

Cloud Search: For each candidate pair, the previously-described Cloud Search step is completed, identifying the union of forward and backward expansion from the seed cells, to produce a sparse subset of dynamic programming matrix cells.

Cloud filter, and Forward filter: Though reduced space Forward/Backward is fast, many of the input alignment candidates will produce such a low-quality alignment that they will not end up being reported. To avoid time spent analyzing such candidates, MMORE performs two filters in sequence. The more robust of these is a filter applied after computing the Forward score using the sparse cloud: using the sparse Forward score, a P-value is computed and matches with P>1e-4 are removed (so that 0.01% of unrelated sequences are expected to pass the filter; this is similar to the Forward filter used in HMMER3).

But even before computing the Forward score on the sparse cloud, MMOREseqs is able to

approximate that score using a method that we call 'cloud filter', which adds the Forward score (starting at the begin cell) and Backward score (starting at the end cell) computed during Cloud Search, approximately adjusting for score shared by the two waves. This adjustment is achieved by estimating how much of the forward pass score must have been missed in the reverse pass, and vice versa. To do this, MMseqs keeps track of the best score observed during forward Cloud Search expansion (best_fwd), and the best score observed before extending past the anti-diagonal containing end cell (best_infwd). The difference (Z = best_fwd - best_infwd) is an estimate of the part of the forward pass's score that is not shared by the two passes of Cloud Search. A similar value is captured during the backward pass of Cloud Search (A = best_bkwd - best_inbkwd). The total Forward score is then estimated as A + max(best_infwd,best_inbkwd) + Z; a P-value is computed for this, and only candidates with corresponding P<1e-3 are passed on to the Forward stage.

Bias correction, alignment boundaries, alignment: For all downstream analyses, MMOREseqs follows the methods of HMMER3, but with a sparse matrix implementation. This (i) includes estimation of the effect of composition bias on the alignment score, and corresponding score adjustment, (ii) identification of the start and end of an aligned region based on posterior probabilities captured in states that precede and follow the core HMMER3 model (HMMER's 'domain definition' step), and (iii) maximum expected accuracy alignment. Resulting (bias-corrected) scores are converted to E-values as in HMMER (see [19]).

Test Environment

All tests were performed on compute nodes in the University of Montana Griz Computer Cluster (GSCC), each with two Intel Xeon Gold 6150 (2.70GHz) 18 core processors and 754GB RAM. All tests were performed with a single thread on a dedicated system, and standard wall clock times were captured.

CHAPTER 4 DISCUSSION

As implemented, MMOREseqs demonstrates that it is possible to employ powerful Forward/Backward inference with significantly reduced time and memory requirements. While we expect that MMOREseqs will play a valuable role for researchers seeking high database search sensitivity with fast performance, we also acknowledge opportunities for improved performance. We highlight ways in which we expect future profile HMM alignment tools to improve on the value of Cloud Search and downstream sparse analysis.

Multiple domains

The current MMOREseqs implementation depends on MMseqs2 as a source of seeds for Cloud Search, and in particular captures only the single best alignment produced by MMseqs2. In cases where there are multiple regions of high-quality alignment (e.g. multiple copies of a domain, or highly fragmented sequence match), MMOREseqs will often capture only a single aligned region. This will often simply fail to identify multiple matches, but in some cases, an unfortunate MMseqs2 seed can mean that the best matching alignment is missed by MMOREseqs (as in Fig **??**). Mechanisms for identifying multiple good begin/end seeds will improve the completeness and sensitivity of analyses based on Cloud Search and downstream processing.

Improved pipeline integration

As implemented, MMOREseqs is decidedly modular, with a burdensome interface and format translation stage between the early MMseqs2 analysis stage and downstream Cloud Search and Forward/Backward analysis. Obvious paths toward improved integration include (i) remaining

tied to the MMseqs2 HMM format and (ii) porting the Cloud Search approach in to the HMMER code base. In either case, further development calls for exploration of preferred methods for seeding the Cloud Search stage; for example, is it really best to depend on a Viterbi prefilter and alignment step, or is it effective to establish Cloud Search begin/end points based on simple k-mer matches?

Faster computations

The Forward/Backward recurrence calculations are modeled after the generic implementation in HMMER, with significant overhead require to support movement back and forth to log-scaled representations of odds ratios. Dynamic scaling in probability space is faster [13] and should be feasible in the sparse representation described here.

BIBLIOGRAPHY

[1] T. F. Smith, M. S. Waterman *et al.*, "Identification of common molecular subsequences," *Journal of molecular biology*, vol. 147, no. 1, pp. 195–197, 1981.

[2] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *Journal of molecular biology*, vol. 215, no. 3, pp. 403–410, 1990.

[3] A. Krogh, M. Brown, I. S. Mian, K. Sjölander, and D. Haussler, "Hidden markov models in computational biology: Applications to protein modeling," *Journal of molecular biology*, vol. 235, no. 5, pp. 1501–1531, 1994.

[4] K. Karplus, C. Barrett, and R. Hughey, "Hidden markov models for detecting remote protein homologies." *Bioinformatics (Oxford, England)*, vol. 14, no. 10, pp. 846–856, 1998.

[5] R. Durbin, S. R. Eddy, A. Krogh, and G. Mitchison, *Biological sequence analysis: probabilistic models of proteins and nucleic acids*. Cambridge university press, 1998.

[6] S. R. Eddy, "Profile hidden markov models." *Bioinformatics (Oxford, England)*, vol. 14, no. 9, pp. 755–763, 1998.

[7] M. Gribskov, A. D. McLachlan, and D. Eisenberg, "Profile analysis: detection of distantly related proteins," *Proceedings of the National Academy of Sciences*, vol. 84, no. 13, pp. 4355–4358, 1987.

[8] J. Mistry, S. Chuguransky, L. Williams, M. Qureshi, G. A. Salazar, E. L. Sonnhammer, S. C. Tosatto, L. Paladin, S. Raj, L. J. Richardson *et al.*, "Pfam: The protein families database in 2021," *Nucleic Acids Research*, vol. 49, no. D1, pp. D412–D419, 2021.

[9] H. Mi, A. Muruganujan, D. Ebert, X. Huang, and P. D. Thomas, "Panther version 14: more genomes, a new panther go-slim and improvements in enrichment analysis tools," *Nucleic acids research*, vol. 47, no. D1, pp. D419–D426, 2019.

[10] M. K. Gibson, K. J. Forsberg, and G. Dantas, "Improved annotation of antibiotic resistance determinants reveals microbial resistomes cluster by ecology," *The ISME journal*, vol. 9, no. 1, pp. 207–216, 2015.

[11] A. L. Grazziotin, E. V. Koonin, and D. M. Kristensen, "Prokaryotic virus orthologous groups (pvogs): a resource for comparative genomics and protein family annotation," *Nucleic acids research*, p. gkw975, 2016.

[12] J. Storer, R. Hubley, J. Rosen, T. J. Wheeler, and A. F. Smit, "The dfam community resource of transposable element families, sequence models, and genome annotations," *Mobile DNA*, vol. 12, no. 1, pp. 1–14, 2021.

[13] S. R. Eddy, "Accelerated profile hmm searches," *PLoS Comput Biol*, vol. 7, no. 10, p. e1002195, 2011.

[14] T. J. Wheeler and S. R. Eddy, "nhmmer: Dna homology search with profile hmms," *Bioinformatics*, vol. 29, no. 19, pp. 2487–2489, 2013.

[15] I. Holmes and R. Durbin, "Dynamic programming alignment accuracy," *Journal of computational biology*, vol. 5, no. 3, pp. 493–504, 1998.

[16] C. B. Do, M. S. Mahabhashyam, M. Brudno, and S. Batzoglou, "Probcons: Probabilistic consistency-based multiple sequence alignment," *Genome research*, vol. 15, no. 2, pp. 330–340, 2005.

[17] M. Steinegger and J. Söding, "Mmseqs2 enables sensitive protein sequence searching for the analysis of massive data sets," *Nature biotechnology*, vol. 35, no. 11, pp. 1026–1028, 2017.

[18] M. Mirdita, M. Steinegger, and J. Söding, "Mmseqs2 desktop and local web server app for fast, interactive sequence searches," *Bioinformatics*, vol. 35, no. 16, pp. 2856–2858, 2019.

[19] S. R. Eddy, "A probabilistic model of local sequence alignment that simplifies statistical significance estimation," *PLoS Comput Biol*, vol. 4, no. 5, p. e1000069, 2008.

[20] M. Farrar, "Striped smith–waterman speeds database searches six times over other simd implementations," *Bioinformatics*, vol. 23, no. 2, pp. 156–161, 2007.

[21] S. Eddy, "Easel - a c library for biological sequence analysis." [Online]. Available: http://bioeasel.org

[22] Y. Mori, "libdivsufsort." [Online]. Available: https://github.com/y-256/libdivsufsort