Supplemental material for the paper: Significant speedup of database searches with HMMs by search space reduction with PSSM family models

Michael Beckstette Robert Homann Robert Giegerich
Stefan Kurtz

1 An efficient chaining algorithm for the PSSM matches chaining problem

This section describes a specialized and improved algorithm for solving the chaining problem for PSSM matches on a set of database sequences $S_1, \ldots, S_k$. The latter are the sequences for which the enhanced suffix array was constructed. Note that k can be in the range of millions, if one, e.g., matches against the complete UniProtKB/TrEMBL database. For a given PSSM family model $\mathcal{M} = M_1, M_2, \ldots, M_L$, all M_i, $1 \leq i \leq L$, are matched one after the other against the enhanced suffix array. This gives match sets $\mathcal{MS}(M_i)$ for PSSM M_i. L is in the range of tens while the number of PSSM matches for a particular sequence S_j is in the order of hundreds if S_j is a protein sequence. For each match f the following information is recorded:

- The ordinal number i of the PSSM M_i involved in the match f. This is denoted by $f.pssm$.
- The length of the PSSM involved in the match f. This is denoted by $f.length$.
- The number j of the sequence S_j the match f occurs in. This is denoted by $f.seqnum$.
- The starting position of the match f in sequence S_j. This is denoted by $f.pos$.
- The weight $\alpha(M_{f.pssm}, s)$ of the match f, where s is the score of the match. The weight of f is denoted by $f.weight$.

In an initial sorting step the union $\mathcal{MS}$ of all match sets $\mathcal{MS}(M_i)$, $1 \leq i \leq L$, is sorted in ascending order of $f.seqnum$. Matches with identical sequence numbers are sorted in ascending order of the ordinal number of the PSSM, i.e., by $f.pssm$. Suppose that b^* is the size of $\mathcal{MS}$. As there are at most b^* sequences with at least one PSSM match, the

sorting according to the sequence numbers can be done in $O(k^* + b^*)$ time and $O(k^*)$ space using the counting sort algorithm [1]. Here, k^* is the number of sequences with at least one PSSM match. As $k^* \leq b^*$, the sorting requires $O(b^*)$ time and space. We obtain disjoint subsets $\mathcal{MS}(S_j)$, $1 \leq j \leq k$, where $\mathcal{MS}(S_j)$ is the set of all matches in $\mathcal{MS}$ matching a substring of S_j. As $\mathcal{MS}$ is ordered by the ordinal number of the PSSM and the counting sort algorithm is stable, the sets $\mathcal{MS}(S_j)$ are also sorted by the ordinal number of the PSSMs. Let $\mathcal{MS}(S_j, M_i)$ denote the matches $f \in \mathcal{MS}(S_j)$ such that $f.pssm = i$. In a second sorting step, each $\mathcal{MS}(S_j, M_i)$ is sorted according to the starting position of the matches. As this is a typical integer sorting problem, it requires $O(b_{j,i} \log b_{j,i})$ time, where $b_{j,i}$ is the size of $\mathcal{MS}(S_j, M_i)$. Altogether, the two initial sorting steps can be performed in $O(b^* + \sum_{j=1}^{k} \sum_{i=1}^{L} b_{j,i} \log b_{j,i})$ time.

For all $S_1, S_2, \ldots, S_k$ one now solves independent chaining problems for sets $\mathcal{MS}(S_j)$, $1 \leq j \leq k$, of matches sorted according to the ordinal number of the PSSM and the starting position of the matches in S_j. Let j be fixed, but arbitrary. For each match $f \in \mathcal{MS}(S_j)$, the weight $f.weight$ is positive. Hence, an optimal chain ends with a match f such that there is no match f' satisfying $f \ll f'$. Similarly, an optimal chain begins with a match f' such that there is no match f satisfying $f \ll f'$.

The chaining problem is solved by a dynamic programming algorithm which tabulates for all matches $f' \in \mathcal{MS}(S_j)$ the maximum score $f'.score$ of all chains ending with f'. In addition, it computes the predecessor $f'.prec$ of f' in a chain with maximum score ending with f'. To obtain $f'.score$, one has to maximize over all matches f such that $f.pssm < f'.pssm$ and $f.pos + f.length - 1 < f'.pos$. This is a two dimensional search problem. As the matches in $\mathcal{MS}(S_j)$ are already sorted according to the first dimension (i.e., by the ordinal number of the PSSM), one can reduce it to a one dimensional sorting problem. This has already been observed [2], and led to the development of an algorithm solving the chaining problem in $O(b \log b)$, where b is the number of matches in $\mathcal{MS}(S_j)$. We follow the basic structure of this algorithm. However, the algorithm of [2] was developed for chaining pairwise sequence matches. The PSSM chaining problem is a special instance of this problem: the first "sequence" consists of the positions $1, \ldots, L$, and a match for PSSM M_i is a match of length one to position i. Moreover, all matches at position i in the first sequence are of equal length because they are matches to the same PSSM M_i of identical length. In addition to this, our initial sorting step delivers, for all i, $1 \leq i \leq L$, the matches in $\mathcal{MS}(S_j, M_i)$ in sorted order according to the starting position in S_j. All these properties allow to simplify and improve the algorithm of [2] in the following aspects:

- While the algorithm of [2] requires a dictionary data structure with insert, delete, predecessor, and successor operations running in logarithmic time (e.g., an AVL-tree or a red-black tree [1]), our approach only needs a linear list, which is much easier to implement and requires less space.

- While the algorithm of [2] requires an initial sorting step using $O(b^* \log b^*)$ time, our method only needs $O(b^* + \sum_{j=1}^{k} \sum_{i=1}^{L} b_{j,i} \log b_{j,i})$ time for this step. Note that the $b_{j,i}$ satisfy $\sum_{j=1}^{k} \sum_{i=1}^{L} b_{j,i} = b^*$.

- While the algorithm of [2] solves the chaining problem for $\mathcal{MS}(S_j)$ in $O(b \log b)$ time, our approach runs in $O(b \cdot L)$ time. If L is considered to be a constant, the running time becomes linear in b, where $b = |\mathcal{MS}(S_j)|$.

To explain our algorithm, let i, $1 \leq i \leq L$ be arbitrary but fixed and assume that all match sets $\mathcal{MS}(S_j, M_{i'})$, $i' < i$ have been processed. In a first loop over the sorted matches in $\mathcal{MS}(S_j, M_i)$ one determines the score of the matches. In a second loop, one inserts them into a linear list if necessary. The linear list contains a subset of the previously processed and scored matches. This split of the computation into two loops is different from the algorithm of [2] where the scoring and insertions are interweaved in one loop, requiring an extra array of length $2b$ containing references to the matches. The separation into two loops allows us to get rid of this extra array.

Now consider the first loop over all elements in $\mathcal{MS}(S_j, M_i)$ in sorted order of the match position in S_j. Let f' be the current element. At this point, all matches f such that $f.pssm < f'.pssm$ have been processed already. In particular, the score $f.score$ and the previous match (if any) in an optimal chain ending with f has been determined. Among the processed matches we only have to consider those matches f satisfying $f.pos + f.length - 1 < f'.pos$. If there is such a match, one takes the one with maximal score, say f. Then, the optimal chain ending with f' contains the previous match f, and the score is $f'.score = f'.weight + f.score$. If there is no such match, then the optimal chain ending with f' only consists of f' and $f'.score = f'.weight$.

Now consider the second loop over all elements in $\mathcal{MS}(S_j, M_i)$ for which the scores and predecessor matches (if any) are already determined. Let f' be the current element to be inserted. As explained in the previous case, one has to make sure that, among the processed matches, one can efficiently determine the match f with the maximum score such that $f.pos + f.length - 1$ is smaller than some value depending on f'. The processed matches are stored in a linear list which is sorted in ascending order of the position of the matches in S_j. Let $\prec_{pos}$ denote this order, that is $f \prec_{pos} f''$ if and only if $f.pos + f.length < f''.pos + f''.length$ for any matches f and f''. If for two processed matches f and f'' one has $f.pos < f''.pos$ and $f.score > f''.score$, then an optimal chain does not include f''. Each chain that uses f'' can also use f and increase the chain score. As a consequence, one has to take care that f'' is not inserted into the linear list or it is deleted if it was inserted earlier. In this way, $f \prec_{pos} f''$ always implies $f.score \leq f''.score$ for two matches f and f'' in the linear list. As the elements to be scored in the first loop and to be inserted in the second loop are ordered in the same way as the elements in the linear list, one can perform the scoring and the insertion loop (which also may involve deletions) by merging two lists of length l_1 and l_2 in $O(l_1 + l_2)$ time where l_1 is the number of matches to be scored and inserted and l_2 is the length of linear list involved. Let $b = |\mathcal{MS}(S_j)|$. As $l_1 + l_2 \leq b$, one obtains a running time of $O(b)$ for each set $\mathcal{MS}(S_j, M_i)$. As there are L such sets, the running time is $O(b \cdot L)$.

References

[1] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.

[2] M.I. Abouelhoda and E. Ohlebusch. Chaining algorithms for multiple genome comparison. *J. Discrete Algorithms*, 3(2-4):321–341, 2005.

Algorithm 1: Compute skip table in $O(n)$ time.

input : The longest common prefix table lcp with $n+1$ entries for some text of length n.
output: The corresponding skip table skp with $n+1$ entries.

```
1  push(suffixes,0);
2  push(depths,0);
3  for i ← 1,...,n do
4  |  while top(depths) > lcp[i] do
5  |  |  skp[pop(suffixes)] ← i;
6  |  |  pop(depths);
7  |  if lcp[i] > 0 then push(depths,lcp[i]);
8  |  else skp[pop(suffixes)] ← n+1;
9  |  push(suffixes,i);
10 skp[pop(suffixes)] ← n+1;
```

Figure 1: Linear time computation of table skp using information from table lcp. The algorithm uses two stacks, one stack depths holding the lcp-values from table lcp, and one stack suffixes holding indexes of the suffix array suf. Both stacks are used synchronously, i.e., their depths are the same after each iteration. There are at most n push-operations on the stacks. Each iteration of the inner while-loop involves a pop-operation on the non-empty stacks. Hence, the overall running time of the inner while-loop is bounded by n. The rest of the for-loop also runs in linear time. Hence, the algorithm requires linear time.

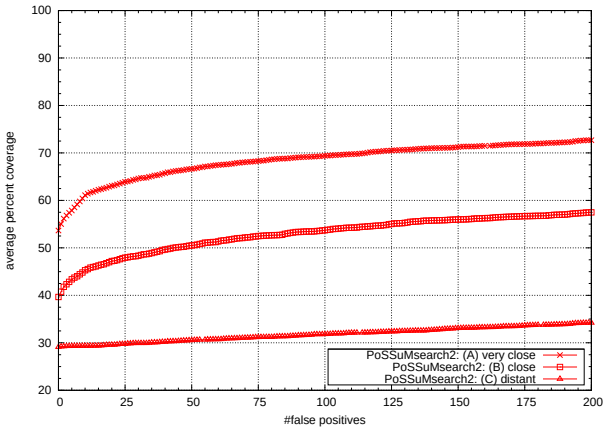

Figure 2: Classification performance of *PoSSuMsearch2* in the three described evaluation scenarios when using a naive chainscore function that simply computes the product of raw p-values without further normalization. Shown are percentage true positive values averaged over all test families (coverage) (y-axis) for different numbers of accepted false positives (x-axis). Compared to the results of Figure 4 of the main document it is obvious that the naive chainscore performs significantly worse than the default chain score defined in Equation 2.

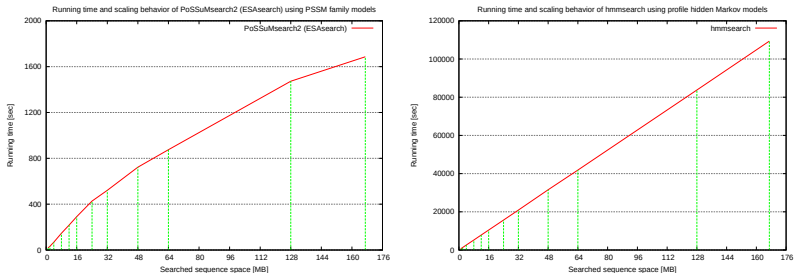

Figure 3: Runtimes in seconds and scaling behavior of *PoSSuMsearch2* (left) and *hmmsearch* (right) when searching for the first 100 *PFAM* protein families in subsets of UniProtKB/Swiss-Prot of increasing sizes.

	$k=2$		$k=3$	$k=4$	$k=5$
	π_{tc}	π_{nc}	π_{tc}	π_{tc}	π_{tc}
Running time *hmmsearch* [min]	65,283.2	65289.7	65,283.2	65,283.2	65,283.2
Running time *PSfamSearch* [min]	1,490.0	4676.0	1,340.4	942.8	966.4
Total speedup	43.8	14.0	48.7	69.2	67.6
Speedup max.	5,176.6	1765.1	7,945.8	8,302.1	9,260.0
Speedup min.	2.6	1.1	1.9	1.9	1.9
Number of sequences passing filter step	0.80%	3.83%	1.14%	0.67%	0.87%
Average percentage true positives	99.54%	99.47%	99.21%	99.05%	98.43%
Minimum percentage true positives	97.50%	95.65%	96.78%	97.06%	95.43%

Table 1: Speedups and percentage true positive values (coverage) obtained by *PSfamSearch* compared to *hmmsearch* when searching with PSSM family models corresponding to the first 200 TIGRFAM models using π_{tc} (π_{nc}, for $k=2$ only) in the complete UniProtKB/TrEMBL database. Measurements were obtained for different values of k corresponding to training sets containing 50, 33, 25, and 20 percent of all family members detected by *hmmsearch*. For details, see corresponding part in the main document. Row 4 (5) gives the maximum (minimum) speedup obtained for a particular model.

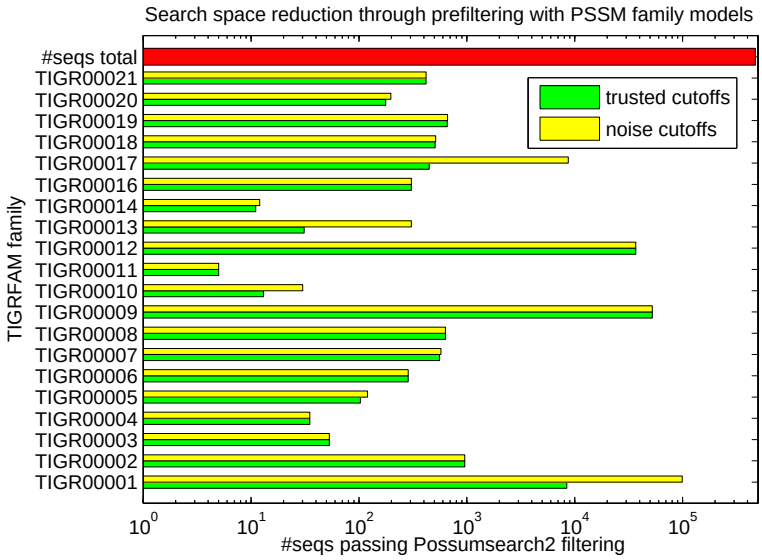

Figure 4: Search space reduction through PSSM family model pre-filtering. We measured the number of sequences passing pre-filtering of the search space with *PoSSuMsearch2* (x-axis, logscale) with p-value cutoffs π_{tc} (dark) and π_{nc} (light) for the first 20 protein families of the TIGRFAM database (Rel. 8.0). The bar on top shows the total number of sequences in UniProtKB/Swiss-Prot release 57.5 (471,472 protein sequences with a total length of $\sim$ 167MB) that needed to be searched by direct *hmmsearch* without filtering. Thus the search space is reduced by at least one order of magnitude. In some cases the reduction is by several orders of magnitude.

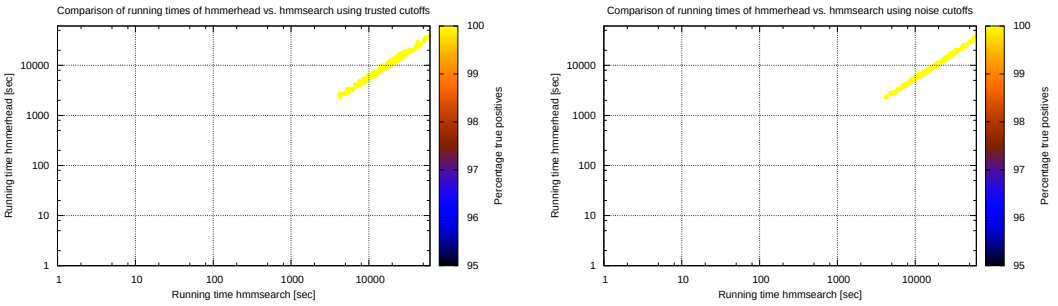

Figure 5: Comparison of running times of *HMMERHEAD* (version 0.6) (y-axis, logscale) and *hmmsearch* (x-axis, logscale), and achieved coverage when searching with the first 200 TIGRFAM models on UniProtKB/TrEMBL using trusted cutoffs (left) and noise cutoffs (right), respectively. Obtained speedups per model were in the range between 1.46 and 1.91 with an average of 1.73 when using trusted cutoffs, and between 1.55 and 1.93 with an average of 1.79 when using noise cutoffs.

TIGR family	#seqs in red. Space	% of total seq. space	min. chain length	#found	#missed	found[%]	missed[%]
TIGR00001	29,729	0.37	2	602	2	99.67	0.33
TIGR00002	78,870	0.99	3	868	1	99.88	0.12
TIGR00003	27,754	0.35	2	454	5	98.91	1.09
TIGR00004	1796	0.02	7	1,274	1	99.92	0.08
TIGR00005	3,962	0.05	8	2,448	1	99.96	0.04
TIGR00006	1,068	0.01	8	1,067	3	99.72	0.28
TIGR00007	468	0.005	8	413	1	99.76	0.24
TIGR00008	1,118	0.01	3	885	5	99.44	0.56
TIGR00009	634,977	8.02	2	627	4	99.37	0.63
TIGR00010	1,788	0.02	8	1,265	1	99.92	0.08
TIGR00011	711	0.009	8	625	3	99.52	0.48
TIGR00012	1,274,316	16.09	2	919	1	99.89	0.11
TIGR00013	13,465	0.17	3	329	6	98.21	1.79
TIGR00014	1,168	0.01	5	690	1	99.86	0.14
TIGR00016	1,089	0.01	8	1,070	8	99.26	0.74
TIGR00017	740	0.009	8	716	0	100.00	0.00
TIGR00018	538	0.006	8	531	11	97.97	2.03
TIGR00019	666	0.008	8	644	3	99.54	0.46
TIGR00020	999	0.01	8	986	3	99.70	0.30
TIGR00021	668	0.008	8	646	2	99.69	0.31
Average:	103,794.5	1.31	5.85	$\sum$ 17,059	$\sum$ 62	99.51	0.49

Table 2: Results of p-value cutoff calibration based on *hmmsearch* matches obtained on UniProtKB/TrEMBL using trusted cutoffs. Cutoffs were calibrated such that half of the sequences (training set) pass *PoSSuMsearch2* filtering. Columns 2 and 3 give the absolute number and percentage of sequences passing the filter. Numbers of found and missed family sequences on complete UniProtKB/TrEMBL are given in columns 5 and 6.

TIGR family	#seqs in red. Space	% of total seq. space	min. chain length	#found	#missed	found[%]	missed[%]
TIGR00001	549,709	6.94	2	657	5	99.24	0.76
TIGR00002	109,392	1.38	3	875	1	99.89	0.11
TIGR00003	42,176	0.53	3	478	3	99.38	0.62
TIGR00004	31,718	0.40	5	1,592	1	99.94	0.06
TIGR00005	32,143	0.40	6	3,545	0	100.00	0.00
TIGR00006	1,086	0.013	8	1,079	3	99.72	0.28
TIGR00007	506	0.006	8	505	2	99.60	0.40
TIGR00008	1,118	0.014	3	890	8	99.11	0.89
TIGR00009	634,977	8.02	2	640	4	99.38	0.62
TIGR00010	3,314	0.04	6	2,157	1	99.95	0.05
TIGR00011	80,265	1.01	3	778	2	99.74	0.26
TIGR00012	1,274,316	16.09	2	972	1	99.90	0.10
TIGR00013	513,962	6.49	2	659	2	99.70	0.30
TIGR00014	1,847	0.02	5	768	5	99.35	0.65
TIGR00016	1,163	0.01	8	1,166	3	99.74	0.26
TIGR00017	7,001	0.08	6	754	5	99.34	0.66
TIGR00018	1,151	0.01	5	625	3	99.52	0.48
TIGR00019	733	0.009	8	711	5	99.30	0.70
TIGR00020	1,092	0.01	8	1,029	4	99.61	0.39
TIGR00021	69,621	0.87	5	680	2	99.71	0.29
Average:	167,864.5	2.12	4.9	$\sum$ 20,560	$\sum$ 60	99.61	0.39

Table 3: Results of p-value cutoff calibration based on *hmmsearch* matches obtained on UniProtKB/TrEMBL using noise cutoffs. Cutoffs were calibrated such that half of the sequences (training set) pass *PoSSuMsearch2* filtering. Columns 2 and 3 give the absolute number and percentage of sequences passing the filter. Numbers of found and missed family sequences on complete UniProtKB/TrEMBL are given in columns 5 and 6.

Strain	Refseq accession
Escherichia coli 536	NC_008253.1
Escherichia coli 55989	NC_011748.1
Escherichia coli APEC O1	NC_008563.1
Escherichia coli ATCC 8739	NC_010468.1
Escherichia coli B str. REL606	NC_012967.1
Escherichia coli BL21	NC_012892.1
Escherichia coli BL21(DE3)	NC_012947.1
Escherichia coli BW2952	NC_012759.1
Escherichia coli CFT073	NC_004431.1
Escherichia coli E24377A	NC_009801.1
Escherichia coli ED1a	NC_011745.1
Escherichia coli HS	NC_009800.1
Escherichia coli IAI1	NC_011741.1
Escherichia coli IAI39	NC_011750.1
Escherichia coli O127:H6 str. E2348/69	NC_011601.1
Escherichia coli O157:H7 EDL933	NC_002655.2
Escherichia coli O157:H7 str. EC4115	NC_011353.1
Escherichia coli O157:H7 str. Sakai	NC_002695.1
Escherichia coli O157:H7 str. TW14359	NC_013008.1
Escherichia coli S88	NC_011742.1
Escherichia coli SE11	NC_011415.1
Escherichia coli SMS-3-5	NC_010498.1
Escherichia coli UTI89	NC_007946.1
Escherichia coli str. K-12 substr. DH10B	NC_010473.1
Escherichia coli str. K-12 substr. MG1655	NC_000913.2
Escherichia coli str. K-12 substr. W3110	AC_000091.1

Table 4: Strains and Refseq accession numbers of 26 publicly available *E. coli* proteomes used in the whole proteome annotation experiment.

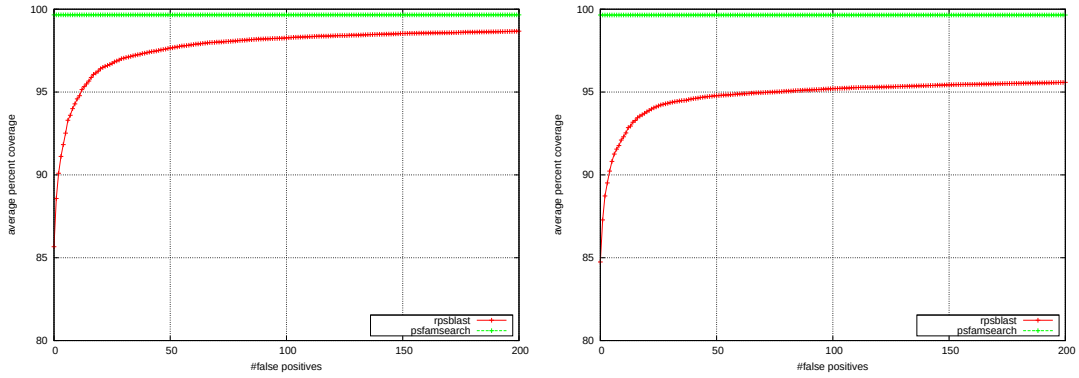

Figure 6: Classification performance of *rpsblast* and *PSfamSearch* with respect to *hmmsearch* when using trusted cutoffs (left) and noise cutoffs (right), respectively. In this experiment we determined the percentage true positives averaged over the first 200 TIGRFAM families (y-axis) when accepting a certain number of false positives (x-axis) in a database search on complete UniProtKB/TrEMBL. For *rpsblast*, which was run with default parameters, we used models from the CDD database Rel. 2.17 corresponding to these families. As 'state of truth' we used the results obtained from *hmmsearch*.