Sequence Alignment as a Database Technology Challenge

Hans Philippi

Dept. of Computing and Information Sciences
Utrecht University
hansp@cs.uu.nl
http://www.cs.uu.nl/people/hansp

Abstract. Sequence alignment is an important task for molecular biologists. Because alignment basically deals with approximate string matching on large biological sequence collections, it is both data intensive and computationally complex. There exist several tools for the variety of problems related to sequence alignment. Our first observation is that the term 'sequence database' is used in general for textually formatted string collections. A second observation is that the search tools are specifically dedicated to a single problem. They have limited capabilities to serve as a solution for related problems that require minor adaptations. Our aim is to show the possibilities and advantages of a DBMS-based approach toward sequence alignment. For this purpose, we will adopt techniques from single sequence alignment to speed up multiple sequence alignment. We will show how the problem of matching a protein string family against a large protein string database can be tackled with q-gram indexing techniques based on relational database technology. The use of Monet, a main-memory DBMS, allows us to realize a flexible environment for developing searching heuristics that outperform classical dynamic programming, while keeping up satisfying sensitivity figures.

1 Introduction

There is no doubt about the importance of sequence alignment for molecular biologists. Homology searching comes down to matching a specific string, the query, to a large collection of already known strings, the database. The database can either contain nucleotide strings, based on the ACGT-alphabet or amino acid strings, based on a twenty letter alphabet. Evolutionary changes force us to deal with inexact matching. So essentially we are talking about approximate string matching on large string collections.

Traditionally, sequence databases have a pure textual format. The actual string contents are mixed with identifiers and annotation. Moreover, dedicated tools like Blast ([3], [4], [1]) and HMMER ([9]) are used for searching. In other words, if a DBMS is used at all, it is only used as a storage engine. So, the challenge for the database community is to show that the query facilities of a DBMS can simplify searching and make it more flexible.

R. Wagner, N. Revell, and G. Pernul (Eds.): DEXA 2007, LNCS 4653, pp. 459–468, 2007.

Single sequence alignment, i.e. the matching of one query string to a large string collection, has been exhaustively investigated ([5]). The exact solution is provided by a dynamic programming algorithm (Smith-Waterman). The need for quick response led to the development of Blast, a heuristic alignment tool based on q-gram indexing.

The q-gram indexing techniques on which the Blast heuristics are based, can be easily translated to a relational database environment. The indexing is realized at the logical level: the q-gram set is added as a table. The filtering process based on q-gram indexing can be concisely expressed in relational algebra. Variations in the filtering heuristics can be investigated by minor changes in the query expression.

To illustrate the versatility of the DBMS approach, we will focus on multiple sequence alignment. The notion of multiple sequence alignment deals with matching a collection of related protein strings (a so called family) to a database. This collection can be represented by a Hidden Markov Model (HMM). Models representing a protein string collection are known as profile HMM's. Matching a profile HMM to a protein string database is generally solved using Viterbi-like dynamic programming principles. The HMMER-package by Sean Eddy ([9]) is a freely available, open source implementation of these techniques. A typical matching operation using HMMER with a medium size database will take several minutes on commodity hardware. The answer is exact, in the sense that it finds all matches within some similarity distance.

In this paper, we will describe a generalized, Blast-like, heuristic search method based on q-gram indexing. Adapting these ideas to the context of multiple alignment turns out to be surprisingly straightforward, due to the flexibility that our DBMS provides. This way, our database supports both single string queries and family queries. The implementation of our methods on the Monet main-memory DBMS enables us to reduce the reponse time compared to HMMER significantly, while keeping up satisfactory sensitivity figures.

2 Preliminaries

In this section, we will introduce the concepts needed to discuss the domain of protein sequence alignment.

2.1 Strings

Our basic objects of interest are strings and q-grams. We will define them here.

- A *string* is a mapping from an integer interval $[k..n]$ to the set of characters. We have the notion of substring. Our alphabet is limited to the twenty amino acid symbols.
- A *q-gram* is a string of length q, a fixed number that typically is 3 for protein databases. We will use the value $q = 3$ in our examples. The term *word* is a synonym for q-gram and more common in the Blast community.

- A *position specific q-gram* is a combination of a q-gram and the position in the string where it refers to. Example: in the string ACDEG, with starting position 1, we identify the position specific q-grams $(1, ACD), (2, CDE)$, $(3, DEG)$.
- Basically, our *database* is a set of strings which all have $k = 1$.
- A *hitlist* is a set of position specific q-grams.

2.2 A Relational View on Q-Gram Indexing

In general, sequence databases are files in a character based format, like the Fasta format. In Fasta collections, the strings are listed interleaved with their annotation. We represent a string collection by two tables: `Strings` and `Annots`.

```
Strings(id, string)
Annots(id, annot)
```

Strings are internally identified with a system generated `id`. It maintains the connection between the protein strings, the annotation strings and the q-grams. So for each string in `Strings`, we have a describing tuple in `Annots`.

The `Q-grams` table contains a set of position specific q-grams. It serves as the index for matching hits between query data and the string database. Note that, on the physical level, we do not make use of either traditional indexing support or specialized string indexing techniques, such as suffix trees. We rely on the power of our main-memory DBMS to process the queries efficiently.

```
Q-grams(id,j,qg)
```

The position of the q-grams in the strings is denoted by a j in the database strings and an i in the query. The annotation table joins in (literally) at a very late stage.

BLASTP works as follows. A query string is, like the database strings, decomposed into q-grams. Suppose we have the query string denoted by `qs` and a database string by `s`.

```
qs =  CWYWRWYY
 s = RRWYWAWYYRR
```

In Table 1, we see a q-gram decomposition of the query string. We see a few q-gram matches between `qs` and `s`. $(2, WYW)$ in `qs` matches $(3, WYW)$ in `s`; $(6, WYY)$ in `qs` matches $(7, WYY)$ in `s`.

Because the distance between the matching q-grams is equal in the two strings, we say that they are 'on the same diagonal'. Technically, the notion of diagonal is represented by the difference of the q-gram positions: 6-2 = 7-3. The essence of the BLASTP filtering approach comes down to looking for two non-overlapping q-gram hits on the same diagonal within a certain distance (default 40).

To improve sensitivity for BLASTP, we also have the notion of 'similar' q-grams. The q-grams $(1, CWY)$ in `qs` and $(2, RWY)$ in `s` will generally be identified as similar, due to the notion of evolutionary distance (see [2] for further

Table 1. Example database and query

Strings	
id	*string*
1	RRWYWAWYYRR
2	RRRWYWAWYWRR
3	RRWYWAAWYYRR

Annots	
id	*annot*
1	comments on string 1 ..
2	comments on string 2 ..
3	comments on string 3 ..

Qgrams		
id	*j*	*qg*
1	1	RRW
1	2	RWY
1	3	WYW
...	...	...
3	10	YRR

Query	
i	*qg*
1	CWY
2	WYW
...	...
6	WYY

details). This means that we should extend the basic q-gram set of a query string with similar q-grams.

Our query string defines a hitlist containing, among others, $(1, CWY)$, $(1, RWY)$, $(2, WYW)$ and $(6, WYY)$. It depends on the parameter settings which similar q-grams will show up in our hitlist.

2.3 Profile HMM Matching

We now direct our attention toward the problem of matching a family of related sequences to a string database. A profile HMM ([6]) is a probabilistic model that represents a collection of related protein strings, often called a family. Figure 1 shows the basic HMM-architecture as used in the HMMER package ([9]). By

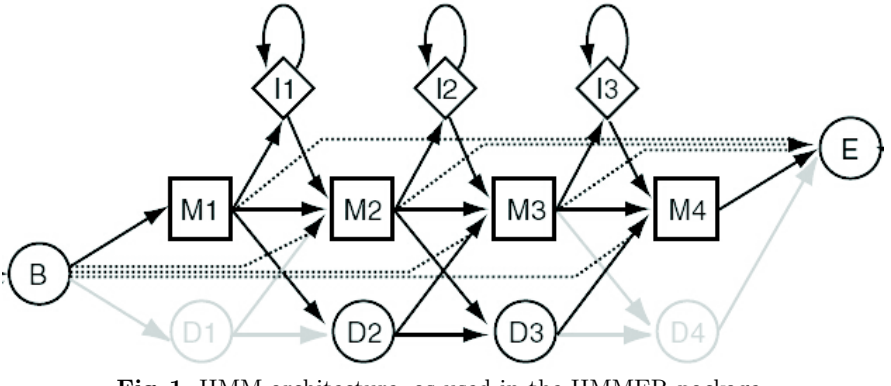

Fig. 1. HMM-architecture, as used in the HMMER package

matching a string to a HMM, we get a quantitative expression for the level of 'relatedness' between this string and the family that is represented by this model. Algorithms exist to calculate the optimal matching. They are based on dynamic programming techniques ([2], [5]). Note that these algorithms are exact: they find all matches with at least a specified minimal similarity.

The optimal match of a string to an HMM is represented by a path through the model. The begin and end-state are straightforward. The M-states represent a match. For each possible character, it specifies the 'emission probability', i.e. the probability of finding this character on this position. The I-states and D-states represent gaps in the match. By adding the rewards for matching and the penalties for mismatching, we get a final value expressing the relatedness.

To get a feeling for the principle, let us take a look at this small fraction of a four string family. The '-' represents a gap.

```
AFVEFEDP
GFVEFEDY
AFV-FEDP
AFVRF-DK
```

Because of these strings have length eight, we need an HMM with eight matching states. Matching state $M1$ (corresponding to the first column in the family) emits character `A` with probability 3/4 and `G` with probability 1/4. State $M2$ emits `F` with probability 1: there is 'consensus', just as in columns 3, 5, 6 and 7. State $M4$ emits character `E` with probability 2/3 and `R` with probability 1/3. State $M8$ emits `P` with probability 1/2 and emits `Y` and `K` with probability 1/4. Note that, if we would match the last string to this HMM, the optimal path for traversing the model would go from state $M5$ to state $M7$ through state $D6$, resulting in a gap in position 6.

In the model, these probabilities are transformed to the log-odds of the emmission probabilities, according to the random amino acid distribution model. This log-odds conversion enables us to transform multiplication of probabilities into additions. States $D1$ and $D4$ are optional, depending on the choice to match globally (i.e. matching the whole model) or locally (i.e. matching the model partially).

A profile HMM defines a hitlist in a straightforward way. For each position we inspect the corresponding matching state with its characters and corresponding log-odds values. The most extended hitlist generated from our example family would be
$(1, AFV), (1, GFV), (2, FVE), (2, FVR), (3, VEF), (3, VRF), ..., (6, EDP)$.

We will use limited HMM-hitlists for filtering purposes, thereby focussing on position-character combinations with high probability. This choice is made at the character level, taking into consideration only the characters that have position specific emmission log-odds values meeting a threshold value T.

Summarizing, we see that the q-gram indexing principles can be extended from single protein query strings to profile HMM's. Both can serve as a query, because from a technical point of view, the matching object is in both cases a hitlist.

3 Filtering

As we have seen, the classical BLASTP approach uses the two-hit-diagonal filtering principle. Thresholds to limit the hitlist can, to some extend, be set by the user, influencing sensitivity and selectivity. In the case of profile HMM matching, we have added a tuning parameter n, making the number of required hits on the diagonal a user defined variable. It is clear that by increasing n, we increase selectivity and decrease sensitivity. Note that the principle of n-hit diagonal filtering corresponds to the *framecount* notion of CAFE ([11]).

We will describe how n-hit diagonal filtering can be expressed as a relational query. We choose to formulate the constituent expressions in an extended version of the relational algebra (RA). See [10] for details.

Note that the query, essentially a hitlist, might represent both a single string and a profile HMM. A hit represents an exact match between a q-gram in the query and a q-gram in the database. Recall that in the hitlist, at the same position, more than one (similar) q-gram may occur.

$$Hits := \pi_{id,diag \leftarrow (j-i)}(Qgrams \bowtie Query) \tag{1}$$

According to the n-hit diagonal filtering method, candidates are defined by n hits in the same string and on the same diagonal. We first apply a self join on $Hits$ to find hit pairs on the same diagonal. We need a copy of the hits table to express this self join. Hits2 should be interpreted as an alias of (or view on) the $Hits$ table, not as a physical copy.

$$Hits2 := \pi_{id2,i2,j2,diag2}(Hits) \tag{2}$$

$$Pairs := \pi_{id,diag,j,j2}(Hits \bowtie_\theta Hits2) \tag{3}$$

where θ denotes the join condition:

$$\theta : (id = id2, diag = diag2, |j - j2| \leq A)$$

Here A denotes the range, i.e. the maximal distance between two hits. Note that, en passant, we have by now expressed BLASTP filtering. Enforcing the n-diagonal is expressed by the grouping operator $\Gamma_{grp;function}$ of the relational algebra.

$$Filter := \sigma_{cnt \geq (n-1)}(\Gamma_{id,diag,j;cnt()}(Pairs) \tag{4}$$

Note that we select on $n-1$ values within the group because j itself is already one hit. The relevant strings can now be selected.

$$Candidates := Strings \ltimes Filter \tag{5}$$

This approach can be refined by taking in account the actual hit positions in the database and query string. That allows us to define a substring as a candidate for matching in stead of the complete string. Defining the boundaries however is quite tricky, because the HMM allows gapping. We will stick to the approach of pure string filtering as defined by the last step.

4 The Monet Approach

Our DBMS of choice is Monet (version 4.10.2), developed at the CWI in Amsterdam ([7], ([8]). To realize a full-fledged Monet based profile search tool, we should incorporate the `hmmsearch` method into our system to execute the expansion phase. For our project, this approach was to laborious, although all the required information was in the database. To get an impression whether our filtering approach would be fruitful, we simply wrote the filtered strings into a file in Fasta format and let the HMMER `hmmsearch` tool run on this selection in stead of the full database. As long as the selectivity was reasonable, the overhead of generating the output after the filtering step could be ignored.

There were two reasons to choose Monet.

- *main-memory approach*
  Monet is a main-memory DBMS (MMDBMS). The data and, possibly, indexes are supposed to be resident in main-memory. Where the classical DBMS focuses on minimizing IO, Monet gains performance by optimizing for main-memory access and by applying cache-conscious techniques. The data easily fits in main memory and it results in a very short running time for the initial join.
- *layered levels of access and extensibility*
  Monet provides the developer with a extensible relational algebra. It offers the possibility to write specific algebraic operators in C, using an API to access the binary tables, and add them to the Mil collection of standard algebraic operators. This makes sense in the case of very performance critical operations. For performance reasons, we decided to write a Mil-extension to execute the combined self-join-grouping step.

The memory requirements of our approach can be quantified easily. A protein collection in Fasta format (i.e. protein character strings and annotation mixed) has a total size of B bytes. All the protein character strings together contain N characters, where N is practically equal to the number of q-grams. The number of protein strings is L. For the selection of the Swissprot database we used, the values are approximately $L = 100,000$, $N = 38,000,000$ and $B = 48,000,000$.

The tables `Strings` and `Annots` can be mapped easily to Monet, requiring $8L + B$ bytes. The `Qgrams` table is split into three columns according to Monet's binary data model. The column `id` can be represented in a minimal sense with a virtual identifier. For the column `qg`, we used two-byte integers. The total space requirement for the `Qgrams` table now is $14N$ bytes, which is about 0,5 GB for the 48 MB Swissprot selection, easily fitting in main memory.

4.1 HMMER

The HMMER package ([9]) by Sean Eddy provides us with several tools to build and use HMM's. Our main tool of interest is `hmmsearch`, that matches a profile HMM to a string database. `Hmmsearch` was run with an expectation value

parameter $E < 0.1$. This value expresses the probability that a match is found purely by chance.

The package contains a tutorial with two prepared HMM's, that we used gratefully. The first one is the 'RNA recognition motif (rrm)' HMM. It has a size of 77 matching states. The second one is the 'globins50' alignment. It is about twice as long as the 'RNA recognition motif' HMM.

We used version 2.3.2 of HMMER.

5 Experiments and Discussion

Our results of the experiments on the Swissprot selection were compared to the results of running `hmmsearch` from the HMMER package.

The choice of the threshold T for the emission log-odds is a bit of an art. It should be positive to make sense, but values that are too high destroy the sensitivity. We varied around $T = 1$, which turned out to be a good choice.

Selectivity was measured by simply counting the number of bytes of the resulting reduced string set after filtering, compared to the original string collection. It is expressed as a percentage, where a low value indicates a strong selectivity. Because `hmmsearch` behaves quite linear in the size of the string database, the selectivity percentage gives a good indication of the response time behaviour on the filtered string selection.

Sensitivity was measured by checking the presence of high scoring domains. Apart from complete HMM alignments, `hmmsearch` gives a list of local high scoring segments of the database. For each of these local matches, we checked if there was overlap with the candidates we found. Sensitivity is also expressed as a percentage, where a high value is good. We give figures for the complete set of matches found by HMMER (sens100) and for the top k lists, where we only compare the best $k\%$ of the HMMER results. We did this for $k = 60, 40, 20$. The range parameter was fixed on 40 (the Blast default).

The most interesting tuning parameter is n, defining the number of diagonal hits within the range. We mention only the interesting values of n, keeping selectivity close to or less than 10%.

The tests were done on a dual processor Xeon 3.2 GHz machine with 4GB of main memory and 2MB of cache, running under Linux.

In general, we observe that, with adequate tuning, we are able to combine high top k sensitivities with selectivities around or less than 10%. Note that the HMM-hitlist generation requires only a fraction of a second and gives the hitlist size, so the user has the possibility to tune the parameters before running the query. Keep in in mind that output lists will be inspected by biologists manually. Therefore, we claim that referring to the 'upper half' sensitivities is justified, analogous to the practice with web search tools.

The measurements also suggest that the log-odds threshold T is less interesting as a tuning parameter. Fixing $T = 1$ and varying n turns out to be a better tuning principle.

Table 2. Test results

HMM = rrm; runtime hmmsearch = 92 sec T=1 Filtering time: 7.2 sec					
n	sens100	sens60	sens40	sens20	selectivity
7	64%	85.4%	96.7%	99%	1.04%
6	74.8%	93.1%	98.9%	100%	3.9%
5	85.8%	98.9%	100%	100%	16.6%
HMM = globins50; runtime hmmsearch = 166 sec T=1 Filtering time: 3.3 sec					
n	sens100	sens60	sens40	sens20	selectivity
7	75.7%	100%	100%	100%	0.11%
6	81.4%	100%	100%	100%	0.2%
5	82.6%	100%	100%	100%	1.2%
4	87.4%	100%	100%	100%	9.2%
HMM = rrm; runtime hmmsearch = 92 sec T=1.2 Filtering time: 3.4 sec					
n	sens100	sens60	sens40	sens20	selectivity
6	58.4%	78.8%	87.3%	92.3%	0.7%
5	74.8%	91.6%	95.1%	96.7%	3.9%
4	86.0%	97.4%	99.5%	100%	19.7%
HMM = rrm; runtime hmmsearch = 92 sec T=0.8 Filtering time: 10.2 sec					
n	sens100	sens60	sens40	sens20	selectivity
8	64.3%	89.1%	96.2%	98.9%	0.7%
7	73.5%	93.8%	97.8%	100%	2.6%
6	84.0%	97.8%	100%	100%	9.2%

6 Conclusions

Our first goal was to investigate whether main-memory database technology can be succesfully applied to biological sequence alignment. The paper shows that the q-gram indexing techniques of Blast, designed for single sequence matching, could be extended to the HMM matching problem with limited effort, due to the support of the DBMS query facilities. The filtering and its tuning possibilities are fully realized with the possibilities offered by Monet, in a rather concise way. In particular, we extended the algebra with a new operator to calculate the candidates efficiently. We were able to reach filtering times that were significantly smaller than the running times of `hmmsearch`, resulting in filtered string sets with the desired sensitivity and selectivity figures.

A secondary goal was to investigate whether Blast-like q-gram indexing techniques could be applied to profile HMM-matching. Especially the behaviour on the 'top k' result lists is very satisfying when we restrict ourselves to the upper

40 or 60 percent. With appropriate tuning of the querying parameters, we can combine top k sensitivity figures close to 100% with selectivities of less than 10%.

Acknowledgements

The author thanks the Monet development team at the CWI, Amsterdam, for hospitality and support during the development of the prototype.

References

1. Korf, I., Yandell, M., Bedell, J.: Blast, O'Reilly (2003)
2. Durbin, R., Eddy, S.R., Krogh, A., Mitchison, G.: Biological Sequence Analysis. Cambridge University Press, Cambridge (1998)
3. Altschul, S.F., Gish, W., Miller, W., Myers, E.W., Lipman, D.J.: Basic local alignment search tool. Journal of Molecular Biology 215, 403–410 (1990)
4. Altschul, S.F., Madden, T.L., et al.: Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. Nucleic Acids Research 25(17), 3389–3402 (1997)
5. Aluru, S. (ed.): Handbook of Computational Molecular Biology, Chapman & Hall/CRC (2005)
6. Krogh, A.: An introduction to Hidden Markov Models for biological sequences. In: Salzberg, S.L., Searls, D.B., Kasif, S. (eds.) Computational Methods in Molecular Biology, pp. 45–63. Elsevier, Amsterdam (1998)
7. Boncz, P.A., Kersten, M.L.: MIL Primitives for Querying a Fragmented World. The VLDB Journal 8, 101–119 (1999)
8. Boncz, P.A.: Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications, PhD thesis, UVA, Amsterdam, The Netherlands (May 2002)
9. `http://hmmer.janelia.org`
10. Garcia Molina, H., Ullman, J.D., Widom, J.D.: Database System Implementation. Prentice-Hall, Englewood Cliffs (2000)
11. Williams, H.E., Zobel, J.: Indexing and Retrieval for Genomic Databases. IEEE Transactions on Knowledge and Data Engineering 14, 63–78 (2002)