JFP **16** (1): 75–81, 2006. © 2005 Cambridge University Press
doi:10.1017/S0956796805005708 Printed in the United Kingdom

Calculating PSSM probabilities with lazy dynamic programming

KETIL MALDE
Department of Informatics, University of Bergen, Bergen, Norway
(*e-mail:* ketil@ii.uib.no)

ROBERT GIEGERICH
Faculty of Technology, University of Bielefeld, Bielefeld, Germany
(*e-mail:* robert@techfak.uni-bielefeld.de)

Abstract

Position-specific scoring matrices are one way to represent approximate string patterns, which are commonly encountered in the field of bioinformatics. An important problem that arises with their application is calculating the statistical significance of matches. We review the currently most efficient algorithm for this task, and show how it can be implemented in Haskell, taking advantage of the built-in non-strictness of the language. The resulting program turns out to be an instance of dynamic programming, using lists rather the typical dynamic programming matrix.

1 Introduction

Searching strings for matches against a pattern is one of the most fundamental tasks in computer science, and consequently encompasses a wide collection of techniques. One way approximate patterns can be expressed is as position specific scoring matrices, or PSSMs. PSSMs describe fixed length patterns over a finite alphabet. Each row in the matrix corresponds to a position in the pattern, while each column corresponds to one character in the underlying alphabet. Each position in the matrix contains the score (usually an integer or floating-point value) for the character corresponding to its column in the position corresponding to its row. Thus, if M is a PSSM with m rows, and $w = w_1 \ldots w_m$ a string over $\mathscr{A}$, we have $\text{score}_M(w) = \sum_{i=1}^{m} M(i, w_i)$, where a high score value suggests a good match to the pattern described by M.

PSSMs are commonly encountered in bioinformatics, where they represent short conserved regions (often called *motifs*) in proteins or DNA (Durbin *et al*., 1998). Although their descriptive power is moderate compared to hidden Markov models or stochastic context free grammars, which also are frequently used in biosequence analysis, PSSMs are the preferred method for searching for transcription factor binding sites on a genomic scale. Their major virtue is simplicity and speed of search, which can easily be done in $O(mn)$ time, where n is the size of the genome or database. In fact, given that the string to be searched has been preprocessed into an

A	C	G	T
3	2	1	4
6	2	1	1
1	1	7	1
0	0	10	0
0	0	0	10
6	2	1	1
7	1	1	1
2	1	5	2
1	2	1	6

Fig. 1. A sample PSSM; each row contain scores corresponding to the characters A, C, G, and T respectively (representing nucleotides). The matrix is taken from http://genome.imim.es/courses/BioinformaticaUPF/T13/MakeProfile.html, and represents an exon-intron boundary (a *donor site*) in a gene.

index structure, search algorithms have been described that achieve sublinear time (Beckstette *et al*., 2004).

Searching a large text or genome data base with a set of PSSMs requires strict score thresholds to avoid a large number of meaningless hits. This leads us to the problem addressed here: How to evaluate the significance of a match, and how to choose the right score threshold before searching?

The scores themselves are inadequate for this purpose – scores from different PSSMs are unrelated, and longer PSSMs tend to yield higher scores. Instead of using scores directly, it is preferable to examine their p-values (the likelihood of exceeding the score on a random string) or E-values (the expected number of matches exceeding the score in a data set). Ideally, the user of the search routine specifies a p-value threshold P, and we need to translate this, for each PSSM M used in the search, into a score cut-off S_M such that prob(score(w) $\geqslant S_M$) $\leqslant P$. To this end, we need to compute the distribution of scores for M for random strings (drawn from a background distribution π of characters, which is derived from the database).

The probability of obtaining a string of length m scoring exactly s is

$$\sum_{w \in \mathscr{A}^m} \{\text{prob}_\pi(w) | \text{score}(w) = s\}$$

where the probability of a string is $\text{prob}_\pi(w) = \prod_{i=1}^{m} \pi(w_i)$. While the complete probability distribution can be calculated naively, this is infeasible as the number of strings to examine grows as $O(|\mathscr{A}|^m)$. Methods to compute the complete distribution efficiently will be discussed next, on the way to our ultimate goal, which is to use laziness to only compute the relevant parts of the distribution.

2 Calculating the score distribution by dynamic programming

There are well known methods to address the problem of probability-to-score threshold conversion. The most common method is to estimate probablilites from the scores of a random sample of strings. Another, non-heuristic solution employs dynamic programming, which makes it possible to calculate the complete distribution in $O(m^2|R||\mathscr{A}|)$ (Staden, 1989; Wu *et al.*, 2000; Rahmann, 2003; Rahmann *et al.*, 2003), where R is the maximum range of score values in a single row in the matrix. To obtain an integer range of scores, original scores in M are scaled, rounded, and divided by their greatest common divisor.

The dynamic programming technique is based on maintaining the $m \times mR$ table T indexed by PSSM row and score, so that entry $T(i, j)$ contains the accumulated probability of achieving a score of j in the first i rows of the matrix. Row i in the DP table can be calculated by combining the accumulated values in row $(i-1)$ in with contributions from row i in the PSSM. This is expressed in the recurrences

$$T(1, j) = \sum \{\pi(a) | a \in \mathscr{A}, M(1, a) = j\}, j \in mR \tag{1}$$

$$T(i+1, j) = \sum \{\pi(a) T(i, r) | a \in \mathscr{A}, M(i+1, a) + r = j\}, j \in mR \tag{2}$$

Note that an entry $T(i, j)$ is zero when score j cannot be achieved at all with a pattern prefix of length i. The threshold conversion is then achieved by

$$\text{threshold}(P) = \max\{j | j \in mR, \sum_{k=j}^{\max(mR)} T(m, j) \leqslant P\} \tag{3}$$

Dynamic programming algorithms can be expressed quite elegantly in a functional language when the algebraic style proposed in Giegerich *et al.* (2004) is applicable. However, that method works for dynamic programming algorithms where the problem decomposition implicit in the recurrences is given by subwords of the input. Our case here is different: One dimension of the recursion is over the input, considering pattern prefixes of length $i = 1, 2, 3 \ldots$. But the other dimension is over the resulting score values. In this case, for implementing the above recurrences in Haskell, we see no better way than literally rewriting them, using the Haskell `Array` data type for T. This bears no insight and is not shown here.

The most recent contribution to the threshold conversion problem is based on the observation that we need the distribution only to translate threshold p-values into threshold scores. Given that the threshold p-values are typically very small (to avoid hits that arise by chance), we really need only the upper end of the distribution. This has been implemented in Beckstette *et al.* (2004). The authors re-order rows and columns of M to compute first the probability of the maximum score (which requires the probabilities of partial scores that contribute to it), then the same for the second best possible score, and so on. Computation stops when the accumulated probabilities exceed the specified p-value. The score value where this happens is the desired score cutoff. Tabulation is still essential to ensure that no partial scores are computed twice.

This leads to a very fast program for sufficiently small p-values, which degenerates to the standard DP implementation (plus some overhead) when we reach the low end of the distribution. The authors report speed-up factors of 10 and 16 for p-value thresholds 10^{-10} and 10^{-40}, respectively. The program, implemented in C, is much more sophisticated than the standard DP approach – and the care taken to compute

```
type PVector = [Prob]
type PSSM = [[Score]]

-- combine scores from each matrix row with respective probabilities
prep :: PVector -> PSSM -> [[(Score,Prob)]]
prep pv = map (\r -> reverse $ sort $ zip r pv)

-- compute path probabilities in order of descending score
paths :: PVector -> PSSM -> [(Score,Prob)]
paths pv = foldr1 add_paths . prep pv
    where add_paths ps rs = foldr1 merge [[combine x y | y <- rs] | x <- ps]
          combine (s1,p1) (s2,p2) = (s1+s2,p1*p2)

merge = merge'

merge' :: [(Score,Prob)] -> [(Score,Prob)] -> [(Score,Prob)]
merge' [] ys = ys
merge' xs [] = xs
merge' (x:xs) (y:ys) = if fst x > fst y then x : merge' xs (y:ys)
                       else y : merge' (x:xs) ys

-- compute score cutoff from probability cutoff (Eqn. 3)
prob2score :: Prob -> [(Score,Prob)] -> Score
prob2score p ((s,r):ps) = if null ps || p <= r then s
                          else prob2score (p-r) ps
```

Fig. 2. Calculating paths through the PSSM by score in descending order. Each path is represented as a pair of a score and its associated probability. A `PVector` gives the background distribution as a probability for each character in the alphabet.

only those intermediate values in the DP table that contribute to the high end of the distribution brings about a flavour of laziness.

Here we study how the above idea can be implemented in a functional language where laziness comes for free. It should not come as a surprise that the implementation is much simpler. The interesting aspect of our derivation of the program is that it does not start from the DP recurrences. Instead, it starts from the (impractical) naive approach, and the equivalent of dynamic programming sneaks in almost unnoticed.

3 Algorithm and implementation

We start from the naive idea of considering all words in $\mathscr{A}^m$, computing their probabilities and scores. This gives the complete distribution, and the threshold conversion can be achieved via Equation 3. Since we are interested only in the high end of the distribution, let us consider the words in order of decreasing score.

Figure 2 shows how the scores and probabilities corresponding to possible paths through the PSSM can be generated. The `prep` function pairs scores from the PSSM with corresponding character probabilities, while the character giving rise to a pair needs not be recorded. The `paths` function iterates over the rows of the PSSM,

```
merge xs = join . merge' xs

join ((s,p):(s',p'):xs) = if s /= s' then (s,p) : join ((s',p'):xs)
                          else join ((s,p+p'):xs)
join xs = xs
```

Fig. 3. Improving efficiency by combining list elements that have the same score when merging the lists.

combining the scores and probabilities from each new row with the paths calculated from the previous rows. A lazy merge sort on the path scores ensures that the paths relevant to the high end of the score distribution are calculated first. Finally, the function `prob2score` implements Equation 3.

We note that any entries in a matrix row having the same score will contribute to the calculation simultaneously, and they can thus be combined into one entry. For example, in Figure 1 the entries for letters G and T in row 2 both contribute a score of 1, hence they may be seen as a joined character G + T that contributes a score of 1 with probability $\pi(G) + \pi(T)$. This consideration also extends to any intermediate result, whenever two initial paths happen to achieve the same score. We can thus eliminate some calculations by combining entries having the same score. As the scores are produced in descending order, it is only necessary to consider adjacent values. We define a function `join` to perform this, and update the `merge` function accordingly. The resulting code is shown in Figure 3.

It is interesting to note that this is exactly what happens in the dynamic programming approach. The DP matrix serves to merge any match prefixes that score the same, and only their combined probability is retained.

4 Results

PRINTS (Attwood *et al.*, 1999) is a collection of PSSMs for finding certain patterns (motifs) in proteins. We measured the efficiency of our implementation[1] running it on 1000 matrices from PRINTs, and comparing it to the implementation of the straightforward dynamic programming algorithm, using a Haskell unboxed array for the DP matrix. The results are shown in Table 1.

We see that for these p-values, there is a significant advantage to using the lazy method, clearly outweighing the advantage unboxed arrays normally provide over lists. Naturally, the advantage disappears as a larger portion of the distribution is calculated for cut-off values above 10^{-10}.

Another advantage of the lazy algorithm is that it is less dependent on the range of the scores. If the scores are, for instance, arbitrary floating point values, the DP matrix will be large. While our lazy algorithm will derive less benefit from the redundancy elimination, it will still be able to take full advantage of laziness. (In practice, scores will usually be rounded to reduce the range of different values).

[1] The programs were compiled with `ghc` version 6.2 using `-O2`, and run on a 1130 MHz Pentium III.

Table 1. *Average time per matrix for calculating the thresholds for PRINTS matrices, given different p-values. Total times and average time per matrix are given*

	DP	s/M	prob2score	s/M	speedup
10^{-30}	1313s	1.31	7.6s	0.01	172x
10^{-20}	1388s	1.39	56s	0.06	25x
10^{-10}	1317s	1.32	306s	0.31	4.3x

5 Concluding remarks

In contrast to Beckstette *et al.* (2004), we started out with lazy generation of paths through the matrix, and added dynamic programming later. Perhaps unsurprisingly, we ended up with a very similar algorithm, and reasonably comparable run times when taking into account the different run-time systems.

While a well-written C program is generally accepted to perform better than the corresponding Haskell program, it is interesting to note that Beckstette *et al.* (2004) reports an average of 0.45 seconds per matrix for the full dynamic programming algorithm on the complete PRINTS set for their PoSSuM implementation, and 0.04–0.06 seconds for lazy calculations with the same thresholds as used in Table 1, using a 900 MHz CPU. While there is a difference in the performance of the hardware platforms, it appears that GHC's unboxed arrays perform well enough in practice to be competitive with C implementations.

We also note that our lazy implementation seems to scale poorly as the number of traversed paths increase, from being a factor of four times slower at 10^{-10} to outperforming PoSSuM at 10^{-30}. However, the bound of 10^{-30} is too aggressive for many of the matrices, and the highest-scoring path alone is often sufficient to exceed the threshold (and thus only a single path will be calculated). We remark anecdotically that, challenged by the competitive performance of the Haskell program, the authors of PoSSuM decided to hand-code their memory allocation, and thereby achieved a marginal speedup for 10^{-10}, but a 34-fold speedup over the published performance for 10^{-30}. Hence, the same scaling behaviour is observed for the hand-coded laziness in C.

The interesting aspect is that the built-in non-strictness of Haskell allows us to express this algorithm in a natural manner, and in fact the complete implementation for calculating the probability distribution fits comfortably on a page, and is comparable in size to the array-based implementation of the dynamic programming algorithm.

Acknowledgments

We wish to thank Michael Beckstette and Dirk Strothman for helpful advice and comments, and Richard Bird for identifying a memory inefficiency issue in an earlier version.

References

Attwood, T. K., Flower, D. R., Lewis, A. P., Mabey, J. E., Morgan, S. R., Scordis, P., Selley, J. N. and Wright, W. (1999) PRINTS prepares for the new millennium. *Nucleic Acids Res.* **27**(1), 220–225.

Beckstette, M., Strothman, D., Homann, R., Giegerich, R. and Kurtz, S. (2004) PoSSuMsearch: Fast and sensitive matching of position specific scoring matrices using enhanced suffix arrays. *Proceedings of the German Conference on Bioinformatics 2004*, pp. 53–64.

Durbin, R., Eddy, S., Krogh, A. and Mitchison, G. (1998) *Biological Sequence Analysis*. Cambridge University Press.

Giegerich, R., Meyer, C. and Steffen, P. (2004) A discipline of dynamic programming over sequence data. *Sci. Comput. Program.* **51**(3), 215–263.

Rahmann, S. (2003) Dynamic programming algorithms for two statistical problems in computational biology. *Proceedings of the 3rd Workshop of Algorithms in Bioinformatics (WABI)*, pp. 151–164.

Rahmann, S., Müller, T. and Vingron, M. (2003) On the power of profiles for transcription factor binding site detection. *Stat. Applications in Genetics & Molecular Biol.* **2**(1).

Staden, R. (1989) Methods for calculating the probabilities of finding patterns in sequences. *Comput. Applic. in the Biosci.* **5**, 89–96.

Wu, T. D., Nevill-Manning, C. G. and Brutlag, D. L. (2000) Fast probabilistic analysis of sequence function using scoring matrices. *Bioinformatics*, **16**(3), 233–244.