Asap: A framework for over-representation statistics for transcription factor binding sites, supplementary material

1 High-order models

Having both a motif model and a background model we can define the PWM as the log-ratio between the motif frequency and the corresponding background frequency across all nucleotides and all positions. Assume that the motif model is an order m Markov model and the background model is a Markov chain of order n. The probability of observing a sequence $x_1, \ldots, x_L$ according to the background model can then be written as:

$$P_{bg}(x_1, \ldots, x_L) = \hat{q}(x_1, \ldots, x_n) q(x_{n+1}|x_1, \ldots, x_n)$$

$$\ldots q(x_i|x_{i-n}, \ldots, x_{i-1}) \ldots q(x_L|x_{L-n}, \ldots, x_{L-1})$$

where $\hat{q}(x_1, \ldots, x_n)$ is the equilibrium probabilities of the Markov chain. We thus assume that the initial sampling distribution from which we start generating the sequence is the equilibrium probabilities. The above is the likelihood of the background model. The probability of observing a sequence with the motif at position i under the motif model and the background model can then be written as:

$$\hat{q}(x_1, \ldots, x_n) q(x_{n+1}|x_1, \ldots, x_n) \ldots q(x_{i-1}|x_{i-n-1}, \ldots, x_{i-2})$$

$$\times p_1(x_i|x_{i-m}, \ldots, x_{i-1}) \ldots p_W(x_{i+W-1}|x_{i+W-m-1}, \ldots, x_{i+W-2})$$

$$\times q(x_{i+W}|x_{i+W-n}, \ldots, x_{i+W-1}) \ldots q(x_L|x_{L-1}, \ldots, x_{L-1})$$

where p_1 to p_W are the position specific nucleotide probabilities. In the general formulation we define the scores as the log of the ratio of the conditional probabilities. The log-likelihood-ratio between the two models is then:

$$\log \frac{p_1(x_i|x_{i-m}, \ldots, x_{i-1}) \ldots p_W(x_{i+W-1}|x_{i+W-m-1}, \ldots, x_{i+W-2})}{q(x_i|x_{i-n}, \ldots, x_{i-1}) \ldots q(x_{i+W-1}|x_{i+W-n-1}, \ldots, x_{i+W-2})} =$$

$$\sum_{k=0}^{W-1} \log \frac{p_{k+1}(x_{i+k}|x_{i-m+k}, \ldots, x_{i+k-1})}{q(x_{i+k}|x_{i-n+k}, \ldots, x_{i+k-1})}$$

Then the general form of a higher-order PWM is essentially the same as the standard PWM: scores are log ratios of the conditional pattern probabilities and conditional background probabilities. Here we have ignored any PWM scoring above threshold within the first $n + 1$ nucleotides and assume that that $n \geq m$.

2 Speed test

2.1 The datasets used

For the performance tests we had to choose some sequence sets and some PWMs to run the algorithms on.

2.1.1 The sequences sets

When choosing the sequence set we found it necessary to take into account that the size of the input DNA and the extend to which the suffixes in the sequences have common prefixes can have an effect on performance. The problem was to create a series of tests that would allow us to asses the isolated effects of sequence size and of common prefix lengths.

In order to asses the influence of size independently, we made a master file containing all transcriptional start sites in the human genome found in the DBTSS database; around 31.000 sequences all of length 1201 bases. We then partitioned these sequences into several sets of files: The first test set simply consisted of the master file, the second test set consisted of two files each containing one half of the master file. The third and the fourth test set consisted of the master file split into 9 and 36 parts, respectively.

The idea was that by taking the average of the running times of all files in a set we would (to a certain extend) able to assess how changes in sequence size alone affect the running time.

In order to assess the influence of the lengths of the common prefixes we created four new test sets consisting of 2, 4, 36 and 72 files, respectively, in the exact same way as before. Each of the files in these sets were doubled, though, in the sense that we added an extra copy of all the sequences in it. In this way we again got four test set with files of the exact same sizes as before (36MB, 18MB, 4MB and 1MB). But these files had an artificially increased extend of prefix sharing. Hence, by comparing the running times of each algorithm on the original partitions and the doubled ones we hoped

File size	File type	Number of files
$\simeq$ 36 MB	single	1
$\simeq$ 36 MB	double	2
$\simeq$ 18 MB	single	2
$\simeq$ 18 MB	double	4
$\simeq$ 4 MB	single	9
$\simeq$ 4 MB	double	18
$\simeq$ 1 MB	single	36
$\simeq$ 1 MB	double	72

Table 1: An overview of our set of test files. The file type double means that we added an extra copy of all the sequences from the single file type.

to get an idea of how the different algorithms reacted to increases in prefix sharing, independently of the file size. It should be noted that we are aware that our method produces an unrealistically high extend of prefix sharing. We still think the test has relevance, though. In table 1 there is an overview of the set of test files we used.

2.1.2 The PWM set

The performance of a search with a PWM is highly dependent on the size and composition of the sequence set that is searched. It is also dependent on the contents of the sequence, the length of the PWM and on the threshold chosen. We therefore chose to look at the average performance of fifty randomly picked PWMs from the TRANSFAC database. For each of them, we chose a threshold that gave us around 1 hit per 10,000 bases in the our master file. In this way we sought to approximate typical queries.

2.1.3 Performance measures

The performance tests were run on a machine equipped with a 2.4 GHz Intel Pentium 4 processor with 1.5 GB of memory running Linux. We used the sum of the `user` and `sys` times as reported by the Linux `time` as the running time measure as opposed to using the real time running time.

2.1.4 Testing different suffix sorting algorithms

We first tested a number of other suffix sorting algorithms, namely `copy`, `cache`, `qsufsort`, a homemade radix sort and the original `ds`. All of them

File size	File type	copy	cache	qsufsort	cr	ds	our ds
$\simeq$ 36 MB	single	33.94	46.40	45.13	323.38	22.53	22.55
$\simeq$ 36 MB	double	T/O	T/O	259.46	311.40	418.00	22.72
$\simeq$ 18 MB	single	15.27	20.97	20.80	158.70	10.41	10.18
$\simeq$ 18 MB	double	T/O	477.83	115.10	151.12	114.74	10.08
$\simeq$ 4 MB	single	2.47	3.55	4.00	30.57	1.76	1.72
$\simeq$ 4 MB	double	T/O	29.66	21.03	28.86	8.78	1.72
$\simeq$ 1 MB	single	0.38	0.53	0.75	4.41	0.26	0.26
$\simeq$ 1 MB	double	67.39	0.98	4.13	4.35	0.87	0.27

Table 2: The results from time testing the 6 different sorting algorithms. They are average running times measured in seconds. T/O stands for timeout and means that the sorting took more than ten minutes on one or more file of the given size and type.

besides the homemade radix sort are C implementations made by Manzini and are available from his homepage:

`http://www.mfn.unipmn.it/~manzini/lightweight/index.html`

It should be noted that when compiling `copy`, `cache` and `qsufsort` there was a single function that all three algorithms used, which `gcc` warned that it was not able to inline. Hence the running times of these might have suffered a bit. The results from running these and our own sorting algorithm can be seen in table 2.

What can be seen is that `ds` and our modified version of `ds` are the fastest algorithms on all the single files and that our algorithm is the fastest on all the doubled files. This suggests that the algorithm we use is indeed competitive to other algorithms available. Especially when the extend of prefix sharing is very high.

2.2 Testing of the lcp algorithms

Next we tested the naive lcp-algorithm and the lcp-algorithm by Kasai. It should be noted that we implemented the naive algorithm ourselves, but that we used an implementation the Kasai-algorithm made by Manzini:

`http://www.mfn.unipmn.it/~manzini/lightweight/index.html`

The results of the test can be seen in table 3.

As can be seen the naive lcp algorithm is actually at least 3 times as fast on the single files and at least twice at fast on the double files as the Kasai-algorithm. Hence on our datasets the naive lcp is preferable.

File size	File type	Naiv Lcp	Kasai
$\simeq$ 36 MB	single	8.65	28.20
$\simeq$ 36 MB	double	10.06	24.20
$\simeq$ 18 MB	single	4.17	12.45
$\simeq$ 18 MB	double	4.79	10.81
$\simeq$ 4 MB	single	0.79	2.58
$\simeq$ 4 MB	double	0.95	2.19
$\simeq$ 1 MB	single	0.12	0.57
$\simeq$ 1 MB	double	0.15	0.51

Table 3: The results from time testing the two different lcp algorithms. The results are measured in seconds and they are average times.

File size	File type	Our `ESAsearch`	Naive w. lookahead	Searches
$\simeq$ 36 MB	single	0.20	2.44	15
$\simeq$ 36 MB	double	0.13	2.44	16
$\simeq$ 18 MB	single	0.13	1.22	14
$\simeq$ 18 MB	double	0.08	1.22	15
$\simeq$ 4 MB	single	0.04	0.27	12
$\simeq$ 4 MB	double	0.03	0.27	14
$\simeq$ 1 MB	single	0.01	0.07	8
$\simeq$ 1 MB	double	0.01	0.07	9

Table 4: The results from time testing the two different search algorithms. The search times are specified as average times measured in seconds.

2.3 The number of searches required

Finally we assessed how many searches that are required for `ESAsearch` to be preferable over the naive algorithm with lookahead when taking the ESA building time into account. We simply ran our two search algorithms on all the test files in our test file set using the entire PWM set. The results can be seen in table 4. In the last column in the table the number of searches for reaching "break-even" is indicated.

What can be seen is that for the bigger files in our data set, around 15 searches are required for the enhanced suffix array to be preferable. For the smaller files the number is slightly lower. In general about one more search was required for the double files than for the single files.