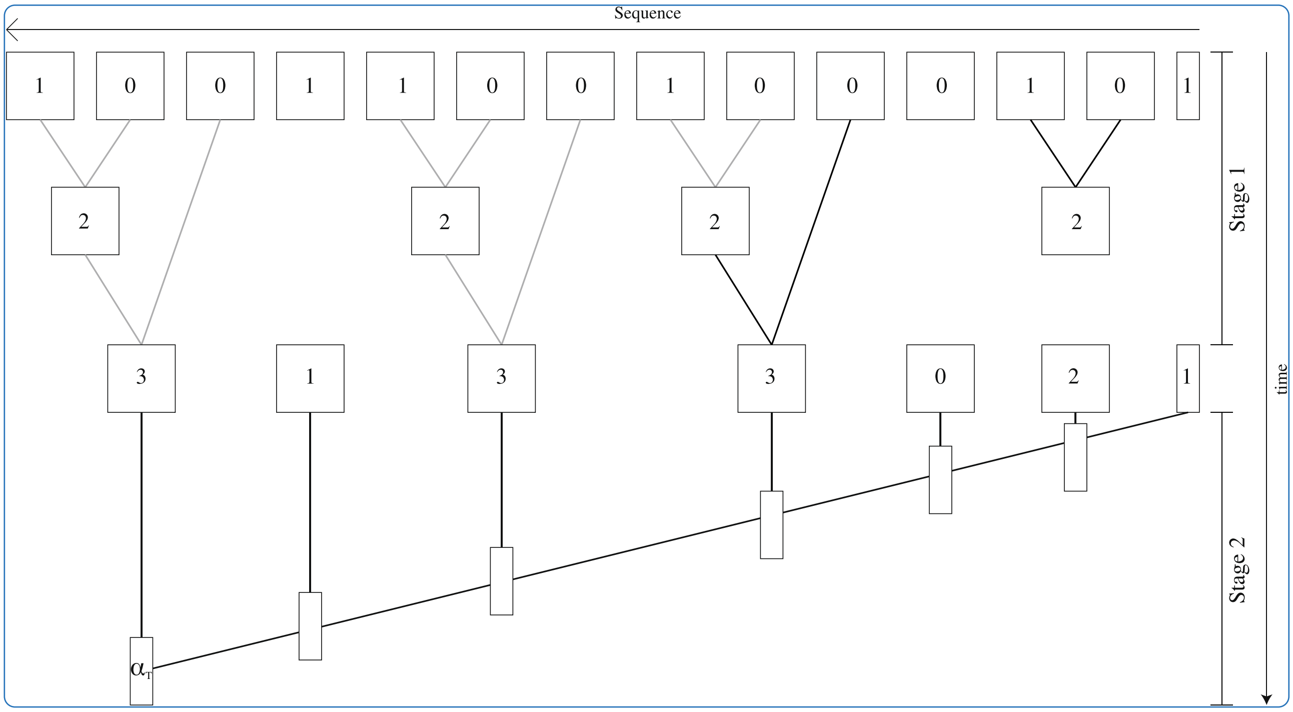
BMC Bioinformatics



zipHMMlib: a highly optimised HMM library exploiting repetitions in the input to speed up the forward algorithm

Sand *et al.*

BMC Bioinformatics

SOFTWARE Open Access

zipHMMlib: a highly optimised HMM library exploiting repetitions in the input to speed up the forward algorithm

Andreas Sand[1,2*], Martin Kristiansen[2], Christian NS Pedersen[1,2] and Thomas Mailund[1]

Abstract

Background: Hidden Markov models are widely used for genome analysis as they combine ease of modelling with efficient analysis algorithms. Calculating the likelihood of a model using the forward algorithm has worst case time complexity linear in the length of the sequence and quadratic in the number of states in the model. For genome analysis, however, the length runs to millions or billions of observations, and when maximising the likelihood hundreds of evaluations are often needed. A time efficient forward algorithm is therefore a key ingredient in an efficient hidden Markov model library.

Results: We have built a software library for efficiently computing the likelihood of a hidden Markov model. The library exploits commonly occurring substrings in the input to reuse computations in the forward algorithm. In a pre-processing step our library identifies common substrings and builds a structure over the computations in the forward algorithm which can be reused. This analysis can be saved between uses of the library and is independent of concrete hidden Markov models so one preprocessing can be used to run a number of different models.
Using this library, we achieve up to 78 times shorter wall-clock time for realistic whole-genome analyses with a real and reasonably complex hidden Markov model. In one particular case the analysis was performed in less than 8 minutes compared to 9.6 hours for the previously fastest library.

Conclusions: We have implemented the preprocessing procedure and forward algorithm as a C++ library, zipHMM, with Python bindings for use in scripts. The library is available at http://birc.au.dk/software/ziphmm/.

Background

Hidden Markov models (HMMs) are a class of statistical models for sequential data with an underlying hidden structure. They were first introduced to bioinformatics in the late 1980s [1] and have since then been used in a wide variety of applications, for example for gene finding [2], modelling of protein structures [3,4], sequence alignment [5] and phylogenetic analysis [6-9]. Because of their computational efficiency HMMs are one of the few widely used statistical methodologies that are feasible for genome wide analysis where sequences often are several hundred million characters long. With data sets of this size, however, analysis time is still often measured in hours and days. Improving on the performance of HMM analysis is therefore important to keep up with the quickly growing amount of biological sequence data to be analysed, and to make more complex analyses feasible.

The most time consuming part of using hidden Markov models is often parameter fitting, since the likelihood of a model needs to be computed repeatedly when optimising the parameters. Depending on the optimisation strategy, this means that the forward algorithm (or both the forward and the backward algorithm) will be evaluated in potentially hundreds of points in parameter space. Optimising the forward algorithm is therefore the most effective strategy for efficient HMM implementations.

The forward algorithm can be seen as a sequence of vector-matrix operations along an input sequence. This, however, can be rewritten as a sequence of matrix-matrix operations. This replaces an $\mathcal{O}(n^2)$ time vector-matrix operation with an $\mathcal{O}(n^3)$ time matrix-matrix operation but opens up possibilities for changing the evaluation

*Correspondence: asand@birc.au.dk
[1]Bioinformatics Research Centre, Aarhus University, Aarhus, Denmark
[2]Department of Computer Science, Aarhus University, Aarhus, Denmark

order since different parts of the computation now can be handled independently. This then makes it possible to reuse computations whenever the input contains repeated substrings [10] or to parallelise the algorithm across a number of independent threads [11].

The main contribution of this paper is a software library that uses both of these ideas to greatly speed up the forward algorithm. We present a preprocessing of the observed sequence that finds common substrings and constructs a data structure that makes the evaluation of the likelihood close to two orders of magnitude faster (not including the preprocessing time). The preprocessing of a specific sequence can be saved and later reused in the analysis of a different HMM topology. The algorithms have been implemented in a C++ library, zipHMMlib, available at http://birc.au.dk/software/ziphmm/. The library also provides an interface to the Python programming language.

Much of the theory used in zipHMMlib was also developed by Lifshits et al. [10], but while they developed the theory in the context of the Viterbi algorithm, where the preprocessing cannot be reused, we concentrate on the forward algorithm and introduce a data structure to save the preprocessing for later reuse. We furthermore extend the theory to make the computations numerically stable and introduce practical measures to make the algorithm run fast in practice and make the library accessible.

Our implementation is tested on simulated data and on alignments of chromosomes from humans with chimpanzees, gorillas and orangutans analysed with the CoalHMM framework [7,8,12], a framework which uses changes in coalescence trees along a sequence alignment to make inference in population genetics and phylogenetics and which has been used in a number of whole-genome analyses [13-16]. Using an "isolation-with-migration" CoalHMM [17], we train the model using the Nelder-Mead-simplex algorithm and measure the preprocessing time and total optimisation time. Looking at the time required to perform the entire training procedure, we achieve up to 78 times shorter wall-clock time compared to the previously fastest implementation of the forward algorithm. Even for data of high complexity and with few repetitions we achive a speedup of a factor 4.4.

Implementation

Hidden Markov models

A Hidden Markov Model (HMM) describes a joint probability distribution over an observed sequence $Y_{1:T} = y_1 y_2 \ldots y_T \in \mathcal{O}^*$ and a hidden sequence $X_{1:T} = x_1 x_2 \ldots x_T \in \mathcal{H}^*$, where $\mathcal{O}$ and $\mathcal{H}$ are finite alphabets of observables and hidden states, respectively. The hidden sequence is a realisation of a Markov process which explains hidden properties of the observed data. We can formally define an HMM [18] as consisting of:

- $\mathcal{H} = \{h_1, h_2, \ldots, h_N\}$, a finite alphabet of hidden states;
- $\mathcal{O} = \{o_1, o_2, \ldots, o_M\}$, a finite alphabet of observables;
- a vector $\Pi = (\pi_i)_{1 \le i \le N}$, where $\pi_i = \Pr(x_1 = h_i)$ is the probability of the model starting in hidden state h_i;
- a matrix $A = \{a_{ij}\}_{1 \le i,j \le N}$, where $a_{ij} = \Pr\left(x_t = h_j | x_{t-1} = h_i\right)$ is the probability of a transition from state h_i to state h_j;
- a matrix $B = \{b_{ij}\}_{1 \le i \le N}^{1 \le j \le M}$, where $b_{ij} = \Pr\left(y_t = o_j | x_t = h_i\right)$ is the probability of state h_i emitting o_j.

An HMM is parameterised by π, A and B, which we will denote by $\lambda = (\pi, A, B)$.

The classical forward algorithm

The forward algorithm [18] finds the probability of observing a sequence $Y_{1:T}$ in a model λ by summing the joint probability of the observed and hidden sequences for all possible hidden sequences: $\Pr(Y_{1:T} \mid \lambda) = \sum_{x_{1:T}} \Pr(Y_{1:T}, X_{1:T} = x_{1:T} \mid \lambda)$. This sum is normally computed by first filling out a table, α, with entries $\alpha_t(x_t) = \Pr(Y_{1:t}, X_t = x_t \mid \lambda) = \sum_{x_{1:t-1}} \Pr(Y_{1:t}, X_{1:t} = x_{1:t} \mid \lambda)$ column by column from left to right, using the recursion

$$\begin{aligned} \alpha_1(x_1) &= \pi_{x_1} b_{x_1,y_1} \\ \alpha_t(x_t) &= b_{x_t,y_t} \sum_{x_{t-1}} \alpha_{t-1}(x_{t-1})\, a_{x_{t-1},x_t}. \end{aligned} \tag{1}$$

After filling out α, $\Pr(Y_{1:T} \mid \lambda)$ can be computed as $\Pr(Y_{1:T} \mid \lambda) = \sum_{x_T} \alpha_T(x_T)$.

The algorithm as linear algebra

In the classical forward algorithm, we compute the columns of α from left to right by the recursion in equation (1). If we can compute the last one of these columns, α_T, efficiently, we can compute $\Pr(Y_{1:T} \mid \lambda) = \sum_{x_T} \alpha_T(x_T)$. Now let α_t be the column vector containing the $\alpha_t(x_t)$'s:

$$\alpha_t = \begin{bmatrix} \alpha_t(h_1) \\ \alpha_t(h_2) \\ \ldots \\ \alpha_t(h_N) \end{bmatrix},$$

let B_{o_i} be the diagonal matrix, having the emission probabilities of o_i on the diagonal:

$$B_{o_i} = \begin{bmatrix} b_{1,o_i} & & & \\ & b_{2,o_i} & & \\ & & \ldots & \\ & & & b_{N,o_i} \end{bmatrix},$$

and let

$$C_{o_i} = B_{o_i}A^* \quad \text{and} \quad C_1 = B_{y_1}\pi, \tag{2}$$

where A^* is the transpose of A. Then α_t can be computed using only C_{y_t} and the previous column vector α_{t-1}:

$$\alpha_t = C_{y_t}\alpha_{t-1} = C_{y_t}C_{y_{t-1}}\cdots C_{y_2}C_1. \tag{3}$$

Thus the classical forward algorithm can be described as a series of matrix-vector multiplications of length $T-1$ as illustrated in Figure 1. The classical forward algorithm corresponds to computing this from right to left, but since matrix-matrix multiplication is associative the product can be computed in any order. Since repeated substrings corresponds to repeated matrix-matrix multiplications, running time can be improved by reusing shared expressions [10,11]. In the following sections we will show how we precompute a grouping of the terms based on $Y_{1:T}$ in order to minimise the workload in the actual computation of the likelihood.

Exploiting repetitions in the observed sequence

Let $o_io_j \in \mathcal{O} \times \mathcal{O}$ be the pair of symbols occurring most often without overlap in $Y_{1:T}$, and let $n_{o_io_j}$ be the number of occurrences. We can then reduce the length of $Y_{1:T}$ with $n_{o_io_j}$ characters by introducing a new symbol o_{M+1} and replacing all occurrences of o_io_j by this symbol. To mimic this in the computation described above, we introduce a new C matrix:

$$C_{o_{M+1}} = C_{o_i}C_{o_j}.$$

Now notice that we only need to compute this matrix once and substitute it for all occurrences of $C_{o_i}C_{o_j}$ in equation (3). Hence we can save $n_{o_io_j}$ matrix-vector multiplications by introducing one matrix-matrix multiplication, potentially saving us a large amount of work.

These observations suggest that we can split the computation of the likelihood of a given observed sequence in a preprocessing of the sequence and in the actual computation of the likelihood. In the preprocessing phase we compress the observed sequence by repeatedly finding the most frequent pair of symbols o_io_j in the current sequence and replacing all occurrences of this pair by a new symbol. This is repeated until $n_{o_io_j}$ becomes too small to gain a speedup (see next section). The result is a sequence $Y'_{1:T'}$ over a new alphabet $\mathcal{O}' = \{o_1, o_2, \ldots, o_M, o_{M+1} = (l_1, r_1), o_{M+2} = (l_2, r_2), \ldots, o_{M'} = (l_{M'-M}, r_{M'-M})\}$, where $l_i, r_i \in \{o_1, o_2, \ldots o_{i-1}\}$. This compression will be identical independent of the HMM, meaning that we can save it along with the observed sequence and reuse it for any HMM.

The actual computation of the likelihood is then split in two stages. In the first stage we compute C_1 and C_{o_i} for $i = 1, \ldots, M$ using (2). We then compute C_{o_i} for increasing $i = M+1, \ldots, M'$ by $C_{o_i} = C_{l_i}C_{r_i}$. In the second stage, we compute α_T by

$$\alpha_T = C_{y'_{T'}}C_{y'_{T'-1}}\cdots C_{y'_2}C_1.$$

This is illustrated in Figure 2 where the actual computation is drawn in solid black, while the saved work due to redundancy is shown in gray.

Compression stopping criterion

While the first iterations of the preprocessing procedure compress the sequence very effectively, the last iterations do not decrease the sequence length by much, since most pairs are uncommon when more characters are introduced. This is illustrated in Figure 3, where we see that the number of occurrences of the most frequent pair of symbols decreases superexponentially as a function of the number of iterations performed on an alignment of the human and chimpanzee chromosome 1. This means that we potentially save a lot of time on the likelihood computation by performing the first iterations, but as the slope of the curve increases towards 0 we risk to spend a long time on the preprocessing and save very little time on the actual likelihood computation. To overcome this problem, we do not compress the input sequence all the way down to a single character. Assume we know that the preprocessing will not be reused for an HMM with less than N_{min} states, and let t_{mv} be the time required for an $(N_{min} \times N_{min}) \times N_{min}$ matrix-vector multiplication and t_{mm} be the time required for an $(N_{min} \times N_{min}) \times (N_{min} \times N_{min})$ matrix-matrix multiplication. In iteration i of the preprocessing we replace the most frequent pair of two symbols in the current sequence

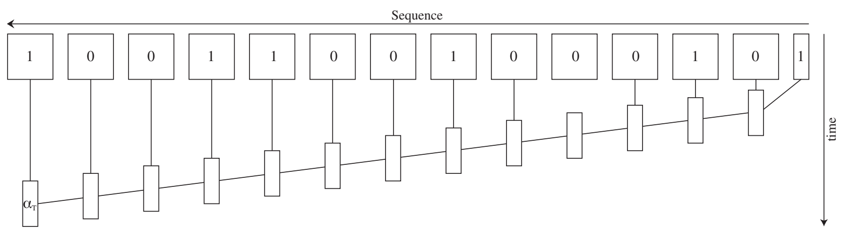


Figure 1 Classical approach to the forward algorithm. The classical forward algorithm, as described by Rabiner [18]. The rectangles represent matrices and vectors. The black lines denote matrix-vector multiplications. The top row is the C_{o_i} matrices. $\boldsymbol{\alpha}_i$ is obtained from $\boldsymbol{\alpha}_{i-1}$ and C_{o_i}. Note that the input sequence is inverted to illustrate that the series of matrix-vector multiplication should be carried out from right to left.

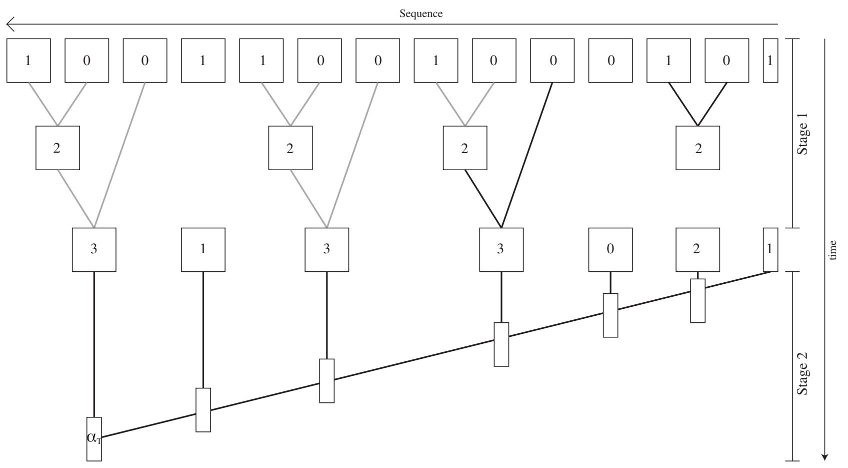


Figure 2 Reusing common expressions to speed up the forward algorithm. The actual computation of the likelihood of the observed sequence 10100010011001 given a specific HMM. The rectangles represent matrices and vectors and lines between them represent dependencies. In stage 1 the C_{o_i} matrices are computed. The solid black lines show the amount of work performed, while the grey lines show the amount of work saved due to redundancy. C_2 is for example computed as the product C_1C_0, and this multiplication is saved three times. In the second stage $\boldsymbol{\alpha}_T$ is computed from the compressed sequence and the C_{o_i} matrices. As in Figure 1 $\boldsymbol{\alpha}_i$ is computed from $\boldsymbol{\alpha}_{i-1}$ and C_{o_i}.

and find the most frequent pair of two symbols in the resulting sequence. Thus if p_i is the number of occurrences of the pair found in iteration i, pre_i is the time required for iteration i, and e is an estimate (given by the user) of the number of times the preprocessing is going to be reused (for example in a number of training procedures each calling forward several times), then, assuming that the matrix-vector multiplications and matrix-matrix multiplications dominate the runtime of the actual likelihood computations, the amount of time that is saved by running iteration i is $e(t_{mv}p_{i-1} - t_{mm}) - pre_i$, as we save p_{i-1} matrix-vector multiplications in each likelihood computation, and we do this by introducing one new matrix-matrix multiplication. This means that the optimal time to stop the preprocessing is before iteration j, where j is the minimal value of i making $e(t_{mv}p_{i-1} - t_{mm}) - pre_i$ less than or equal to 0. However, we do not know pre_i before iteration i has been completed, but we can estimate it by pre_{i-1}. Thus we stop the preprocessing just before iteration j, where j is the minimal value of i making $e(t_{mv}p_{i-1} - t_{mm}) - pre_{i-1}$ less than or equal to 0.

The values t_{mv} and t_{mm} are measured prior to the preprocessing, whereas the user has to supply an estimate, e, of the number of reuses of the preprocessing and N_{min}. If a single value of N_{min} can not be determined, we allow the user to specify a list of state space sizes $(N^1_{\text{min}}, N^2_{\text{min}}, ...)$ for which he wants the preprocessing to be saved. If no N_{min} values are provided, the compression is stopped whenever $p_i = p_{i-1}$ for the first time.

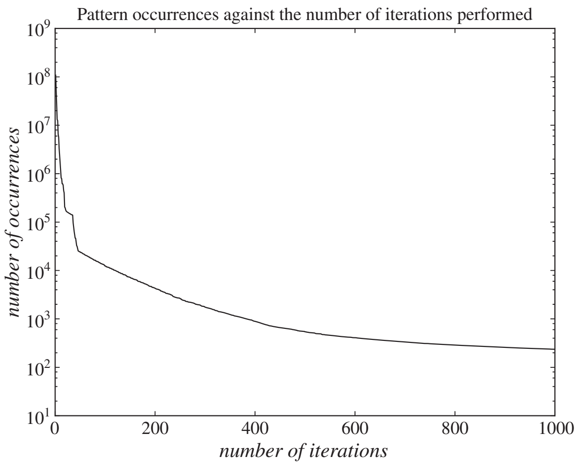


Figure 3 Number of pattern occurrences against the number of iterations performed. The number of occurrences of the most frequent pair of symbols plotted on a log-scale against the number of iterations performed of the preprocessing procedure on an alignment of the human and chimpanzee chromosome 1 encoded using the alphabet {0, 1, 2} for identical sites, differing sites or missing data, respectively.

Numerical stability

All our matrices contain probabilities, so all entries are between 0 and 1. This means that their products will tend towards 0 exponentially fast. The values of these products will normally be stored in one of the IEEE 754 floating-point formats. These formats have limited precision, and if the above was implemented naïvely the results would quickly underflow.

If we can make do with $\log(\Pr(Y_{1:T} \mid \lambda))$, we can prevent this underflow by repeatedly rescaling the matrices, much in the same way as the columns are rescaled in the numerically stable version of the classical forward algorithm [18]. To make this work in our case, we will normalise the results of each matrix-matrix multiplication or matrix-vector multiplication we do throughout the algorithm and work with the normalised matrices instead. We first take care of the rounding errors that can propagate through the first stage of the likelihood computations (depicted in the top part of Figure 3) if the dependency graph between the new symbols is deep. Let

$$c_{o_i} = \sum_j \sum_k (Co_i)_{jk}, \text{ for } i = 1, \ldots, M$$

be the sum of all entries in C_{o_i} for all symbols in the original observed sequence, and let $\bar{C}_{o_i} = C_{o_i}/c_{o_i}$ be the corresponding normalised matrix. Now for each new symbol $o_i = (l_i, r_i)$ in the compressed sequence define

$$c_{o_i} = \sum_j \sum_k (\bar{C}_{l_i}\bar{C}_{r_i})_{jk}, \text{ for } i = M+1, \ldots, M',$$

and let

$$\bar{C}_{o_i} = \frac{\bar{C}_{l_i}\bar{C}_{r_i}}{c_{o_i}}, \text{ for } i = M+1, \ldots, M'.$$

Finally let

$$s_{o_i} = \begin{cases} c_{o_i}, & \text{, if } i = 1, \ldots, M \\ c_{o_i} c_{l_i} c_{r_i}, & \text{, if } i = M+1, \ldots, M'. \end{cases}$$

Then

$$\begin{aligned} \alpha_T &= C_{y'_{T'}} C_{y'_{T'-1}} \cdots C_{y'_2} C_1 \\ &= \left(\prod_{t=2}^{T'} s_{y_t}\right) \bar{C}_{y'_{T'}} \bar{C}_{y'_{T'-1}} \cdots \bar{C}_{y'_2} C_1 \end{aligned} \tag{4}$$

Thus to handle the underflow in the first stage, we compute s_{o_i} along with $\bar{C}_{o_i}$ for $i = 1, \ldots M'$ (see Figure 4) and compute the product above in the second stage of the likelihood computation.

However, the $\bar{C}_{o_i}$ matrices still only contain values between 0 and 1, and their product will therefore still tend towards 0 exponentially fast, causing underflow. To prevent this we introduce a scaling factor d_i for each of the $T' - 1$ matrix-vector multiplications in (4), set to be the sum of the entries in the resulting vector. Each d_i is used two times: First we normalise the corresponding resulting vector by dividing each entry by d_i, and next we use it to restore the correct result at the end of the computations. Assume that $\bar{\alpha}_T$ is the result of the $T'-1$ normalised matrix-vector multiplications. Then

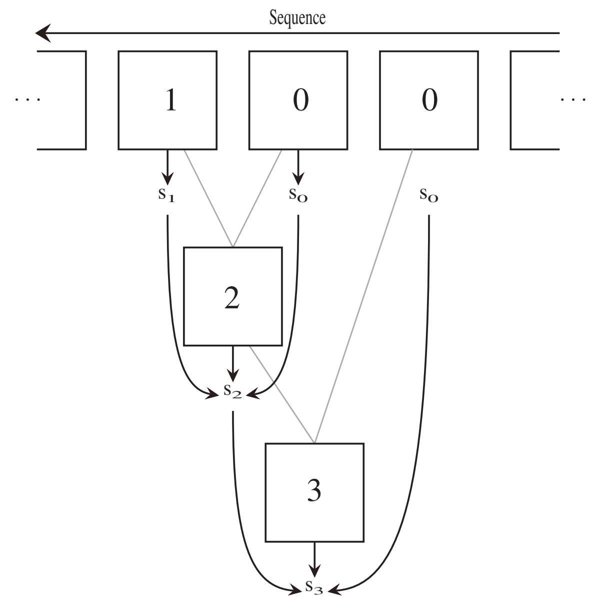


Figure 4 Dependencies between scaling factors. Dependencies between scaling factors. The square boxes are the $\bar{C}_{o_i}$ matrices and s_0, s_1, s_2 and s_3 are scaling factors. The gray lines show the dependencies between the $\bar{C}_{o_i}$ matrices, while the solid black arrows show how the scaling factors are computed. For example, s_0 is computed directly from $\bar{C}_0$ as the sum of all its entries. s_2 is the sum of all entries in $\bar{C}_2$ times s_1 and s_0.

$$\alpha_T = \left(\prod_{t=2}^{T'} s_{y_t}\right)\left(\prod_{j}^{T'-1} d_j\right)\bar{\alpha}_T,$$

and we can compute the final likelihood as

$$\begin{aligned} \Pr(Y_{1:T} \mid \lambda) &= \sum_i \alpha_T(i) \\ &= \sum_i \left(\prod_{t=2}^{T'} s_{y_t}\right)\left(\prod_{j}^{T'-1} d_j\right)\bar{\alpha}_T(i) \\ &= \left(\prod_{t=2}^{T'} s_{y_t}\right)\left(\prod_{j}^{T'-1} d_j\right)\sum_i \bar{\alpha}_T(i) \\ &= \left(\prod_{t=2}^{T'} s_{y_t}\right)\left(\prod_{j}^{T'-1} d_j\right). \end{aligned}$$

Notice, however, that we now risk getting an underflow when computing these products if T' is big. We handle this by working in log-space. Define

$$\tilde{s}_{o_i} = \begin{cases} \log(c_{o_i}) & \text{, if } i = 1, \ldots, M \\ \log(c_{o_i}) + \tilde{s}_{l_i} + \tilde{s}_{r_i} & \text{, if } i = M+1, \ldots, M' \end{cases}$$

and

$$\tilde{d}_i = \log(d_i) \quad \text{, for } i = 1, \ldots, T' - 1.$$

Then

$$\log\left(\Pr\left(Y_{1:T} \mid \lambda\right)\right) = \left(\sum_{t=2}^{T'} \tilde{s}_{y_t}\right)\left(\sum_{j}^{T'-1} \tilde{d}_j\right).$$

Practical implementation details

In our implementation of the preprocessing phase described above, we simply build a map symbol2pair, mapping each new alphabet symbol o_i to its two constituents (l_i, r_i). In each scan every pair of symbols is counted, and the most frequent pair in the previous round is replaced by a new symbol. The data structure being saved in the end is symbol2pair along with two other maps: nstates2alphabetsize and nstates2seq. The map nstates2alphabetsize maps each N^i_{min} to the size of the alphabet M'_j after j iterations, where j is the number of iterations determined by the stopping criterion. The map nstates2seq maps $N^i_{\min}$ to the resulting sequence after j rounds of compression. These maps can be saved to disk for later use along with the original observed sequence.

Given a specific HMM with N states, the first stage of the actual computation of the likelihood builds a list, symbol2matrix, containing the $\bar{C}_{o_i}$ matrices, and a list of scaling values, symbol2scale, containing the $\tilde{s}_i$ values. These are computed in $\mathcal{O}\left(M'N^3\right)$ time in an iteration over the first M' symbols in the symbol2pair map, where M' is the alphabet size saved in nstates2alphabetsize in the preprocessing procedure. In the second stage $\log\left(\Pr\left(Y_{1:T} \mid \lambda\right)\right)$ is computed in $\mathcal{O}\left(T'N^2\right)$ time by

$$\log\left(\Pr\left(Y_{1:T} \mid \lambda\right)\right) = \left(\sum_{t=2}^{T'} \tilde{s}_{y_t}\right)\left(\sum_{j}^{T'-1} \tilde{d}_j\right)\bar{C}_{y'_{T'}}\bar{C}_{y'_{T'-1}}\cdots\bar{C}_{y'_2}C_1, \tag{5}$$

using the two maps created in the first stage. To obtain maximal performance, we use a BLAS implementation for C++ to perform the series of matrix multiplication.

Our implementation uses $\mathcal{O}(Tk)$ space in the preprocessing phase and $\mathcal{O}\left(N^2(T'+M')\right)$ space in the actual computation, where k is the number of N_{min} values supplied by the user, N is the number of states in the HMM used in the actual computation, and M' is the number of symbols in the extended alphabet corresponding to N in nstates2alphabetsize. If the preprocessed data structure is saved to disk, it will take up $\mathcal{O}(Tk)$ space.

We have also implemented the algorithm in a parallelised version. In this version, stage 2 is parallelised much like the implementation in parredHMMlib [11], where the series of matrix multiplications in (5) is split into a number of blocks which are then processed in parallel. Stage 1 can clearly also be parallelised by computing independent $\bar{C}_{o_i}$ matrices in parallel. However, we found that this does not work well in practice, as the workload in stage 1 is not big enough to justify the parallelization. Stage 1 is therefore not parallelised in the library. The parallelisation of stage 2 gives the greatest speedup for long sequences that are not very compressible. This is because the parallelisation in general works best for long sequences [11], and if the input sequence is very compressible then the compressed sequence will be short and more work will be done in the non-parallelised stage 1. The experiments presented in the next section have all been run single-threaded to get a clearer picture of how the runtime of the basic algorithm is influenced by the characteristica of the input sequence and model. But in general a slightly faster running time can be expected if parallelisation is enabled, especially for long sequences of high complexity.

Results and discussion

We have implemented the above algorithms in a C++ library named zipHMM. The code provides both a C++ and a Python interface to the functionality of reading and writing HMMs to files, preprocessing input sequences and saving the results, and computing the likelihood of a model using the forward algorithm described in the previous section. The library uses BLAS for linear algebra operations and pthreads for multi-threaded parallelisation.

Using the library

The library can be used directly in C++ programs or through Python wrappers in scripts.

Using zipHMM from C++

When using the library in C++ the most important objects are from the Forwarder class, which is responsible for both preprocessing sequences, reading and writing the preprocessed data structure, and for computing the likelihood of a hidden Markov model. The code snippet in Figure 5(a) shows a complete C++ program that reads in an input sequence, `f.read_seq(...)`, preprocess it (as part of reading in the sequence), stores the preprocessed structure to disk, `f.write_to_directory(...)`, reads in an HMM from disk, `read_HMM(...)`, and computes the likelihood of the HMM, `f.forward(...)`.

The sequence reader takes the alphabet size as parameter. This is because we cannot necessarily assume that the observed symbols in the input sequence are all the possible symbols the HMM can emit, so we need to know the alphabet size explicitly. It furthermore takes an optional parameter in which the user can specify an estimate, e, of the number of times the preprocessing will be reused. The default value of this parameter is 1.

```cpp
#include <hmm_io.hpp>
#include <forwarder.hpp>
#include <matrix.hpp>

#include <iostream>

using namespace zipHMM;

int main(int argc, char **args) {
    // Create forwarder object and read input
    Forwarder f;
    size_t alphabet_size = 3;
    size_t min_num_of_evals = 500;
    f.read_seq("example.seq", alphabet_size, min_num_of_evals);

    // Save for future runs
    f.write_to_directory("example_prep");

    // Read in a concrete HMM
    Matrix pi, A, B;
    read_HMM(pi, A, B, "example.hmm");

    // Compute the log-likelihoood
    std::cout << "loglikelihood: " << f.forward(pi, A, B) << std::endl;

    return 0;
}
```

(a) C++

```python
from pyZipHMM import *

# Create forwarder object and read input
f = Forwarder.fromSequence("example.seq",
                           alphabetSize = 3,
                           minNoEvals = 500)

# Save for future runs
f.writeToDirectory("example_prep")

# Read in a concrete HMM
pi, A, B = readHMM("example.hmm")

# Compute the log-likelihoood
print "loglikelihood: ", f.forward(pi, A, B)
```

(b) Python

Figure 5 Using the library in C++ (a) and Python (b).

If the preprocessed sequence is already stored on disk, we can simply read that instead like this:

```
Forwarder f;
number_of_states = 4;
f.read_from_directory("example_
preprocessed", number_of_states);
```

This will cause the saved sequence matching $N_{min} \leq 4$ to be read from the directory `example_preprocessed` together with additional information on the extended alphabet used in this sequence.

In the library, HMMs are implicitly represented simply by a vector and two matrices, the π vector of initial state probabilities and the transition, A, and emission, B, matrices as described in the Implementation section. These are all represented in a `Matrix` class, and in the program in Figure 5(a) these are read in from disk. They can also be directly constructed and manipulated in a program. In our own programs we use this, together with a numerical optimisation library, to fit parameters by maximising the likelihood.

The `f.forward(...)` method computes the likelihood sequentially using the preprocessed structure. To use the multi-threaded parallelisation instead, one simply uses the `f.pthread_forward(...)` function, with the same parameters, instead.

For completeness the library also offers implementations of the Viterbi and posterior decoding algorithms. To use these in C++ the headers `viterbi.hpp` and `posterior_decoding.hpp` should be included and the functions `viterbi(...)` and `posterior_decoding(...)` should be called as described in the README file in the library.

Using zipHMM from Python

All the C++ classes in the library are wrapped in a Python module so the full functionality of the zipHMM is available for Python scripting using essentially the same API, except with a more Python flavour where appropriate, e.g. reading in data is handled by returning multiple values from function calls instead of pass-by-reference function arguments and with a more typical Python naming convention. Figure 5(b) shows the equivalent of the C++ code in Figure 5(a) in Python.

Performance

To evaluate the performance of zipHMM we performed a number of experiments using a hidden Markov model previously developed to infer population genetics parameters of a speciation model. All experiments were run on a machine with two Intel Sandy Bridge E5-2670 CPUs, each with 8 cores running at 2.67GHz and having access to a 64Gb main memory. We compare the performance of our forward algorithm to the performance of the implementations of the forward algorithm in HMMlib [19] and in parredHMMlib [11] and to a simple implementation of equation (3) using BLAS to perform the series of matrix-vector multiplications. HMMlib is an implementation that takes advantage of all the features of a modern computer, such as SSE instructions and multiple cores. The individual features of HMMlib can be turned on or off by the user, and we recommend only enabling these features for HMMs with large state spaces. In all our experiments we enabled the SSE parallelisation but used only a single thread. The parredHMM library implements equation (3) as a parallel reduction, splitting the series of matrix multiplications into a number of blocks and processing the blocks in parallel. The parredForward algorithm was calibrated to use the optimal number of threads.

For performance evaluation we wanted to evaluate how well the new algorithm compares to other optimised forward implementations, evaluate the trade-off between preprocessing and computing the likelihood, and explore how the complexity of the input string affects the running time.

Our new implementation of the forward algorithm is expected to perform best on strings of low complexity because they are more compressible. To investigate this we measured the per-iteration running time of the forward algorithm for parredHMMlib, HMMlib and the simple implementation of equation (3) on random binary sequences (over the alphabet $\{0, 1\}$) of length $L = 10^7$ with the frequency of 1s varying from 0.0001 to 0.05, and divided it by the per-iteration running time for zipHMMlib (excluding the preprocessing time) to obtain the speedup factor. This experiment is summarised in Figure 6, where we note that the speedup factor decreases linearly with the complexity of the input sequence; however, speedup factors of more than two orders of magnitude are obtained for less complex sequences, and even for sequences of low complexity a (modest) speedup is obtained.

In the rest of our experiments, we used a coalescent hidden Markov model (CoalHMM) from [17] together with real genomic data for the experiments. A CoalHMM

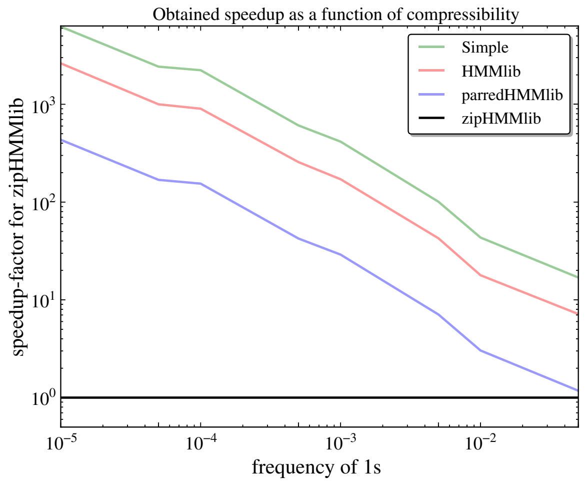


Figure 6 Speedup vs. sequence complexity. The speedup factor decreases linearly with the sequence complexity.

[7,8,12] exploits changing gene-trees along a genomic alignment to infer population genetics parameters. The "Isolation-with-Migration" CoalHMM from [17] considers a pairwise alignment as its observed sequence and a discretisation of the time to the most recent common ancestor, or "coalescence time", of the aligned sequences as its hidden states. The coalescence time can change from any point to another, so the transition matrix of the CoalHMM is fully connected, and the number of hidden states can be varied depending on how fine-grained we want to model time. Varying the number of states lets us explore the performance as a function of the number of states. The performance as a function of the length of the input was explored by using alignments of varying length. Finally, to explore how the complexity of the string affects the performance we used alignments of sequences at varying evolutionary distance, since closer related genomes have fewer variable sites and thus the alignments have lower complexity. The CoalHMM model uses a Jukes-Cantor model in its emission probabilities and thus only distinguishes between if a specific site has two identical nucleotides or two different nucleotides in the alignment. We therefore also varied the complexity of the strings by compressing either the actual sequence alignment or summarising it as an alphabet of size three, $\{0, 1, 2\}$, for identical sites, differing sites, or missing data/gaps. This way we obtain sequences with alphabets of size $M = 3$ ($\{0, 1, 2\}$), $M = 16$ (full alignments over $\{A, C, G, T\} \times \{A, C, G, T\}$, where columns with missing data or gaps were deleted) and $M = 25$ (full alignments over $\{A, C, G, T, N\} \times \{A, C, G, T, N\}$, where N is either missing data or a gap).

For all experiments we trained the model using the Nelder-Mead-simplex algorithm and measured the preprocessing time and total optimisation time, and the expected number of likelihood computations was set to $e = 500$.

Figure 7(a) shows how the performance of zipForward changes, when the size of the model is increased. We note that the total time, as expected, depends very heavily on the number of states (the time complexities of stage 1 and 2 are qubic and quadratic in the number of states, respectively), while the time used on preprocessing, also as expected, varies very little and shows no clear pattern as the number of states is increased.

Figure 7(b) shows how the runtime for training the CoalHMM with zipForward changes when the sequence length is varied. We expected the runtime to increase with the sequence length, however this is not what the results show for the shorter sequences. This is due to the optimisation procedure, which required more iterations of the likelihood computation for the shorter sequences than for the longer sequences. For the longest sequences the runtime grows sublinearly, which was expected, since longer sequences often compress relatively more than shorter sequences.

We expected alignments of sequences at short evolutionary distance to be more compressible than alignments of sequences at longer evolutionary distance, and therefore expected the training procedure to be faster for alignments of sequences at short evolutionary distance. We recognise this in Figure 7(c) except for the sequences with $M = 16$, where the human-orangutan alignment was processed faster than the human-gorilla alignment. However, this was caused by the trade-off between the time for the preprocessing and the time for the actual training procedure: the preprocessing procedure took significantly longer time for the human-gorilla alignment (because it was more compressible) than for the human-orangutan alignment, but this extra time was not all gained back in the training procedure, although the compressed sequence indeed was shorter (the per-iteration running time for the human-gorilla alignment was 7.913s and 8.488s for the human-orangutan alignment).

We also expected the total time of the training procedure to increase as the number of symbols in the initial alphabet was increased, because sequences with small initial alphabets are expected to be more compressible than sequences with larger initial alphabets. But as Figure 7(c) shows, the sequences with an initial alphabet of size $M = 25$ were processed faster than the sequences with an initial alphabet of size $M = 16$. This is again caused by the optimisation procedure, which converges faster for the sequences with $M = 25$ than for the sequences with $M = 16$ (e.g. for the human-gorilla alignments, the number of evaluations of the likelihood were 860 and 1160 for $M = 25$ and $M = 16$, respectively). This may be a result of

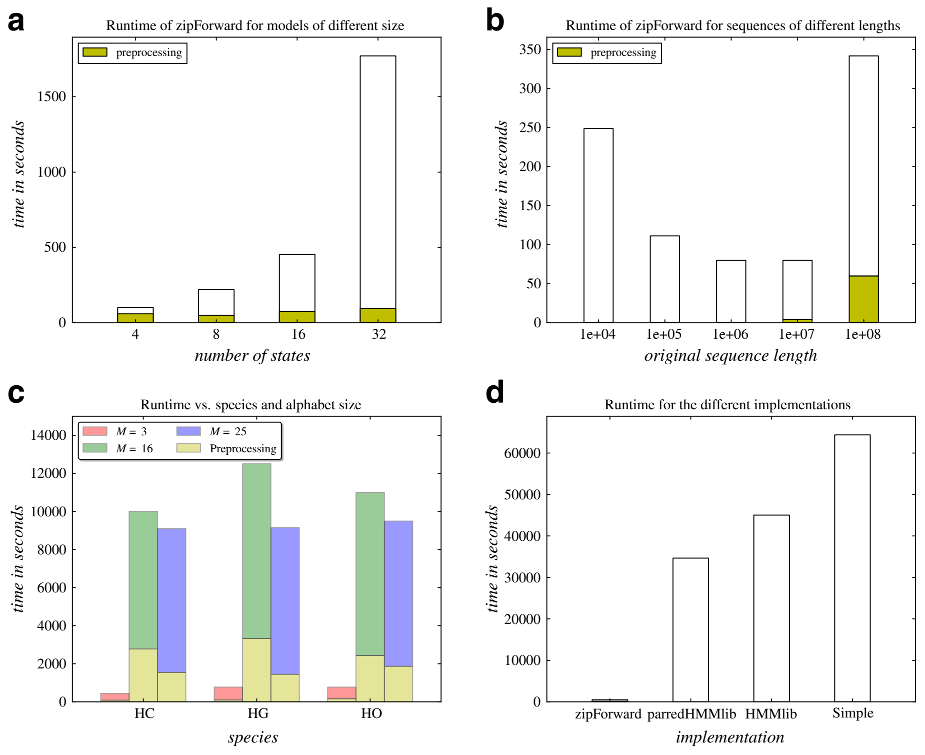


Figure 7 Performance experiments with the CoalHMM framework, showing the running time of zipHMMlib as a function of (a) model size, (b) sequence length, (c) sequence complexity, and (d) compared to other implementations.

the sequences with $M = 25$ containing more information than the sequences with $M = 16$. The lengths of the compressed sequences and the per-iteration running times match our expectations, and for the sequences with $M = 3$ and $M = 25$, which contain the same data, the algorithm behaves as expected.

Figure 7(d) shows the performance of the four different implementations of the forward algorithm. Each of the four algorithms was used to train the CoalHMM on an alignment of the entire human and chimpanzee chromosome 1, using an HMM with 16 states and an initial alphabet of size 3. The training procedure was finished in 7.4 minutes (446 seconds) using zipForward and including the preprocessing time. This gives a speedup factor of 77.7 compared to the previously fastest implementation using parredHMMlib, which used 9.6 hours (34, 657 seconds). It is therefore evident that zipForward is clearly superior to the three other implementations on this kind of input. The time used per iteration of the likelihood computation was 0.5042 seconds for zipHMMlib, while it was 46.772 seconds for parredHMMlib, leading to a speedup of a factor 92.8 on the actual optimization procedure (excluding preprocessing time). Repeating the same experiment on full alignments over alphabets of size 16 and 25 (not shown here), where zipForward clearly performs worse than for sequences with alphabets of size 3 (see Figure 7(c)), we still obtained total speedup factors of 4.4 for both experiments.

Conclusions

We have engineered a variant of the HMM forward algorithm to exploit repetitions in strings to reduce the total amount of computation, by exploring shared subexpressions. We have implemented this in an easy to use C++ library, with a Python interface for use in scripting, and we have demonstrated that our library can be used to achieve speedups of 4 - 78 factors for realistic whole-genome analysis with a reasonably complex hidden Markov model.

Availability and requirements

Project name: zipHMM
Project home page: http://birc.au.dk/software/ziphmm/
Operating system(s): Platform independent
Programming language: C++, Python
License: LGPL

Competing interests

The authors declare that they have no competing interests.

Authors' contributions

AS, MK and TM developed the algorithms. AS implemented the library. AS, CNSP and TM designed experiments. AS, CNSP and TM drafted the manuscript. All authors read and approved the final manuscript.

Acknowledgements

TM receives funding from the Danish Council for Independent Research. We also thank the three anonymous reviewers for their helpful comments which helped to improve this paper.

Received: 18 June 2013 Accepted: 14 November 2013
Published: 22 November 2013

References

1. Churchill GA: **Stochastic models for heterogeneous DNA sequences.** *Bull Math Biol* 1989, **51:**79–94.
2. Burge C, Karlin S: **Prediction of complete gene structures in human genomic DNA.** *J Mol Biol* 1997, **268:**78–94.
3. Krogh A, et al.: **Predicting transmembrane protein topology with a hidden Markov model: application to complete genomes.** *J Mol Biol* 2001, **305**(3):567–580.
4. Bateman A, Coin L, Durbin R, Finn RD, Hollich V, Griffiths-Jones S, Khanna A, Marshall M, Moxon S, Sonnhammer EL, et al.: **The Pfam protein families database.** *Nucleic Acids Res* 2004, **32**(suppl 1):D138–D141.
5. Eddy S: **Profile hidden Markov models.** *Bioinformatics* 1998, **14**(9):755.
6. Siepel A, Haussler D: **Phylogenetic hidden Markov models.** In *Statistical Methods in Molecular Evolution*. Edited by Nielsen R. New York: Springer; 2005:325–351.
7. Hobolth A, et al.: **Genomic relationships and speciation times of human, chimpanzee, and gorilla inferred from a coalescent hidden Markov model.** *PLoS Genet* 2007, **3**(2):e7.
8. Dutheil JY, et al.: **Ancestral population genomics: the coalescent hidden Markov model approach.** *Genetics* 2009, **183:**259–274.
9. Kern AD, Haussler D: **A population genetic hidden Markov model for detecting genomic regions under selection.** *Mol Biol Evol* 2010, **27**(7):1673–1685.
10. Lifshits Y, Mozes S, Weimann O, Ziv-Ukelson M: **Speeding up HMM decoding and training by exploiting sequence repetitions.** *Algorithmica* 2009, **54**(3):379–399.
11. Nielsen J, Sand A: **Algorithms for a parallel implementation of hidden Markov models with a small state space.** In *Proceedings of the 2011 IEEE IPDPS Workshops & PhD Forum, IEEE* 2011:452–459. Anchorage, Alaska, USA. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6008865
12. Mailund T, et al.: **Estimating divergence time and ancestral effective population size of Bornean and Sumatran Orangutan subspecies using a Coalescent Hidden Markov model.** *PLoS Genet* 2011, **7**(3):e1001319.
13. Locke DP, et al.: **Comparative and demographic analysis of orang-utan genomes.** *Nature* 2011, **469**(7331):529–533.
14. Hobolth A, et al.: **Incomplete lineage sorting patterns among human, chimpanzee, and orangutan suggest recent orangutan speciation and widespread selection.** *Genome Res* 2011, **21**(3):349–356.
15. Scally A, et al.: **Insights into hominid evolution from the gorilla genome sequence.** *Nature* 2012, **483**(7388):169–175.
16. Prüfer K, et al.: **The bonobo genome compared with the chimpanzee and human genomes.** *Nature* 2012, **486:**527–531.
17. Mailund T, et al.: **A new isolation with migration model along complete genomes infers very different divergence processes among closely related great ape species.** *PLoS Genet* 2012, **8**(12):e1003125. http://dx.doi.org/10.1371.
18. Rabiner L: **A tutorial on hidden Markov models and selected applications in speech recognition.** *Proc IEEE* 1989, **77**(2):257–286.
19. Sand A, et al.: **HMMlib: A C++ Library for General Hidden Markov Models Exploiting Modern CPUs.** In *2010 Ninth International Workshop on Parallel and Distributed Methods in Verification/2010 Second International Workshop on High Performance Computational Systems Biology., IEEE* 2010:126–134. Enschede, The Netherlands. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5698478

doi:10.1186/1471-2105-14-339
Cite this article as: Sand *et al.*: zipHMMlib: a highly optimised HMM library exploiting repetitions in the input to speed up the forward algorithm. *BMC Bioinformatics* 2013 **14**:339.