HMMoC – a hidden Markov model compiler

Version 1.3, February 2008
Gerton Lunter

1 Introduction

Do you use hidden Markov models, are you annoyed at having to re-implement standard algorithms, do you need the flexibility and efficiency of C++, and are you, basically, a lazy programmer? Then HMMoC is for you!

The HMMoC compiler translates a high-level description of a hidden Markov model into efficient C++ code for computing likelihoods, posterior probabilities, for path decoding, and for training. It was designed to deal with HMMs occurring in computational biology, but it is completely general and not tied to any particular application. This document is a very brief manual for HMMoC. It mainly describes the format of HMMoC's XML input files. At the end of the manual, there is a brief description of four examples.

2 Supported features

- Forward, Backward, Viterbi and Baum-Welch algorithms
- Multiple output tapes (”pair HMMs”, ”triple HMMs” etc.)
- Banding and sparse dynamic programming tables
- Extended-exponent reals (both precise and efficient; no need to work in log-space)
- Log space reals for the stubborn (or when you're interested only in the Viterbi algorithm)
- Higher-order states (transitions and emissions may depend on previously emitted symbols)
- Inhomogeneous chains (Position-dependent transition and emission probabilities)
- Mealy and Moore machines, and mixtures (emissions are associated to states or transitions)
- Generalized HMMs (states and transitions may emit more than one symbol at a time)
- Silent states are dealt with using matrix inversion if necessary

- Memory-efficient implementations of Forward/Backward/Baum-Welch.
- State grouping into cliques for better memory efficiency
- XML Macro facilities

For a fuller description of the features of HMMoC, please see this paper: Lunter, G.A., *HMMoC - a compiler for hidden Markov models*, Bioinformatics, 2007 Jul 10. `doi:10.1093/bioinformatics/btm350`.

3 Requirements

You need Java (version 1.4 or later) to run HMMoC. Type `java` to find out if you have it. If not, download the end-user Java Runtime Environment from `java.com`, or the development kit from `java.sun.com`.

The Gnu Java interpreter (gij) and the run-time library at some point did not work correctly with HMMoC, but the latest version (4.1.2) works fine. HMMoC has been tested with the Gnu C++ compiler `gcc`, versions 4.0.0, 4.0.1, 4.0.2 and 4.1.2, and Microsoft Visual C++ 2005. Older versions of both Sun java and `gcc` did give problems, so you might want to upgrade in case things don't work. Version 4.1.1 of `gcc` seemed to be broken.

HMMoC requires the jdom library to run. This library is pre-installed in the `lib/` directory. The `hmmoc` script will run HMMoC for the default directory structure; if you want to use a different library version, you'll have to edit this file. The jdom library can be obtained from `www.jdom.org/dist/binary/jdom-1.0.tar.gz`.

4 Installation

There's no need to install `hmmoc` – simply type `bin/hmmoc` to run it.

To make the documentation `.pdf` file, type `make`. The documentation should now be in `doc/manual.pdf`. You need `pdflatex` for this to work. A package `times.sty` will be used if it exists; if not, simply ignore the warning by pressing `enter` twice.

If you want to re-compile the `.jar` file, type `make jar` in the top `hmmoc` directory.

5 Input format – HMM description

The input to HMMoC splits logically into two parts: definition of the HMM, and specification of the algorithms. First we describe how to define the HMM.

An HMM is defined by its state topology, the number of tapes it emits symbols on, and the set of transition and emission probabilities. The probabilities are defined by pieces of C or C++ code that will appear in the final C++ algorithms. All of these are defined in by XML code.

The ordering and, to some extent, the location of many XML elements in the file is not important. This is most useful for the <code> and <macro> elements, which may be defined anywhere, and re-used. It is also useful for the <hmm> element, so that entire HMMs may be used several times, to generate the various algorithms. Every element may have an "id" attribute, which can then be referred to. When certain elements need to appear as children of another element, they can be referred to by an idref, and be defined somewhere else:

```
<code idref="identifier-of-code-element-defined-elsewhere"/>
```

As a shortcut, whenever an element can have text, it can have a "value" attribute that contains the intended text instead. For example, these two elements are equivalent:

```
<code id="sqrt2">
  1.0/sqrt(2.0)
</code>

<code id="sqrt2" value="1.0/sqrt(2.0)"/>
```

The root element is <hml>, for "HMM markup language". The element that makes HMMoC produce output is "codeGeneration", which indirectly refers to all other elements. We begin at the other end, and define the building blocks first.

5.1 alphabet

The ”alphabet” element defines an alphabet of allowed emission symbols. All non-whitespace characters appearing as text are taken to be allowed symbols.

In addition, ”code” elements (see later) evaluating to a single byte-sized value may be used to include e.g. whitespace symbols.

A range of characters can be represented by the <range> child element. This element has two attributes, ”from” and ”to”, giving the start (inclusive) and end (exclusive) of the intended range, in decimal.

Example:

```
<alphabet id="nucleotides">
 ACGT
 <code> '\n' </code>
</alphabet>

<alphabet id="codons">
 <range from="0" to="64"/>
</alphabet>
```

5.2 code

This is a very general element, used for any piece of C++ code that will appear in the final algorithm. It has a number of possible attributes and children, depending on the role of the element. This role can be (1) to compute a value, (2) to initialize temporary variables, (3) to define an input parameter, (4) to associate an output to a coordinate for banding (see below) or (5) to generate a piece of verbatim code in the output file.

Attributes:

”value” The actual program code. Shortcut for small code elements; equivalent to text content.

”type” One of ”parameter”, ”statement”, “coordinate” or ”expression” (the default). A statement computes the result and assigns it to some variable (see ’identifier’ below). An expression is used for simpler computations, where a single formula (or number) suffices. Parameters specify an input parameter that are to be fed into the function containing a piece of code, and makes sure that it will appear in the function’s header. Parameters can appear only as sub-elements of ”code” or ”output” elements. For the use of coordinates, refer to the <banding> element below.

”language” This is just to remind me that I intended to support Java as well. Currently defaults to C++, the only valid choice.

”init” Refers to another code element that needs to appear before the current code element is used. This may be some initialization code, a parameter declaration or definition, a #include, etc. By default the code will appear after the function’s variable declarations, and before any loops, but this can be altered by the ”where” attribute.

”where” Used to specify the location in the output source file for an ”init” code element. Possible values are “header-includes”, ”includes”, ”classdefinitions”, ”declarations”, ”subroutines”, which will appear in the output file in that order. If no ”where” attribute is specified, the init element will appear in the function itself, before any loops but after the variable declarations.

“output” Used only when type=”coordinate”, and specifies the coordinate within a position vector corresponding to an <output> sequence. See <banding>.

Content:

Text: The actual code. Often enclosed in <![CDATA[...]]> brackets to quote < and & characters.

<code>: A child code element specifies a parameter that the current code’s text refers to.

<identifier>: Used to specify input variables (for emitted symbols, for example), or the output variable (for statements). See below.

Example:

```
<!-- Code to compute the value 1.0 -->
<code id="one-expr"> 1.0 </code>

<!-- Same, but using a statement instead of an expression -->
<code id="one-stmt" type="statement">
  iP = 1.0;
  <identifier type="output" value="iP"/>
</code>

<!-- This returns a variable value, obtained from a parameter that
     should be passed to the subroutine that uses this code -->
<code id="var">
  iVar
  <code type="parameter" value="double iVar"/>
</code>

<!-- The ’log2’ code returns log(2.0), but computes it just once -->
<code id="initlog2">
  const double cLog2 = log(2.0);
</code>

<code id="log2" init="initlog2" value="cLog2"/>
```

5.3 identifier

This element binds an identifier that is used in a surrounding `<code>` element to an ’internal variable’ that represents an output (for instance, a probability that is computed by the code element) or an input (for instance, a symbol in a sequence, or the sequence length). HMMoC will substitute the identifier with a compiler-generated variable that either contains the expected value (for an input), or will be expected to contain the computed value (for an output).

Internal variables must always be bound to some identifier; if not, HMMoC generates an error message. However, internal variables referring to sequence symbols may be omitted, making the code element independent of those symbols.

Note that all identifiers must be simple expressions that may serve as a C++ identifier. For instance, it is not permitted to use an expression like ”inputSequence.size()” to specify the sequence length.

Substituting the identifier by the actual variable is done by simple text replacement. To make sure that no undesired substitutions will occur, make sure that the identifier is unique, and not a prefix of any other identifier. A good way to avoid problems is to surround any identifier by underscores, e.g. `_position_` (and not use underscores internally).

The main use of this element is to specify a variable to hold the currently emitted symbol for an <emission> element. However, <identifier> elements can be part of any <code> element, including those specifying probabilities for transitions. Either of these can therefore be made to depend on (previously emitted) symbols, as well as the position within the sequence.

Attributes:

”value” Shortcut; equivalent to text content.

”type” Specifies the meaning of this identifier. It can be ”length” or ”sequence” if the identifier is child of an ”output” element (see below), or ”result” if it is child of a ”code” element containing a statement. It may also be “position”, in which case the identifier refers to the position (0-based) within some sequence specified an <output> element. By default, the identifier refers to an emitted symbol. Which symbol this is, is specified by the ”output” and ”depth” attributes.

”output” The output tape; for emitted symbols or position identifiers.

”depth” Specifies the lookback depth, for higher-order Markov chains. It defaults to ”0”, for the symbol that is actually emitted. Symbols at higher depths are previously emitted, and the probability can be conditional on such symbols. Transition probabilities may also be dependent on previously emitted symbols, and depth values from 1 upwards refer to these symbols (regardless of whether this transition emits a symbol or not). Note that the default ”0” is not useful in the context of transitions.

”height” The converse of ’depth’; it is used in the case of generalized HMMs, to specify which of the multiple symbols this identifier refers to. Also defaults to 0, referring to the first emitted symbol. It cannot be used together with ’depth’.

Content:

Text: The identifier.

5.4 output

The output element defines an emission tape.

Attribute:

“speed” Defines the ordering of the coordinates, from 0 (fast, representing the innermost loop) to increasingly slower (higher integers, representing outer loops). For efficiency, assign the slowest coordinate to the longest sequence. The speed ordering also determines in which order the coordinates are listed in the `FoobarDPTable::getProb()` function (fastest first). The speed attribute is not required, except when banding is used; then the slowest variable must be nondecreasing (for forward iterations) or nonincreasing (for backward iterations).

Content:

<alphabet> The alphabet used on this tape.

<identifier> Two identifiers are required, one of type ”length”, one of type ”sequence”. These specify the variables holding the length and content of the sequence, as an integer variable, and a char array (or more generally, any type for which operator[] returns a char).

<code> These optional elements specify parameters; usually the variables introduced by the identifiers for this particular sequence.

Example:

```
<output id="sequence1">
 <alphabet idref="nucleotides"/>
 <identifier type="sequence" value="iFirstSequence"/>
 <identifier type="length" value="iLen1"/>
 <code type="parameter" value="char* iFirstSequence"/>
 <code type="parameter" value="int iLen1"/>
</output>
```

5.5 probability

Specifies that a ”code” element computes a probability. Not doing anything useful besides; I might get rid of this element.

Content:

<code> The code element computing the probability.

5.6 emission

This element specifies a particular emission on a set of output tapes.

Content:

<probability> Specifies the emission probability.

<output> Specifies a tape to output on. Tapes not listed here will be silent for this emission. The ordering of tapes is not significant; ”output” and ”depth” attributes in the ”code” element are used to bind particular identifiers to output tapes. To emit multiple symbols onto one tape, simply specify the same <output> tape multiple times (and make sure that identifiers are bound to all relevant “height”s in the <code> element in the <probability> element.)

Example:

```
<emission id="emita">
 <output idref="sequence1"/>
 <probability>
  <code type="statement">
   <identifier output="sequence1" value="iSymbol"/>
   <!-- This is a statement, so specify the variable that holds the result -->
   <identifier type="result" value="iProb"/>
   <![CDATA[
     switch (iSymbol) {
     case ’A’: iProb = 0.15; break;
     case ’C’: iProb = 0.15; break;
     case ’G’: iProb = 0.35; break;
     case ’T’: iProb = 0.35; break;
     }
   ]]>
  </code>
 </probability>
</emission>
```

Note the use of the ”<![CDATA[” and ”]]>” tags. These are the quote tags of XML, and allow you to freely use, amongst others, the < and > symbols.

To make the emission dependent on the position, add an `<identifier>` element with `type=’’position’’` to the `<code>` element. The position referred to is that reached before emitting the symbol(s).

5.7 state

This element simply labels a state, by its ”id” attribute.

Attribute:

”emission” If not specified, the state is a Mealy state and emissions are specified by transitions going to this state. If present, the state is a Moore state, and transitions going to this state may not specify an emission. The attribute’s value refers to an `<emission>` element.

Content:

<order> If not specified, this state is a zeroth-order state. If present, it specifies the order of this state with respect to a particular output tape. The "order" element has an attribute "output" and an attribute "depth" (which should be at least 1). More than one "order" element may be present, to specify the order w.r.t. several output tapes. Not every assignment is valid: the order of a destination state may be at most the order of the originating state, increased by one if a symbol is emitted. For a given state, this requirement holds for every tape, and for every transition into that state. Different transitions may result in different requirements, and if this happens, the most stringent is used (and a warning is printed). By definition, the order of the start state is 0 for all tapes.

5.8 clique

Cliques are sets of states such that once a clique is left, it can never be entered again. In other words, the graph induced on the cliques by the transitions on the states should be a-cyclic. It is permissible to group all states into a single clique, but this gives less efficient code. At least, the start and end states can always be put a clique of their own.

Content:

<state> Specifies a member state.

<banding> Specifies a band in coordinate space to which the recursion should be restricted.

<code> Specifies the type of dynamic programming table to use. By default (i.e. when no `<code>` element is provided) this is `DPTable`, which is implemented as a multidimensional array. A sparse dynamic programming table (implemented using a hash map) is also available, as `SparseDPTable`. To use this table, add an element `<code idref=''SparseDPTable''/>`. To use a custom table, specify a template identifier (such as `MyDPTable`) in the body of the `<code>` element, and define a template that takes two parameters: a state array (itself a template), and an integer specifying the number of dimensions. See `dptables.h` for an example implementation.

5.9 banding

This element, which is a child of a <clique> element, allows you to restrict the recursion to a 'band' in the dynamic programming table, where most of the probability is supposed to be concentrated. HMMoC can perform no magic, you will need to figure out yourself what is a reasonable restriction.

To implement a band, you must implement a class derived from `Banding<dim>`, where `dim` is the dimension of the dynamic programming table. This class implements two iterators, which loop over the restricted band of the DP table in the forward and backward directions. The forward iteration is started by a call to `Banding<dim>::forwardIterator()`, which initializes and returns a position (of type `Banding<dim>::Position`), which is a reference to an `int[dim]` array. At the end of each iteration, HMMoC calls `Banding<dim>::hasNextForward()`, which updates the position and returns `true` if successful. The analogous functions for backward iterations are `Banding<dim>::backwardIterator()` and `Banding<dim>:hasNextBackward().`

HMMoC has limited sanity checking: it checks whether positions are within boundaries. If not, no computation is performed, and `Banding<dim>:warning()` is called, but the iteration will continue. What HMMoC does not do is to make sure that the iteration is done in the right direction. For forward (backward) iterations, this means that when a position v is visited, all positions of the form $v-e$ $(v+e)$ have either already been visited, or will not be visited at all. Here e is any nonzero vector with nonnegative entries.

It is usually sensible to combine banding with a sparse dynamic programming table - otherwise the only gain is execution time, not memory usage. See section 7.1 for details.

Content:

<code> Two type of <code> elements define a <banding> element. A single element of type 'expression' defines the instance of a Banding<dim> element that will be used for iteration. Then, each of the dim sequences that are emitted to in this clique, one <code> element of type 'coordinate' must associate the sequence (i.e., <output> element) to a coordinate in the Position variable. The <output> element is referred to by an 'output' attribute, and the (0-based) coordinate given as a 'value' attribute.

Example:

```
<banding id="clique2banding">
 <code>
   <!-- Use a parameter to pass the banding instance to the functions: -->
   <code type="parameter" value="Banding<2> bandingInstance"/>
   bandingInstance
 </code>
 <code type="coordinate" output="sequence1" value="0"/>
 <code type="coordinate" output="sequence2" value="1"/>
</banding>
```

Example implementation of a Banding element:

```
class MyBanding : Banding<2> {

  int iLen1, iLen2;
  Position pos;

 public:

  MyBanding( int iLen1, int iLen2 ) : iLen1(iLen1), iLen2(iLen2) {}

  Position& forwardIterator() {
    pos[0] = pos[1] = 0;
    return pos;
  }

  bool hasNextForward() {
    if (pos[0]<iLen1) {
      ++pos[0];
      return true;
    }
    if (pos[1]<iLen2) {
      pos[0] = 0;
      ++pos[1];
      return true;
    }
    return false;
  }

  /* implementations for backward iteration skipped */

  void warning() {
    cout << "Warning - out of bounds at position ("
         << pos[0] << "," << pos[1] << ")" << endl;
  }

};
```

5.10 transition

What can I add to that.

Attributes:

”from”	Originating state for this transition
”to”	Destination state for this transition
”probability”	Probability for this transition
”emission”	Emission for this transition. Should not be specified if ”to” state is Moore.

Example:

```
<probability id="half">
  <code value="0.5"/>
</probability>

<transition from="start" to="end" probability="half" emission="emita"/>
```

To make a transition position-dependent, add an `<identifier>` element with `type=''position''` to the `<code>` element, just as with emissions. The position referred to is that of the tail (’from’) end of the transition.

5.11 hmm

The hmm element combines all elements mentioned above to describe a hidden Markov model.

Content:

<description>	An element containing some descriptive text
<outputs>	An element containing ”output” elements, listing all the emission tapes for this HMM
<graph>	An element containing ”clique” elements, together listing all states for this HMM
<transitions>	An element containing ”transition” elements between the states

The identifier of the hmm element (given as usual by the ’id’ attribute), ends up as part of the name of the dynamic programming table and Baum-Welch classes (so that several HMMs can be defined in a single XML file). For this reason it is a good idea to capitalize the hmm’s name.

6 Input format – Macro facility

Occasionally, you will need to write a lot of repetitive XML code to specify your HMM. To ease the burden somewhat, HMMoC includes a simple macro template facility. A macro is any XML expression or subtree, which may be used several times in any other part of the XML document. It can contain ”slots”, simple unique strings that are replaced by arbitrary other strings when the macro is used. These slots may appear in attribute values or text content (both normal and CDATA sections).

Slot variables may be any string, but may not contain any of the following special characters:

```
\ ( ) [ ] { } . ^ $ ? * + $
```

One way to make reasonably sure that no unintended replacements get made is to surround the slot variable by underscores.

The macro facility is very simple, and does not allow any programming (e.g. looping, or conditionals). The idea was to keep the language simple, and anyway it is easy to write a small Python (or Perl, if you must) script to generate the XML file you need.

Since macros sometimes make errors hard to find, HMMoC has a macro debugging facility. If you include the ”debug” attribute (with any string value) on the top-level ”hml” tag, HMMoC will produce a ”.debug” file that holds the expanded XML file, just before it is interpreted by the compiler.

6.1 macro

This defines the macro. Its content is completely free. It must have a single ”id” attribute, which is used to refer to this macro. When the XML file is processed, the macro definition is removed from the document. Macro definitions may appear anywhere in the document.

Attributes:

”id” The label used to refer to this macro (required)

6.2 expand

This uses the macro. The ”expand” element gets replaced by the macro body. The macro tag is also removed at replacement.

Attributes:

”macro” The macro to be expanded.

Content:

<slot> This element specifies a slot, or variable, to be filled in the macro body. The slot has a ”var” attribute, which specifies the key string. Wherever this string appears as a substring in attribute values, text content or CDATA sections, it will be replaced by the slot value specified in the ”value” attribute, or alternatively, by the textual content of the slot element. Macros are allowed to have several slots.

Example:

The fragment

```
<macro id="macro1">
 <transition from="state_cls_1" to="state_cls_2" emission="emit_cls_"/>
 <transition from="state_cls_2" to="state_cls_3" emission="emit_cls_"/>
 <transition from="state_cls_3" to="state_cls_4" emission="emit_cls_"/>
</macro>
<expand macro="macro1"><slot var="_cls_" value="A"/></expand>
<expand macro="macro1"><slot var="_cls_" value="B"/></expand>
<expand macro="macro1"><slot var="_cls_" value="C"/></expand>
```

is expanded to

```
 <transition from="stateA1" to="stateA2" emission="emitA"/>
 <transition from="stateA2" to="stateA3" emission="emitA"/>
 <transition from="stateA3" to="stateA4" emission="emitA"/>
 <transition from="stateB1" to="stateB2" emission="emitB"/>
 <transition from="stateB2" to="stateB3" emission="emitB"/>
 <transition from="stateB3" to="stateB4" emission="emitB"/>
 <transition from="stateC1" to="stateC2" emission="emitC"/>
 <transition from="stateC2" to="stateC3" emission="emitC"/>
 <transition from="stateC3" to="stateC4" emission="emitC"/>
```

7 Input format – algorithm specification

7.1 forward, backward, sample, viterbi

These specify the algorithm to build, for a specific HMM. This is also the place where the implementation of the dynamic programming table is controlled.

Attributes:

”name” The name of the function as it will appear in the code

”outputTable” Either ”yes” or ”no”, defaults to ”no”. Specifies whether the dynamic programming table should be retained or not. Retaining a DP table requires more memory for the forward and backward algorithms, but is needed for sampling, calculation of posterior likelihoods, and Viterbi traceback. This attribute has no meaning for the ”sample” algorithm.

”baumWelch” ”yes” or ”no”, or “transitions”, or “emissions”. Can only be used with the forward or backward algorithm, and then requires additional input of the complementing (backward or forward) dynamic programming table. Computes posterior usage counts of the transitions or emissions or both.

“cacheValues” “yes” (default) or “no”. Specifies whether emission and transition probabilities must be computed once and re-used (the default), or computed every time they’re needed. The default is almost always the best choice, except for large HMMs with few shared emission and transition probabilities among its nodes and edges where choosing “no” may give a small performance boost.

Content:

<hmm> The HMM for which to generate this algorithm.

Note that the sampling algorithm requires the backward dynamic programming table as input, not the forward table.

7.2 codeGeneration

Ask the compiler to generate code.

Attributes:

”file” File to write code to.

“header” Header file. If not present, all code is output to a single file.

”realtype” Can be ”double”, ”bfloat” or “logspace”. Double values go down to about 10^{-300}, after which they underflow. BFloats (”more buoyant floats”) have essentially unlimited exponents, (down to about $10^{-31000000000}$) and the same precision as ordinary floats, while requiring the same memory as doubles. Logspace reals are slow and should be avoided, except possibly for the Viterbi algorithm. Note that it is not possible to mix real number types within a single output unit. (Defaults to ”double”).

”language” Must be ”C++”.

Content:

<forward> Generates forward algorithm.

<backward> Generates backward algorithm.

<sample> Generates sample algorithm.

<viterbi> Generates Viterbi algorithm.

<code> Simply dumps code in file.

8 Remarks relating to efficiency

8.1 Compiling

A great deal of HMMoC efficiency rests on extensive inlining. Normally GCC will not allow code the grow too much, so it has to be told to be more lenient. Other optimizations also make a noticable difference (in efficiency and compile time). The following options for GCC generally give good results:

```
-O3 -ffast-math -finline-limit=1000
```

For large HMMs, excessive inlining can result in cache misses. Try reducing the inline limit when this happens.

8.2 Pre-computing transition and emission probabilities

Transition probabilities are computed once using the code supplied in the xml file, and stored in an array for fast retrieval. However, emission probabilities are re-computed for every symbol (or set of symbols) encountered in the input sequences. If this computation takes time, it may be more efficient to precompute all possible emissions, and use lookup in the recursion. Whether this is more efficient depends on the number of outputs, the alphabet size, and the expected input sequence length.

Also, when transitions depend on previous emission (as with higher-order Markov states), they too are re-computed at every position, and the same remark applies.

9 Output classes

9.1 Dynamic programming tables

Dynamic programming (DP) tables for a HMM called `foo` have the type `fooDPTable`, and contain numbers of the type `fooReal` (which are either `doubles` or `bfloats`).

These DP tables are organized per clique. The position coordinates within a clique are relative to the lowest coordinate reachable for the states in the clique. For instance, if all outgoing transitions from the start state emit a symbol, there will be no reachable states at position 0 other than in the start clique. Similarly, the end state can only be reached at the end of the sequence. Furthermore, for memory efficiency each state is internally assigned a clique-specific identifier, so that these make up a consecutive range.

These optimizations make accessing DP table entries a bit cumbersome. For this reason, they provide two access functions:

```
double getProb(int iState, int iPos0, int iPos1, ...)
double getProb(const string iStateId, int iPos0, int iPos1, ...)
```

These take either a string or integer to identify the state, and integer position coordinates, and return the corresponding DP table value. If the position is out of range, the value 0 is returned, but no such sanity-checking on the state identifier is done.

To obtain the integer identifiers associated to transitions, states or emissions, use the getId member function of the DP class. (These identifiers are also valid on the Baum-Welch classes below.)

```
int getId(const string id);
```

It is often not necessary to name transitions, in order to define an HMM. However, you need to do so in order to refer to them via getId. If left unnamed, they are assigned dummy identifiers of the form id$number.

9.2 Baum Welch posterior counts

If the attribute `baumWelch=''yes''` is added to the `forward` (`backward`) class, it expects two additional tables as arguments to the call: first, a dynamic programming table of the `backward` (`forward`) algorithm, and second, a class that holds the Baum-Welch posterior counts of transitions and emissions.

For an HMM with identifier `Foo`, the class holding these counts is called `FooBaumWelch`. It contains a number of arrays which hold the posterior counts. These arrays are split into groups according to their dependence on emitted symbols, summarized by an ”order signature”.

This is best explained by giving an example. Suppose that an HMM has two output tapes. The transitions that have an first-order dependence on the first output tape, and no dependence on the second output tape, have order signature ”10”. For each group, the class defines three variables:

```
double transitionBaumWelchCount10[8][2];
int    transitionIdentifier10[2];
int    transitionDimension10 = 2;
```

The first array, `transBaumWelchCount10`, contains the posterior counts. Its first array index refers to the deepest (most ancient) symbol on the first output tape; in this case, the last symbol emitted. In the example above, the alphabet size is 8. The second index refers to the transition. The second array, `transIdentifier10`, gives the numerical identifiers of these two transitions; they correspond to the numerical identifiers used in the dynamic programming table and Path classes. The last variable, `transDimension10`, gives the number of transitions with the signature 10.

Symbols from the various alphabets are mapped to indices by sorting the alphabets in the natural way (i.e., alphabetically), and mapping them to consecutive numbers starting from 0.

The variables for the posterior emission counts are organized in the same way. Suppose there are emissions that emit on both tapes simultaneously, and have an additional (first order) dependence on the second output tape; then, the corresponding variables may appear like this:

```
  double emissionBaumWelchCount12[8][12][12][3];
  int    emissionIdentifier12[3];
  int    emissionDimension12 = 3;
```

Note that the order signature for these emissions is “12”: the emitted symbol is included. In this example, there are three emissions with this signature, and the alphabet size on the second output tape is 12. The first and third array indices refer to the emitted symbols; the second index refers to the first-order symbol on the first tape (the symbol emitted just prior to the current emission).

Two member functions, called `transitionIndex` and `emissionIndex`, translate the identifiers used in the DP table to the indices into the Baum-Welch counters. They are overloaded and will translate numerical indices as well as string identifiers:

```
  int tidx = baumWelch->transitionIndex(``transition1'');
  int eidx = baumWelch->transitionIndex(``emission1'');
  tcount = baumWelch->transitionBaumWelchCount10[ symbolA ][ tidx ];
  ecount = baumWelch->
    emissionBaumWelchCount12[ symbolA ][ symbolB1 ][ symbolB2 ][ eidx ];
```

Note that it is up to the user to make sure that the right array is accessed, with a signature corresponding to the transition or emission.

In the code above, `baumWelch` is an instance of the `FooBaumWelch` class that keeps track of the counts. This class may be re-used, and by default counts get accumulated. Calling `baumWelch.reset()` resets all counters to 0.

For efficiency, if only counts for emissions (or transitions) are used, you can specify this by the attribute values `baumWelch=''emissions''` and `baumWelch=''transitions''` respectively.

10 HMMoC warnings and errors

When the XML file contains errors, and cannot be parsed, the resulting error messages can be incomprehensible. It often helps to view the XML file in a browser to spot e.g. unclosed `<![CDATA[` tags.

When this hurdle is taken successfully, HMMoC may still complain in various ways. If a problem is encountered during interpretation of the XML file, HMMoC prints out part of the XML document tree up to the element causing the trouble, with each element uniquely labelled with an identifier of the type `id$bar`. Adding the `debug=''yes''` attribute to the top `<hml>` tag produces a copy of the input XML file with these identifiers added, which may help to find the error.

```
Warning - emission 'emitfoo' used from states with varying orders
  (with respect to <output> 'sequencebar'), using lowest.
```

The order of an emission (the numbers of symbols its probability depends on, not counting the emitted symbol), is determined by the order of the state the corresponding transition is coming from. Since the same emission may be assigned to several transitions, this may result in ambiguity. HMMoC chooses the lowest order among all relevant states, and gives this warning in case this behaviour was not intended.

```
Error: Could not find <identifier> for <output> foo1 (at depth bar)
```

If `bar`=0, you probably associated a `<code>` element to an `<emission>` element whose emission signature doesn’t match the `<code>` element’s. Even when emission distributions for two sequences are identical, you still need to specify two code blocks, bound to the two sequences.

If `bar`>0, the depth of the emission or transition to which the emission is associated and does not match the number of identifiers in the `<code>` element.

11 Compiler errors

```
error: cannot convert 'Algebra<BFloatMethods>[4]' to 'double*' in assignment
```

Probably you used both doubles and bfloats as real number types; this is not supported.

12 Examples

All of the examples can be built by typing `make`. They should all compile without any warning.

12.1 aligner

A simple probabilistic aligner. This program iterates 5 times through a Baum-Welch training procedure to optimize parameters, then computes a Viterbi alignment, and outputs this alignment with posterior column probabilities. This code gives an example how to use banding (although the banding implemented simply iterates over the entire DP table).

12.2 casino

The occasionally dishonest casino, the example HMM used in Biological Sequence Analysis by Durbin et al. The XML file defines two HMMs, with and without emissions. The emission-less HMM is used to sample from the HMM (which in HMMoC is always conditional on an emitted sequence). The resulting path is then adorned with emissions, and emission parameters are learned using Baum-Welch. The path is decoded using the Viterbi algorithm, and the true state, the decoded state, and the posterior probability is printed for the first 30 steps.

12.3 classifier

This is a piece of research code which performs latent class analysis on codon usage of human protein-coding genes, using an HMM. Two classes of genes (highly expressed, and with low expression) are used to intialize two of the 15 classes; randomized subset of the remaining genes are used to initialize the parameters of the other 13. After this, Baum-Welch training is used to optimize the classification based on codon usage bias, making sure that any signal from differential amino acid usage is taken out.

HMMoC issues a warning regarding the orders of the states; this can be ignored.

12.4 cpgisland

A simple HMM to identify CpG islands in nucleotide sequence, to demonstrate the use of neighbour-dependent (first-order) emission (and transition) probabilities, and how these can be learned using a Baum-Welch procedure.

HMMoC issues a warning regarding the orders of the states; this can be ignored.

12.5 hmmer

This example comprises a script that converts a HMMER (Sean Eddy, http://hmmer.janelia.org/) profile HMM into an XML file for HMMoC. The makefile builds programs for two profiles, TK and a zinc finger motif, and runs a homology search through a protein database using both HMMER and the HMMoC-generated algorithms. The HMMoC-generated program does not include any post-processing, and simply runs the Forward and the Viterbi algorithms, and reports the likelihoods.

13 Acknowledgements

A tool for automatically generating code for dynamic programming algorithms for biological sequence analysis, called Dynamite, was first implemented by Ewan Birney. Ian Holmes suggested to describe HMMs in particular in a structured language for various uses, including automatic code generation, and a proposal for this language, called Telegraph, was influential in the design of HMMoC. Alexei Drummond suggested to use XML and Java, which both turned out to be very good design choices.

Aaron Darling has kindly fixed several problems in HMMoC version 0.5 compiling under GCC version 4.0.2, and Microsoft Visual Studio.

The extended exponent reals uses a wrapper C++ class written by Ian Holmes, which was originally designed for a logspace real number type.

Several people have contributed by giving feedback, including Naila Mimouni, Ahmad Chughtai, Jon Churchill, Thomas Mailund, Rune Lyngsoe and David Dale.

HMMoC includes software developed by the JDOM project (http://www.jdom.org/), and software developed by the Apache Software Foundation (http://www.apache.org/).

14 License

This version of HMMoC is released under the GNU General Public License - see the documentation included for full details. Please contact me if you wish to use HMMoC under a different license.

If you use HMMoC in published work, please include a reference to this paper: Lunter, G.A., *HMMoC - a compiler for hidden Markov models.* **Bioinformatics**, 2007 Jul 10. `PMID: 17623703. doi:10.1093/bioinformatics/btm350`