Appendices

Appendix A

Software

A.1 Introduction

This appendix describes the software that was developed for the work described in this dissertation and is available for public use under the conditions of the Gnu Public License [GPL].

The tools fall into four groups, namely:

LogSpace A library of C++ classes for finding log-likelihoods, posterior probabilities and alignments using HMMs.

BayesPerl Perl modules and associated scripts for performing common manipulations (such as integration, transformation and marginalisation) on tables of log-likelihood data.

GFFTools A collection of perl scripts and C code for working with GFF format.

bigdp A program for joining together BLAST hits by dynamic programming.

Each of these groups will be described in turn.

A.2 LogSpace: C++ classes for working with HMMs

LogSpace is a set of classes for finding logs of likelihoods associated with single and pairwise hidden Markov models of any architecture, and for performing algorithms associated with these models.

In the following brief introduction to LogSpace, class names are shown in `typewriter` font.

The atoms of the LogSpace object model are:

- the `Parameter` class (P), representing a parameter of a probabilistic model;
- the `ParamVals` class ($\Theta : P \rightarrow \Re$), a mapping of `Parameter` objects to `doubles`), representing a point in parameter space); and

- the abstract `Function` class ($\log F : \Theta \to \Re$, $\frac{d \log F}{dP} : \Theta \to \Re$), representing the log-value of a mathematical function and the first derivatives of the log-value.

The most powerful and general of these classes is `Function`. This class can be overridden to describe any function on a parameter space that is everywhere non-negative and differentiable. Calling the methods of this object, supplying a `ParamVals` object, causes the value of the function (or its derivatives) to be evaluated at that point in parameter space. The emphasis in the API on logarithms of the function value (rather than the values themselves) encourages all calculations to be performed in log-space, where they are robust to scaling. A number of function calls facilitating common calculations in log-space (such as the log-sum-exp function $\Sigma(x, y) = \log[\exp x + \exp y]$) are also provided; these functions are zero-safe and often accelerated by means of lookup tables.

A range of useful `Function` subclasses are provided, some implementing basic mathematical functions such as the exponential and polynomial families, others allowing a `Function` to be defined in terms of other `Functions` (e.g. `FunctionSum` $F(\Theta) = G_1(\Theta) + G_2(\Theta)$, `FunctionProduct` $F(\Theta) = G_1(\Theta)G_2(\Theta)$, `ChainFunction` $F(\Theta) = G(\Phi)$ where $\Phi = \{H_i(\Theta)\}$), implementing basic results from calculus such as the chain rule and the product rule.

A pairwise hidden Markov `Model` M is represented in LogSpace as a set of links between the states of the model. Links can be added dynamically. Each link has associated with it a probabilistic substitution matrix, each entry of which is a `Function` on the parameter space. So this allows for quite general functional forms for the transition probabilities, corresponding to (for example) a time-dependent evolutionary model, or a model where several transitions are constrained to have the same (or related) probabilities. Links that correspond to insertions or deletions and hence emit only on one side of the model are also possible, and single-sequence models (like profile HMMs) are obtained as special cases of pairwise models.

The abstract `Envelope` class describes an iteration over a dynamic programming matrix, says which transitions are allowed and also describes how the matrix is to be laid out in memory. An `Envelope` requires a `Model` and a `SequencePair` (the latter is fairly self-descriptive); the default `Envelope`, `FullEnvelope`, just iterates over all the cells of the matrix. A `SparseEnvelope` class implementing the sparse envelopes described in Chapter 2 is also provided.

The abstract `DPMBase` is the base class for all dynamic programming algorithms including Viterbi, sum-over-paths and optimal-accuracy, using integer precision or double precision. `DPMBase` takes an `Envelope` in its constructor. The `FBM` (forward-backward matrix) provides posterior probabilities for every point in the matrix.

The details of the dynamic programming are invisible if the `Likelihood` class $L(\Theta) = \Pr[S|M,\Theta]$ is used. `Likelihood` is a subclass of `Function` that takes an `Envelope` in its constructor. Recall that an `Envelope` describes a pair of sequences S and a model M; so, a `Likelihood` object calculates the value and derivatives of the log-likelihood score of a pair of sequences given a model for any particular parameterisation of that model.

The derivatives of the likelihood can be fed straight into a discriminative classifier [JH98] or used to train the model [DEKM98]. A number of classes such as `JointLikelihood`

There are many other classes and methods in the LogSpace libraries, including alignments, time-dependent models and optimisation algorithms, that are not described here. It is hoped that the above short introduction is sufficient to give an idea of the range of these libraries.

A.2.1 Posterior probabilities for profile HMMs

The LogSpace classes were used to develop a posterior probability framework for HMMER2.0 profiles, in parallel with the code described in Chapter 4. This parallel implementation (and a perl program `hmm2mf.pl` for converting between

HMMER2.0 and LogSpace model file formats) is included in the LogSpace distribution.

A.2.2 Availability

The LogSpace source code can be found at the following URL:

`http://www.sanger.ac.uk/Users/ihh/LogSpace/`

A.3 BayesPerl: Perl modules and scripts for working with tables of log-likelihood data

The LogSpace code described above can be used to sample the log-likelihood of sequences over the parameter space of a hidden Markov model. These operations are typically compute-intensive and it is convenient to save the log-likelihood tables in intermediate data files before performing further numerical manipulations. BayesPerl is a set of perl modules and scripts for manipulating these data files and the tables they contain.

The format of the data files operated on by these programs is as follows. Each line of the file represents a single entry in the table, and thus a single data point. Each line has $N + 1$ numeric fields separated by whitespace, where N is the dimensionality of the parameter space. The first N fields are the parameter values (the co-ordinates in the parameter space) and the $(N + 1)$'th field is the value of the log-likelihood at that point.

The main component of the BayesPerl modules is the `LogLikeTable.pm` package, which contains the following main methods:

`new` Creates a new table of log-likelihood values.

`clone` Clones an existing table.

`newFromHandle` Reads a table from a file handle.

`newFromFile` Reads a table from a file.

`newFromArray` Creates a table from an array of values.

`newFromFunction` Creates a table from a perl function reference, which is evaluated at every point.

`combine` Combines two tables to give a new table with higher dimensionality.

`combineEntriesRule` Sets the rule for combining two log-likelihoods at the same point (by default, the log-likelihoods are summed).

`paramRange` Returns the range of values taken by a particular parameter in a table.

`findMode` Finds the mode of a table (the maximum-likelihood parameters).

`absorb` Multiplies (or adds, depending on the combineEntriesRule) a table by another table; useful for e.g. priors.

`marginalise` Marginalises parameters of a table by integrating them out; the result is a table of lower dimensionality.

`integrate` Finds the log-integral of the likelihood across the entire space; equivalent to marginalising all parameters.

`normalise` Subtracts the log-integral from all the log-likelihoods; turns a likelihood distribution into a posterior distribution.

`interpolate` Uses straight-line nearest-neighbour interpolation to find the log-likelihood anywhere in the parameter space.

`transform` Projects the table onto a new co-ordinate system.

`print` Displays a table (or dumps it to a file).

Much of the functionality of the `LogLikeTable.pm` package is duplicated by the (slightly more efficient) `LogLikeGrid.pm` package, which assumes that its data points lie on an irregular grid.

There is also a package `LogSpace.pm` that supplies some basic constants and functions compatible with the LogSpace C++ classes described above.

Some of the methods are quite slow on large tables. Much of this slowness is due to the Perl object-orientation layer, so a set of procedural scripts that mirror some of the `LogLikeTable.pm` methods (but faster) has also been developed.

A.3.1 Availability

The BayesPerl release can be found at the following URL:

`http://www.sanger.ac.uk/Users/ihh/Bayes/`

A.4 GFFTools: Perl scripts for processing GFF files

GFF (Gene-Finding Format) is a one-line-per-record format for marking up genomic sequence. It was originally designed as a common format for sharing information between gene-finding sensors such as splice site and coding sequence predictors, but its uses go beyond gene-finding: a GFF file is a convenient way of representing a set of many kinds of feature. The chief drawback of GFF - its simplicity - could also be said to be its chief strength, since a wide range of perl scripts and modules for operating on GFF sets has been developed in a short space of time ($\sim$ 1 year). The latest version of `ACeDB` has the facility to export all *C.elegans* genome annotation in GFF format.

A single record in a GFF data set is a line with 9 tab-separated fields. These fields are:

- Sequence name
- Source (the program that generated the data)
- Feature name
- Sequence startpoint

- Sequence endpoint
- Score of feature
- Strand (can be "+", "-" or ".")
- Frame (for frame-sensitive features such as introns)
- Group (a catch-all miscellaneous field)

Fields can contain spaces (but not tabs or newlines). A proposed extension to GFF is the "GFF pair", wherein first three whitespace-delimited words in the "group" field represent the sequence name, startpoint and endpoint of a homologous sequence. Other than this, there is no consensus on syntax for fields, though a "tag=value" approach to including extra information in the "group" field may be favoured.

Soon after the GFF format was agreed, Tim Hubbard at the Sanger Centre developed a set of useful GFF-related Perl modules [GFF]. Many of the scripts described here duplicate functionality included in these modules, yet they were developed later. Why is this? The reasons, simply, were speed and space. Perl's object-orientation slows computation time considerably and greater interactivity was required than the GFF Perl modules allowed; as for space, reading entire chromosome-sized GFF files into memory is often impractical.

A GFF record is a special case of a class of objects that may be described as annotated NSE's. A basic NSE consists of a *(name,start,end)* tuple. Many of the algorithms described here would work without modification on other NSE-like formats.

The following list contains brief descriptions all the GFF programs developed, together with several programs for working with EXBLX, an alternative format for representing NSE pairs that is used by the `MSPcrunch` program [SD94].

- GFF sorting/filtering programs:

`gffsort.pl` Sorts a stream of GFF records by sequence name and start-point.

`gfffilter.pl` Filters a GFF stream according to a user-specifiable test condition.

`gffmerge.pl` Merges two or more pre-sorted GFF streams.

- Programs to convert between GFF co-ordinate systems and manipulate GFF-described sequences:

  `gfftransform.pl` Converts from one GFF co-ordinate system to another (e.g. from clones to chromosomes). Works with GFF pairs.

  `gff2seq.pl` Given chromosome co-ordinates, a clone database and a physical map co-ordinate file, returns the specified section of chromosomal sequence.

  `gffmask.pl` Masks GFF-specified sections of a FASTA sequence database with N's.

  `GFFTransform.pm` Perl module to convert between GFF co-ordinate systems. Used by the `gfftransform.pl` and `exblxtransform.pl` scripts.

  `SequenceMap.pm` Perl module to access a clone database given a map file. Used by the `gff2seq.pl` script.

  `FileIndex.pm` Perl module to build a quick lookup index for flatfiles. Used by the `SequenceMap.pm` module, and others.

- Programs to find intersections and connections between GFF data sets:

  `gffintersect.pl` Efficiently finds the intersection (or exclusion) between two (sorted) GFF streams. Definition of "intersection" allows for near-neighbours and minimum-overlap.

  `intersectlookup.pl` Used with `gffintersect.pl` to do inverse lookups and other manipulations on the result of an intersection test. Can

be useful for self-comparisons, e.g. to find the highest-scoring non-overlapping subset of a GFF file.

`gffhitcount` C++ program that counts the number of times each base in a GFF file is hit, and outputs the results as a GFF tiling. Uses a lot of memory. Works with GFF pairs.

`exblxgffintersect.pl` Similar to `gffintersect.pl`, but finds the intersection between a GFF set and an EXBLX file (similar to a list of GFF pairs). Useful for e.g. filtering out all hits between genes from an all-vs-all comparison of genomic DNA.

`gffdp.pl` Parses GFF data using a finite-state automaton with a push-down stack. The FSA is entirely user-specifiable and may include Perl expressions that are evaluated dynamically for each GFF record. This program is described in greater detail below.

- Miscelleneous GFF-related programs:

  `blasttransform.pl` BLASTs a clone database against itself, then transforms, sorts and merges the results into chromosome co-ordinates according to a physical (sequence) map.

  `cfilter.pl` Uses a sliding-window oligomer-counting method to find GFF co-ordinates for low-entropy regions in a sequence database.

- Miscellaneous EXBLX-related programs:

  `exblxsym.pl` Symmetrises an EXBLX file (ensures that for every pair $A \leftrightarrow B$ there is a single corresponding pair $B \leftrightarrow A$).

  `exblxasym.pl` Asymmetrises an EXBLX stream (filters through only those pairs $A \leftrightarrow B$ for which $B > A$).

  `exblxcluster.pl` Builds single-linkage clusters from an EXBLX stream, optimising for cluster size.

`exblxfastcluster.pl` Builds clusters from an EXBLX stream using a fast incremental heuristic.

`seqcluster.pl` Builds single-linkage clusters from an EXBLX stream, optimising for cluster size and ignoring sequence start and endpoint.

`exblxindex.pl` Builds a quick lookup index for an EXBLX file.

`exblxsingles.pl` Filters through only non-overlapping entries from an EXBLX stream.

`exblxsort.pl` Sorts an EXBLX stream.

`exblxtidy.pl` Tidies up an EXBLX stream (joins overlapping matches, prunes out BLAST errors, etc.).

`exblxtransform.pl` Converts from one EXBLX co-ordinate system to another (e.g. from clones to chromosomes).

- Format conversion programs:

`exblx2gff.pl` From EXBLX to GFF pairs.

`gff2exblx.pl` From GFF pairs to EXBLX.

`scan2gff.pl` From scan (GCG) to GFF.

`tandem2gff.pl` From tandem (GCG) to GFF.

`spcwise2gff.pl` From spcwise (GeneWise) to GFF.

`cluster2gff.pl` From single-line lists of NSE clusters to GFF.

`hmm2gff.pl` From HMMER1.7 to GFF.

`hmmsearch2gff.pl` From HMMER2.0 to GFF.

The most flexible of the GFF programs is `gffdp.pl`. This assembles GFF segments using dynamic programming. The model architecture for the dynamic programming is specified in a text file using a syntax that allows perl expressions to be evaluated on-the-fly. The finite state automaton has a LIFO stack that

allows nested structures (such as inverted repeats) to be handled in a sensible way. All alternative states of the stack are maintained in the dynamic programming matrix. All the arcs in the finite state machine can emit variable-length sequences, so the program can emulate a generalised HMM [Hau98]. The dynamic evaluation of perl expressions during the dynamic programming allows for calculation of complex scoring schemes, such as logarithmic gap penalties. Transitions along an arc may be required to overlap or align with a GFF segment exactly, loosely or not at all. GFF pairs are also provided for; the co-ordinates of the paired segment can accessed and it can even be inserted into the upcoming GFF buffer.

Figure A.1 shows a simplified version of one of the `gffdp.pl` model files used for the repeat-mediated duplications search described in Section 5.4 of Chapter 5. The `gffdp.pl` program was also used in Chapter 7 to find invrep sequences flanking predicted transposase genes. There are many other uses for the `gffdp.pl` program; one obvious use is genefinding - assembling exon predictions from a variety of sensors. This is the task that GFF was originally designed for. A `gffdp.pl` model file for genefinding is available from the GFF website.

A.4.1 Availability

Further information and resources relating to the `gffdp.pl` program and the GFF format may be obtained from the GFF website, whose URL is:

`http://www.sanger.ac.uk/Software/GFF/`

A.5 bigdp: A program for assembling BLAST hits by dynamic programming

`bigdp` is a program that joins together BLAST hits with an affine gap penalty by doing linear space divide-and-conquer dynamic programming [DEKM98]. The program does not itself examine the sequence to which the BLAST HSPs refer

```
name { rep1rep2 } flushlen { 30000 }

link { from { start } to { repeat1 } maxlen { 3000 }
       endfilter { $gffsource eq "repeat"
                   && $linkend == $gffend + 1 }
       startfilter { $linkstart == $gffstart }
       push { $gfffeature } push { $gffstrand }
}

link { from { repeat1 } to { match1 } maxlen { 5000 }
       endfilter { $gffsource eq "match"
                   && $gffstrand eq '+'
                   && $linkend == $gffend + 1
                   && $gffend-$gffstart > 20 }
       startfilter { $linkstart <= $gffstart }
       insertgff { }
       popfilter { $temp_repstrand = $_; 1 }
       popfilter { $temp_repname = $_; 1 }
       push { $insertgffid } push { $temp_repname } push { $temp_repstrand }
}

link { from { match1 } to { repeat2 } maxlen { 4000 }
       endfilter { $gffsource eq "repeat"
                   && $linkend == $gffend + 1 }
       startfilter { $linkstart <= $gffstart }
       popfilter { $_ eq $gffstrand }
       popfilter { $_ eq $gfffeature }
}

link { from { repeat2 } to { end } maxlen { 5000 }
       endfilter { $gffsource eq "match"
                   && $gffstrand eq '+'
                   && $linkend == $gffend + 1 }
       startfilter { $linkstart <= $gffstart }
       popfilter { }
       display { print "Found a hit ending at $gffend\n" }
}

link { from { end } to { start } }
```

Figure A.1: Model file for the *repeat* → *match* → *repeat* → *match* pattern. Whether a GFF line represents a *match* or a *repeat* is indicated in the `endfilter` field. Matches are parsed as GFF pairs.

but merely finds optimal-scoring connections between the HSPs given their coordinates. `bigdp` was designed to cope with large amounts of data, such as might be generated by BLASTing whole chromosomes against one another. It is essentially similar to the Smith-Waterman algorithm [SW81], except that all *match* $\rightarrow$ *match* transitions must correspond to BLAST hits and consequently many cells in the dynamic programming matrix can effectively be dispensed with.

The `bigdp` program returns all (non-overlapping) alignments above a certain score threshold by a method of excluding previously-used segments; the closest relative of this method is the procedure described by Waterman and Eggert [WE87]. Rather than covering the whole dynamic programming matrix, the algorithm stops if an alignment above the score threshold has been found and there have been no better alignments after a distance δ has been covered. Additionally, rather than start from the beginning of the dynamic programming matrix after an optimal alignment has been found, the algorithm starts a distance ϵ left of the startpoint of the highest-scoring alignment found on the previous run. The startpoint information is propagated using a "shadow matrix" technique [BD97]. Choosing δ to be much greater than the maximum gap length and ϵ to be much greater than the maximum low-scoring subalignment length reduces the expected running time to $O(MN^2)$, where M and N are the sequence lengths (since there are $\sim MN$ expected alignments and each takes time $\sim N$ to find), rather than $O(M^2N^2)$ (the expected running time if the entire matrix were to be scanned for each alignment) without missing any hits. Alignments may be missed if they overlap high-scoring alignments and contain subsegments longer than ϵ that score lower than the alignment detection threshold.

Other programs to extend BLAST to give gapped alignments include gapped BLAST [AG96, AMS+97] and `MSPcrunch` [SD94]. There are other ways of searching large sequences quickly for multiple matches, e.g. [VBA+98]. All

these programs use heuristic methods and not full dynamic programming; in some cases the heuristics may be more sensitive. The statistical bias induced by the heuristic methods on the observed data is likely to be different than the dynamic programming.

A.5.1 Availability

The `bigdp` program is available from the following URL:

`http://www.sanger.ac.uk/Users/ihh/bigdp/`