Chapter 2

GAZE

2.1 Introduction

Many of the gene identification techniques discussed in the previous chapter have two things in common: (i) signal and content measures are used to detect components and regions belonging to genes; (ii) these are assembled into a complete gene structure prediction for the sequence that is optimal with respect to some discrimination measure, often by dynamic programming. For the first of these steps, choices must be made as to the nature of the specific signal and content measures used, for example whether to use simple weight matrices or weight array models for splice site detection or whether to use pentamer or hexamer frequencies for coding-region scoring. For the second of these steps, a choice must be made as to the *model* of gene structure over which the assembly is to be performed, i.e. how the gene components relate to each other and fit together into complete genes. It is usually the case that choices in both of these steps are made to produce a system that has the the highest possible accuracy.

As we find out more about the biological processes of transcription, RNA processing and translation, we might want to adjust or extend these gene prediction methods to reflect our increased understanding. For example, promoter identification methods, although advancing, are still not sufficiently accurate be able to

identify the 5' ends of genes with any confidence [40]. But if an improved method becomes available, then there would be much to gain from incorporating it into a gene prediction system.

Furthermore, programs that make use of similarity information (for example alignments of cDNAs to the genomic sequence) give more accurate results than *ab initio* methods (see chapter 1) where such evidence exists. If *ab initio* methods can be extended to make use of similarity evidence where it exists, we might hope to improve gene prediction accuracy without compromising the ability to predict novel genes where it does not.

The incorporation of new information into many existing gene prediction systems may not be straightforward due to rigidities inherent in their implementation. At best, knowledge of the underlying software is required, and even given this, it is often necessary to produce a custom version of the software that is designed to work *only* when the new data is present. This is true for at least four of the gene prediction systems discussed earlier [69] [92] [96] [119].

My work has focused on the development of a framework for gene prediction that decouples the assembly of signal and content data into gene structure predictions from the generation of this data itself. I have implemented a program called **GAZE** for the assembly of **features** (corresponding to signal sensors) and **segments** (corresponding to content sensors) that is tied neither to any specific signal or content detection techniques nor any assumed model of gene structure. Both of these elements are external to the system. The goal has been to provide a method for the rapid and seamless integration of new/improved methods and data into the gene prediction process.

The main novelty of GAZE is that it does not work directly with genomic DNA sequence. It instead predicts gene structures from input files containing the results of various signal and content sensors with associated scores, typically log probability ratios. These files are assumed to be in the General Feature Format [GFF; http://www.sanger.ac.uk/Software/GFF], a format which has rapidly become

a widely used standard for the exchange of gene prediction information. The assembly of this information is directed by a *configuration* file (in the eXtensible Mark-up language, XML [http://www.w3.org/XML]), which affords the user control over the validation and scoring of candidate gene structures.

This chapter describes the details of the GAZE system. I start by outlining the main approach taken, explaining how gene structures might be inferred from lists of features and segments. Next, I describe the details of the GAZE configuration file, and how it affords control over both the *validation* and *scoring* of candidate gene structures. Two technical sections follow, detailing firstly the GAZE scoring function and secondly how it is used to obtain a probability distribution over gene structures and why this is useful. I then discuss some of the innovations implemented to improve the efficiency of the algorithms, including a novel search-space pruning technique. To end, I relate the GAZE approach to other gene prediction programs and methods. The following chapters show examples of the use of GAZE for implementing gene finders, and further theory and technical issues relating to estimating parameters for GAZE.

2.2 From features and segments to gene structures

The primary input to GAZE is a file containing the results of arbitrary signal and content sensors. Each comes with a *position* on the sequence (i.e. a start and end) and a *score*. From this file, collections of *features* (from the signal sensors) and *segments* (from the content sensors) are constructed.

The GAZE approach is that a gene structure can be described by an ordered subset of specific features taken from the given collection. For example, for a sequence of 1400 nucleotides, the following describes a structure with two genes, consisting of two exons and a single exon respectively (unless otherwise stated, I will use the term "exon" to mean the protein-coding part only; this definition is inconsistent with the classical definition used by molecular biologists, but is both convenient and consistent with other literature on gene finding):

Feature	Start	End
BEGIN	1	1
start	201	203
donor	305	306
acceptor	900	901
stop	1040	1042
start	1101	1103
stop	1218	1220
END	1400	1400

whereas a list describing a structure for which the protein-coding part is found on a single exon on the reverse strand might be:

Feature	Start	End
BEGIN	1	1
stop_rev	1151	1153
start_rev	1208	1210
END	1400	1400

Gene structures that consist of no genes can also be described by an effectively empty list which includes only the features marking the beginning and end of the sequence:

Feature	Start	End
BEGIN	1	1
END	1400	1400

Given a candidate set of features, GAZE predicts genes by obtaining the ordered subset (list) of features that according to its model is most likely to correspond to the correct gene structure. It does this by assigning a score to each list and and defining the most likely gene structure to be the list with the highest score. As explained in more detail in section 2.4, the score assigned to each candidate structure is a

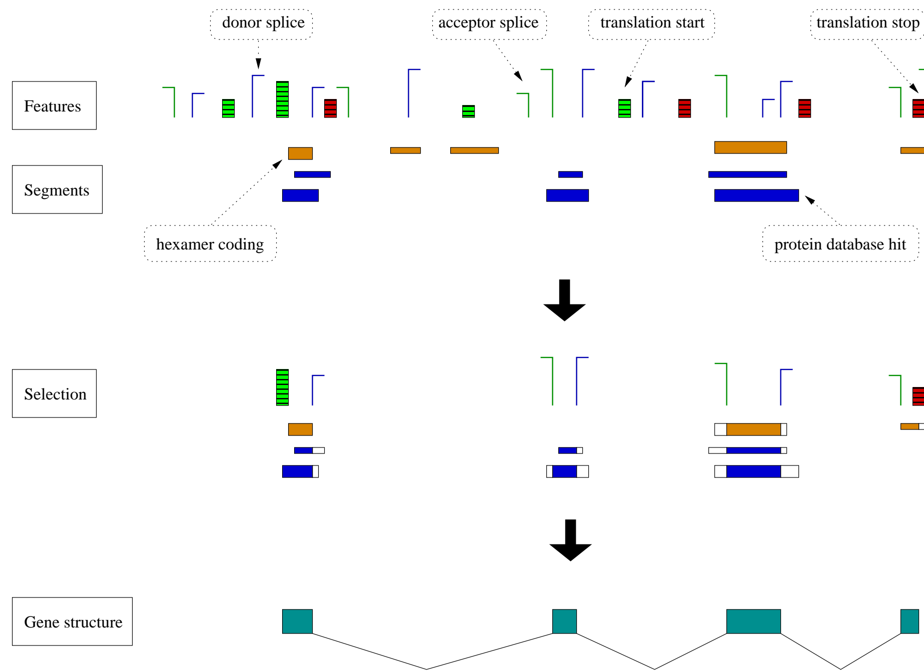

Figure 2.1: How GAZE predicts genes. The input is a list of ordered features and segments, drawn here in different sizes to reflect their scores. A candidate gene structure corresponds to a selection of these features, and is assigned a composite score based on (i) the scores of the *features* themselves and (ii) the scores of specific *segments* that lie in the appropriate regions between adjacent pairs of the features (here, both types of segment represent regions with a high likelihood of being protein-coding, so therefore contribute only towards the scores for protein coding exons). The feature selection with the highest composite score is output as the most likely gene structure.

function of the given scores of the features which make it up, and the segments lying in the regions defined by pairs of adjacent features in the structure (see figure 2.1).

Not all lists of features correspond to sensible gene structures. For example, any list which contains a donor splice site immediately followed by a stop codon cannot possibly be correct, because we know that in real gene structures, a donor splice site is necessarily followed by an acceptor splice site. Furthermore not all segments should contribute towards the score for all regions. For example, a segment corresponding to a match to a protein database should not contribute towards the score for a region between a donor splice and an acceptor splice; that is, evidence for

protein-coding regions should not be used to support candidate non-coding introns. For many gene prediction systems, such rules and constraints upon gene structure are encoded into the logic of the program itself. In GAZE however, the *gene structure model* is external to the system and supplied by the user in a *configuration* file.

2.3 Elements of a GAZE configuration

Specific examples of the GAZE configuration language are shown in the next chapter, but here I describe the elements of the most important aspect of a configuration, namely the gene structure model. The model has two main purposes: firstly to define which lists of features are *valid* gene structures, and secondly to define how valid structures are to be scored, with reference to both the segments and a set of *length penalty* functions.

2.3.1 Defining the validity of candidate gene structures

The model is initially constructed by giving a set of rules for each type of *target* feature, defining which types of *source* feature can immediately precede them in a valid structure. In the first gene structure above, a "stop" target feature can be immediately preceded upstream by "start" or "acceptor" source features, and the model would therefore need to to contain rules for *start* → *stop* and *acceptor* → *stop* (as well as others) to allow this gene structure to be recognised as valid.

The *source* → *target* rules themselves can be qualified with constraints that candidate (source, target) pairs of features should satisfy. There are four types of constraint:

Distance constraints, indicating that there should be no more than a maximum and no fewer than a minimum number of bases between the source and target.

Phase constraints, indicating that the source and target should occur 0, 1, or 2 nucleotides (modulo 3) apart.

Interruption constraints, indicating that a (source, target) pair is invalid if the region defined by the pair is interrupted by the occurrence of the specified feature at the specified distance (modulo 3) from either the source or target. These constraints are used to invalidate potential coding exons that are interrupted by an in-frame stop codon.

DNA constraints, indicating that a (source,target) pair is invalid if the DNA located at the source and/or target has a specified sequence. These constraints are used to invalidate gene structures that would give rise to in-frame stop codons across exon-exon boundaries in the spliced messenger RNA.

The space of valid gene structures can be further refined by denoting specific features in the input set as **selected** or **de-selected**. Any candidate gene structure that does not include *all* of the features flagged as selected is considered invalid. Likewise, any candidate gene structure that includes *any* of the features flagged as de-selected is also considered invalid.

Technical caveats

For practical purposes, I define the "distance" between a pair features as the length, in nucleotides, of the region between them. For example, the region between bases 567 and 890 has length $890 - 567 + 1 = 324$ nucleotides. Conceptually then, the distance between a source feature and a target feature is the location of the target minus the location of the source (plus one). A technical problem with this is that firstly features often do not have single nucleotide positions, and secondly, some features are considered part of the region they delimit, and some not. Furthermore, this can depend on whether a feature is acting as a source or a target (marking respectively the left or right boundary of the region). For example, a feature representing a candidate translation start site not only covers three nucleotides (ATG), and therefore does not have a position on the sequence that be described in a single number, but needs to be treated differently depending on whether it is a source or

a target. When acting as a source, it marks the start of a candidate protein-coding exon, and should itself be considered part of this region, whereas when it acts as a target, it marks the end of a non-protein-coding region, and should not be itself considered part of the region.

To address this problem, GAZE requires the definition, for each feature type, of a “source offset” and a “target offset”. The source offset is an integer number that is *added* to the *start* position of the feature when it is treated as a source, and the target offset is an integer *subtracted* from the *end* location of the feature when it acts as a target. The translation start feature above will commonly be defined as having a start offset of 0 and an end offset of 3. So, for an instance of this feature-type occurring at 567-569 (say), the region beginning with the feature when it acts as a source starts at $567 + 0$, whereas the region ending with the feature when it acts as a target will end at $569 - 3 = 566$, i.e. one nucleotide before the given start of the feature, which has the desired effect.

The imprecise notion of feature location needs to be addressed when sorting features by their position on the sequence. The concept of a total order over all candidate features is important not only for the dynamic programming recursions described in section 2.4, but also because such an ordering is assumed when defining the space of valid gene structures; interruption constraints are violated when a designated feature C occurs between feature A and B, i.e. when the relative order that the features occur in the list is A, C, B.

Because features have both a start and an end point in sequence co-ordinates, the obvious approach is to construct the total order by sorting the features first by start point, and then by end point, and finally (in the case of 2 features having the same start and end location) by feature type. The first part of figure 2.2 shows why this sorting strategy can give incorrect results.

The problem is caused by features that overlap, in particular features involved in interruption constraints (in most uses of GAZE, stop codons). If an acceptor splice site occurs in the middle of a candidate stop codon, then the protein-coding region

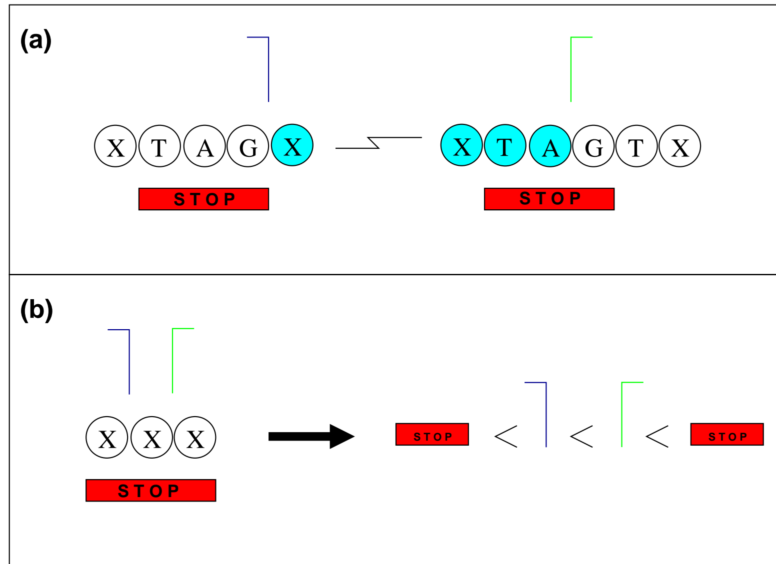

Figure 2.2: The difficulty in defining a total order over the positions of features in the sequence. (a) The candidate exon (indicated by the light-blue residues) is valid as neither candidate stop codon (red boxes) occurs completely within it. However, when the features are ordered naturally by their start points, the second stop occurs before the splice donor at the 3' end of the exon (green hook), giving it the appearance of invalidating the region. Similar problems arise when the features are ordered by their end-point. (b) In this case, we would like to engineer an ordering strategy whereby the stop codon occurs upstream of the candidate donor splice site (blue hook) and downstream of the candidate splice acceptor, with the two splices in their correct orientation. No such ordering exists. Although this specific situation is implausible (the candidate internal exon is 1 nucleotide long) such problems need to be considered because they become more likely as the range and diversity of the gene prediction data being used increases.

upstream of the splice acceptor is not invalidated by it. We would therefore like to place the stop codon *before* the splice acceptor in the list. If the same situation occurs with a stop codon and a splice donor at the 3' end of en exon, we would like to place the stop *after* the splice donor, for the same reason.

The aim therefore is to define a sorting strategy that takes this information into account when deciding upon the relative order of a pair of features. Unfortunately, it is also possible to imagine a situation where such a strategy might break down. The second part of figure 2.2 shows that a circularity in the ordering function arises when candidate splice acceptor, splice donor and stop codon appear in close proximity.

Such a circularity can lead to non-determinism in feature sorting, whereby the final ordering depends not only upon the condition dictating the relative order of a pair of features, but also on the original order of the features. For this reason, it was decided to abandon the idea of trying to determine definitive total order over a list of features. Features are therefore sorted in a natural, deterministic way, first by start-point, then by end-point, and then by type. The problem of the interruption constraints is dealt with in the dynamic programming algorithms themselves. Specifically, an apparent violation of an interruption constraint is checked to see if the interrupting feature really does lie within the region of interest. The primary disadvantage of this is that it makes the code more complicated and therefore more difficult to maintain.

2.3.2 Defining the scoring of valid gene structures

The overall score of a gene structure is the sum of of scores for the individual *features* (as given in the GFF file) and for the *regions* between each adjacent pair of features in the structure (this is defined more precisely in section 2.4). The region scores can be tailored precisely for each (source,target) pair, by specifying in the $source \rightarrow target$ rule two elements: the name of a **length penalty** function, and a list of **Segment Qualifiers**.

Length Penalty Function

Length penalty functions reflect the fact that it is often more likely for a given source and target to appear at certain distances apart than others. Each $source \rightarrow target$ rule can be qualified with the name of a length penalty function, mapping distances to a floating point number that will be subtracted from the score for the region between the source and target (see section 2.4). The functions themselves are defined by simply listing (distance, penalty) pairs, with linear interpolation used to derive penalties for distances not given. For distances greater than the largest given, the final two given points are extrapolated. It is straightforward to define penalty functions that are eventually uniform by making the penalties for the last two given

distances equal.

Segment Qualifiers

Segment qualifiers control which segments contribute to the score for the region between the source and target, and under what conditions. Since the region between an acceptor splice site and a donor splice site (for example) defines a candidate protein-coding region, both a *likely_coding* segment (indicative of a region of high protein-coding potential by some statistical measure), and a *protein_match* segment (corresponding to a region of strong similarity to an entry in a database of protein sequences) lying in this region provide evidence that the region *is* protein-coding, and can therefore be used to increase the score of the region.

Each *source* → *target* rule can contain several segment qualifiers. A separate score is calculated for each qualifier, and these are added to get a total segment score for the region (see section 2.4). The qualifiers themselves can contain constraints to restrict which segments should be considered *relevant* in the calculation of the score for that qualifier, namely:

Type constraints, indicating that only segments of the designated type should be considered. This is a compulsory constraint.

Phase constraints, indicating that that the starts of relevant segments should occur at 0, 1, or 2 nucleotides (modulo 3) away from the source or target. This gives the facility to consider only segments that are in-frame with respect to the source/target.

match constraints, indicating that the start and/or end of the segment must lie at the same position as the source and/or target. This gives the facility to consider only segments that fit the region precisely, allowing the use of the output of of programs that identify potential exact intron, exons, or other gene regions.

completeness constraints, indicating (when specified) that the segment must lie completely within the region to be considered relevant.

2.4 Prediction with a GAZE gene structure model

GAZE predicts genes by choosing from a large set of candidate features, the ordered subset (list) that (a) is consistent with the gene structure model (i.e. does not violate any constraints), and (b) has a score at least as high as all other consistent gene structures.

2.4.1 The GAZE scoring function

Given firstly a list $\phi = \phi_1, \phi_2, \ldots, \phi_n$, of features ordered by sequence position defining a *valid* gene structure according to a GAZE model, their types $t(\phi_i)$, their locations[1] on the sequence $l(\phi_i)$, and their given scores $g(\phi_i)$, then the score of ϕ, $E(\phi)$ is calculated as:

$$E(\phi) = \sum_{i=0}^{n} Reg_{t(\phi_i) \rightarrow t(\phi_{i+1})}(l(\phi_i), l(\phi_{i+1})) + g(\phi_{i+1}) \tag{2.1}$$

The features ϕ_0 ("BEGIN")and ϕ_{n+1} ("END") are not supplied by the user but are present in all gene structures and act to mark the beginning and end (respectively) of the sequence. Their "given" scores are always 0. $Reg_{src \rightarrow tgt}(x, y)$ is the region score for the interval [x,y], where the interval is bordered on the left and right by features of type *src* and *tgt* respectively:

$$Reg_{src \rightarrow tgt}(x, y) = Seg_{src \rightarrow tgt}(x, y) - Len_{src \rightarrow tgt}(y - x + 1) \tag{2.2}$$

$Len_{src \rightarrow tgt}(x)$ is the distance penalty function specified for the rule $src \rightarrow tgt$. Each function maps a distance (in base pairs) to a penalty score. If no penalty

[1] location is a function of the start co-ordinate and start offset when the feature acts as a source, and end co-ordinate and end offset when the feature acts as a target, as explained in section 2.3.1.

function is specified for the $src \rightarrow tgt$ rule, a default, mapping all distances to zero, is assumed.

$Seg_{src \rightarrow tgt}(x, y)$, the segment score, is sum of separate scores for each segment qualifier in the $src \rightarrow tgt$ rule. If $Q_{src \rightarrow tgt}$ is the list of segment qualifiers appearing in the $src \rightarrow tgt$ rule, and ψ^q is the relevant subset of segments for segment qualifier q, i.e. those that satisfy the type, phase, match and completeness constraints defined in q, then

$$Seg_{src \rightarrow tgt}(x, y) = \sum_{q \in Q_{src \rightarrow tgt}} Seg^Q(\psi^q, x, y) \tag{2.3}$$

If the given scores of each of the segments in a list ψ are denoted by $g(\phi_i)$, then $Seg^Q(\psi, x, y)$ itself is calculated in one of two ways:

$$Seg^Q(\psi, x, y) = \max_{\psi_i \in \psi} \frac{|\psi_i \cap x \ldots y|}{|\psi_i|} g(\psi_i) \tag{2.4}$$

$$Seg^Q(\psi, x, y) = \sum_{r=x}^{y} \max_{\psi_i \in \psi} \frac{|\psi_i \cap r \ldots r|}{|\psi_i|} g(\psi_i) \tag{2.5}$$

where $|\psi_i|$ is the length of segment ψ_i and $|\psi_i \cap x \ldots y|$ is the number of bases of overlap between segment ψ_i and the region $x \ldots y$. The first function corresponds to a "maximum single" approach, using the single segment with the highest score (after it has been scaled according to the proportion of segment lying in the region being considered). The second function corresponds to a "projected per-base" approach, summing the scores for the individual bases lying in the region, each of which is calculated by taking the maximal per-base score of all segments covering that base.

It is up to the user to decide which of these two functions should be used to score segments of each type. The default is the second, but some segments, by their construction, may give better results if the first function is used to score them.

2.4.2 Obtaining the highest scoring valid gene structure

A straightforward way for GAZE to obtain the highest scoring model-consistent gene structure would be to enumerate all gene structures, calculate the score for each one (a simple linear combination of terms, as shown above), and retain the structure with the highest score. This direct approach is not practicable since the number of possible gene structures grows exponentially with sequence length [116]. However, dynamic programming can be used to explore the search-space efficiently, by constructing partial solutions in a left-to-right manner, at each stage in effect discarding partial gene structures that cannot possible be prefixes of the optimal structure.

The GAZE algorithm for obtaining the highest scoring gene structure is in many ways similar to others described for the same problem ([89], [49], [109], [72], [116]). It relies upon an ordering of all candidate features by their position on the sequence. Taking now $\phi_1, \phi_2, \ldots, \phi_n$ to be a complete set of candidate features ordered by position on the sequence (with again ϕ_0 being "BEGIN" and ϕ_{n+1} being "END"), the maximal-scoring valid gene structure is obtained by the following dynamic-programming recurrence:

$$v(0) = 0 \tag{2.6}$$

$$v(i) = \max_{j<i}[v(j) + Reg_{t(\phi_j)\rightarrow t(\phi_i)}(l(\phi_j), l(\phi_i)) + g(\phi_i)] \tag{2.7}$$

The score of the optimal gene structure is $v(n+1)$, and the gene structure itself can be obtained with a *traceback* procedure. This involves maintaining a separate vector d, where $d(i)$ contains the index j that was found to be the maximum. The maximal scoring structure itself can be obtained by successively pushing features onto a stack, starting with ϕ_{n+1} (the "END" feature) and continuing with $\phi_{d(n+1)}$, $\phi_{d(d(n+1))}$ and so on until ϕ_0 (the "BEGIN" feature) has been pushed. The gene structure itself is then obtained by successive popping of features from the stack

until it is empty.

This algorithm is almost identical to the Viterbi algorithm for finding the sequence of states through a Hidden Markov Model that maximises the joint probability of the state path and the sequence. It is also very similar to Dijkstra's algorithm [32] for obtaining the shortest path through a directed weighted graph, generalised to account for negative edge-weights by Bellman [6].

The procedure above will find the gene structure with the highest score, regardless of whether it is consistent with the model or not. However, the constraints that define model consistency are all defined at the $source \rightarrow target$ level, as explained earlier. Since the dynamic procedure above works at this level too, it is straightforward to check that all constraints have been satisfied. When looping back over the sources for a given target, sources that give rise to a violation of a constraint are not considered valid sources for this target. If there are no valid sources for a given target, the target itself is invalidated, and not itself considered as a valid source for any subsequent target.

2.5 A probability distribution over gene structures

No matter how well the scoring function represents the characteristics of gene structure, it is often the case that the optimal (i.e. highest scoring) structure is not the correct one. It is therefore useful to know the relationship of the optimal gene structure to other candidate gene structures. I have adopted a probabilistic approach in assigning a *posterior probability* to firstly each input feature, and secondly each potential region (formed by candidate pairs of adjacent features). One can then ask for the features and/or regions with posterior probability greater some threshold, regardless of whether those features/regions are part of the optimal structure or not ("sub-optimal").

To calculate posterior probabilities, I first define a probability distribution over gene structures. If the given feature and segment scores are log-probabilities, then the probability of a gene structure can be calculated simply as an exponentiation of

the score. Some care is needed in the model to ensure that the whole DNA sequence is accounted for in every gene structure and that the sum of the probabilities of all gene structures sums to 1.

GAZE takes the somewhat more pragmatic stance that it is often impossible (or at the very least, extremely difficult) to formulate the scores as log-probabilities. Indeed, the scores presented by most signal-recognition programs are usually log probability *ratios* with respect to some background or "null" model. For this reason, GAZE imposes no restrictions upon the given feature scores except that they should generally increase monotonically with the degree of confidence in the correctness of the feature, i.e. that large scores are good, and small scores (or large negative scores) are bad. This means that the score for a complete gene structure can no longer be assumed to be a log-probability. A more general approach is therefore necessary.

2.5.1 Gene Structure probabilities

By treating gene structure scores $E(\phi)$ as "energy" values, we can use the Boltzmann distribution, ubiquitous in statistical physics, to define a probability distribution over all possible gene structures Φ:

$$Z = \sum_{\phi \in \Phi} \mathrm{e}^{E(\phi)} \tag{2.8}$$

$$\mathrm{P}(\phi) = \frac{\mathrm{e}^{E(\phi)}}{Z} \tag{2.9}$$

The Z of this fraction is known in statistical physics as the "partition function", and acts as a normalisation factor, making all gene structure sum to 1, satisfying the conditions for a discrete variable probability distribution.

In this formalism, the gene structure scores are interpreted as logarithms. This assumption is implicit in the design of the scoring function in that the scores of individual gene components are *added* to obtain the total score. The assumption of *natural*-logs specifically is not limiting since a log score with respect to a base k

can be transformed into a log score with respect to base e by multiplication by a constant.

The partition function can be computed with the following dynamic programming recurrence, similar to the forward algorithm for Hidden Markov Models [90], and almost identical to the score-maximisation algorithm presented earlier (the differences being the exponentiation step, and the replacement of maximisations with sums):

$$f(0) = 1 \tag{2.10}$$

$$f(i) = \sum_{j<i} f(j) \mathrm{e}^{Reg_{t(\phi_j) \to t(\phi_i)}(l(\phi_j), l(\phi_i)) + g(\phi_i)} \tag{2.11}$$

Each $f(i)$ denotes the sum of exponentiated scores of all of the "upstream" partial gene structures ending at feature ϕ_i. The sum of exponentiated scores of all upstream-partial gene structures ending at the "END" feature, i.e. all complete gene structures, is contained in $f(n+1)$. The probability of a gene structure ϕ can therefore be computed as:

$$\mathrm{P}(\phi) = \frac{\mathrm{e}^{E(\phi)}}{f(n+1)} \tag{2.12}$$

This approach is essentially due to Stormo and Haussler [109], the difference there being that gene structure scores were treated as the log of the joint probability of ϕ and the sequence S, and posterior probabilities are presented as explicitly conditional upon the sequence S, i.e. $\mathrm{P}(\phi|S)$. In the GAZE framework, the sequence itself is implicit. Considering GAZE posterior probabilities as conditional upon S however leads to some interesting correspondences to other methods, particularly those involving Hidden Markov models. This is discussed briefly at the end of this chapter, and in more detail in chapter 4.

2.5.2 Feature and Region posterior probabilities

Having defined a probability distribution over gene structures, it is now possible to define posterior probabilities for features and regions. The posterior probability of a feature ϕ_i, $\mathrm{P}(\phi_i)$, is the sum of the probabilities of all model-consistent gene structures that contain the feature ϕ_i. Likewise, the posterior probability of a region $\phi_i \rightarrow \phi_j$, $\mathrm{P}(\phi_i, \phi_j)$, is the sum of the probabilities of all model-consistent gene structures that include ϕ_i and ϕ_j as adjacent pairs of features.

Informally, a feature posterior probability can be interpreted as a measure of belief in the correctness of the feature, conditional upon the surrounding gene structure landscape. More informally still, it can be interpreted as an indicator of how well it can be accommodated in a "good" gene structure.

It is straightforward to calculate the sum of the probabilities of all gene structures consistent with a feature by running the forward algorithm while using the feature selection mechanism (section 2.3) to force the inclusion of the feature. This will give a new partition function value, corresponding to the sum over all gene structures that include the feature. Dividing this by the unrestricted partition function gives the desired posterior probability. However, this strategy requires a separate execution of the forward algorithm for each feature, and the computational expense of the algorithm makes the strategy infeasible.

To make the computation more efficient, we can compute "backward" analogues of the forward variables, $b(i)$, which store the sums of the exponentiated scores of all "downstream" partial gene structures that *start* with feature ϕ_i:

$$b(n+1) = 1 \tag{2.13}$$

$$b(i) = \sum_{k>i} b(k) \mathrm{e}^{Reg_{t(\phi_i) \rightarrow t(\phi_k)}(l(\phi_i), l(\phi_k)) + g(\phi_k)} \tag{2.14}$$

It can be shown that multiplying $f(i)$ by $b(j)$ $(i \leq j)$ corresponds to summing the exponentiated scores of all possible pairings of a partial upstream gene structure

ending at feature ϕ_i with a partial downstream structure *beginning* at feature ϕ_j [109]. Hence $f(i)b(i)$ is the sum of the exponentiated scores of all gene structures that include feature ϕ_i, and the posterior probability of ϕ_i, $\mathrm{P}(\phi_i)$ is:

$$\mathrm{P}(\phi_i) = \frac{f(i)b(i)}{f(n+1)} \tag{2.15}$$

Posterior *region* probabilities can be defined in a similar way:

$$\mathrm{P}(\phi_i, \phi_j) = \frac{f(i)\mathrm{e}^{Reg_{t(\phi_i)\to t(\phi_j)}(l(\phi_i), l(\phi_j)) + g(\phi_j)} b(j)}{f(n+1)} \tag{2.16}$$

A straightforward generalisation results in the posterior probability of a *partial* gene structure $\phi_i, \phi_j, \dots \phi_x$, i.e. covering an internal sub-region of the original sequence:

$$\mathrm{P}(\phi_i, \phi_j, \dots, \phi_x) = \frac{f(i)\mathrm{e}^{E(\phi_i, \phi_j, \dots, \phi_x)} b(x)}{f(n+1)} \tag{2.17}$$

If ϕ_i and ϕ_x, the boundary features of the sub-region, are chosen to be the start and stop of a single, individual gene, then the above is a posterior probability for that gene. However, one of the unfortunate consequences of the general approach adopted by GAZE is that it has no knowledge of which feature-types define the boundaries of individual genes. For this reason, individual gene probabilities are not reported by GAZE.

2.5.3 Stochastic traceback

A probability distribution over gene structures offers the possibility of a *stochastic* traceback procedure. In the standard traceback procedure, we choose, for a target feature ϕ_i, the source feature ϕ_j that gives rise to the highest scoring partial gene structure ending at ϕ_i. With stochastic traceback, we instead make use of a Boltzmann probability distribution over all model-consistent sources for a given target. This follows simply from the definition of the forward variables presented earlier:

$$P(k|i) = \frac{e^{Reg_{t(\phi_k)\rightarrow t(\phi_i)}(l(\phi_k),l(\phi_i))+g(\phi_i)}f(k)}{\sum_{j<i} e^{Reg_{t(\phi_j)\rightarrow t(\phi_i)}(l(\phi_j),l(\phi_i))+g(\phi_i)}f(j)} \tag{2.18}$$

Instead of choosing source that gives rise to the maximum partial-gene-structure score, we instead sample a source stochastically, with the relative probabilities of each source being computed with the above equation.

Stochastic traceback is an alternative method of identifying confident parts of a gene structure; parts of a gene structure that are conserved over many samples can be construed as more reliable (similar to the bootstrapping technique used in phylogenetics). Although I have provided an implementation of it in GAZE, my work has concentrated on the use of the posterior feature and region probabilities.

2.6 Practical considerations

2.6.1 Maintaining numerical stability

Implementing dynamic programming recursions in the obvious way can often lead to numerical underflows and overflows that even the most sophisticated modern floating point processor are unable to deal with gracefully. In the standard HMM formalism for example, each (state, residue) pair is assigned a probability, calculated as the product of probabilities for the state (conditional upon the previous state) and for the residue (conditional on the state). The joint probability assigned to a complete sequence of the order of a 10^6 bases, for even a simple HMM with few states, will therefore be of the order of $0.5^{1000000}$ (assuming an average probability per state-residue pair of 0.5, which is generously high). This gives an underflow error on my desktop calculator, and even though floating point units in modern processors would be expected to handle higher degrees of precision, it is not difficult to imagine a set of transition and emission probabilities for a given HMM architecture that will lead to underflow even on the most sophisticated processors.

The standard technique used in the field of HMMs is to work in log-space. Rather than multiplying probabilities, we add logarithms of probabilities. For example,

assuming that we use base 2 logarithms, $\prod_1^{1000000} 0.5$ becomes $\sum_1^{1000000} \log_2 0.5 = -1000000$. Classically, replacing many multiplications with additions would also lead to a performance improvement on some older computers. Modern floating-point unit technology makes this less true nowadays, but even on modern processors, addition should be no slower than multiplication.

For the dynamic programming performed in GAZE, the log transformation is not required in the maximum-based computation used to find the highest scoring gene structure consistent with the model (the Viterbi algorithm analogue). This is because the design of the scoring function, and its additive nature, places a log-based interpretation on the feature and region scores anyway. However, the log-transformation is required for the sum-based computations. For the forward algorithm, we define a new vector $F(i) = \ln f(i)$, and the recursions are defined in terms of $F(i)$ directly:

$$F(0) = 0 \tag{2.19}$$

$$F(i) = \ln \sum_{j<i} \mathrm{e}^{F(j)+Reg_{t(\phi_j)\to t(\phi_i)}(l(\phi_j),l(\phi_i))+g(\phi_i)} \tag{2.20}$$

The posterior probability of a gene structure is now calculated as:

$$\mathrm{P}(\phi) = \mathrm{e}^{E(\phi)-F(n+1)} \tag{2.21}$$

and the posterior feature probabilities calculated thus:

$$\mathrm{P}(\phi_i) = \mathrm{e}^{F(i)+B(i)-F(n+1)} \tag{2.22}$$

It is necessary to perform one last trick to avoid overflow when performing the exponentiations in 2.20. We can use the following observation:

$$\ln \sum_{x=a}^{b} \mathrm{e}^x \quad = \quad \ln \left(\mathrm{e}^k \sum_{x=a}^{b} \frac{\mathrm{e}^x}{\mathrm{e}^k} \right) \tag{2.23}$$

$$= \quad k + \ln \sum_{x=a}^{b} e^{x-k} \tag{2.24}$$

Any k can be used, but by storing the exponents in the summation, and choosing k to be the maximum of these exponents, we ensure that all exponentiations are 0 or less, eliminating the possibility of overflow.

2.6.2 Working within practical limits of space and time

Complexity of naive implementation

As noted earlier, the dynamic programming recursions for identifying the highest scoring gene structure (2.7) and for calculating the "partition function" over all gene structures, (2.11, 2.14) are essentially the same. The run-time and memory usage of the algorithms depends of course on the specific problem, but we can use complexity theory to reason about the growth in the requirement of these resources with respect to the size of the input. For GAZE, the algorithms proceed over input features, but since the number of features for a given DNA sequence would be expected to grow linearly with the length of the sequence, it makes no difference whether we define the problem size in terms of sequence length or in terms of the number of features. It is therefore convenient to talk about a problem size of n, which can be interpreted both as the length of the sequence region being considered, or the number of features attached to that sequence region.

Because the dynamic programming recursions are one-dimensional, the recursion variables can be stored as a vector rather than a matrix (as is the case with classical sequence alignment dynamic programming). The storage requirements are therefore $O(n)$.

Examination of 2.7, 2.11 and 2.14 shows that any algorithm must essentially examine all feature pairs in the list ϕ and perform a region score calculation. Since there are $\frac{n}{2}(n+1)$ pairs, this implies $O(n^2)$ region score calculations. Each region score calculation involves a length penalty component (which can be calculated in constant time by table look-up), and a segment score component. Assuming that

both 2.4 and 2.5 are implemented the way suggested by their definition, then in the worst case the segment calculation is linear in the number of segments. Although the actual number of respective features and segments for a given sequence may be, and often are, quite different, we would expect them to both scale in the same way with respect to the length of the sequence, i.e. linearly. This means that the segment calculation can be described as $O(n)$ in complexity, giving the algorithms a run-time complexity of $O(n^3)$ overall, making it apparently prohibitively expensive for large sequences.

In the remainder of this section, I outline two methods employed in the implementation of GAZE to improve both the theoretical worst-case and practical average-case run-time and storage complexity. In the next subsection, I describe a novel search-space pruning strategy employed in GAZE, which is the biggest contributing factor to its efficiency.

Segment pre-processing

The segment calculation for a candidate region, as defined by 2.3, 2.4 and 2.5, consists of a separate calculation for each segment qualifier listed in the rule that applies to the feature-pair defining the region. Each of these calculations in turn requires an ordination of the list of segments, for each firstly checking that it meets constraints defined in the qualifier, and secondly scaling the score according to how much of lies in the region of interest. Partitioned storage of the segments, primarily by type, but also by reading frame, allows the consideration of a much smaller list of candidate segments for a given segment qualifier, but the number of segments that need to be examined is still $O(n)$.

Since segments falling outside the considered region do not contribute to the score, sorting the segments by position along the sequence is the natural starting point towards reducing the number of segments that need to be examined. By sorting the segments by start position on the sequence, the segment with the *leftmost* start lying completely to the *right* of the end of the considered region can be identified

by a simple binary search. All segments to the right will lie completely outside the region. It is tempting to assume that the segment list can now be processed from this point to the left, stopping when a segment that has an end-position that is strictly to the left of the start of the considered region. In general segments may overlap and in the extreme the segment with the smallest start position may have the largest end position. In that case it will always be necessary to traverse leftwards from the point in the list identified by the binary search, right back to the start of the list. However, the rarity of this situation can be exploited by *indexing* the segments. In essence, we calculate and store an additional piece of information for each segment: the maximal right-position of all segments to the left:

$$I(\psi_i) = \max_{j<i} \psi_j.end \tag{2.25}$$

$$= \max(I(\phi_{i-1}), \psi_{i-1}.end) \tag{2.26}$$

The second equality gives rise to a simple linear-time dynamic programming algorithm to calculate the segment indices, and this only needs to be performed once for each segment list, before the main score/probability dynamic programming.

It is interesting to note that although partitioned storage and indexing of the segments improve the worst-case time spent performing the segment score calculations, they do not change the theoretical worst-case *complexity*. At the extreme, the when the region being considered encompasses the whole sequence the computation is still O(n). This is a classic example of where worst-case complexity is a misleading indicator of the expected increase in run-time with problem size.

As a final note on this subject, it must be said that it *is* straightforward to implement the segment calculation in such a way as to make it constant time. This technique is employed in several gene prediction programs, but not GAZE. The technique involves keeping cumulative per-residue arrays for each specific set of constraints referred to in the segment qualifiers of a model. If ψ^q is the list of segments that match a specific set of segment qualifier constraints q then the cumulative array for this constraint set, C_{ψ_q} can be defined as:

$$C_{\psi^q}(i) = C_{\psi^q}(i-1) + Seg^Q(\psi^q, i, i) \quad (2.27)$$

where $Seg^Q(\psi, x, y)$ is as defined in 2.5. Like the segment indexing presented earlier, the $C_{\psi^q}(i)$ arrays can in theory be calculated in linear time in advance of the main dynamic programming. Then, the segment calculation performed during the Viterbi, forward and backward algorithms becomes:

$$Seg^Q(\psi^q, x, y) = C_{\psi^q}(y) - C_{\psi^q}(x-1) \quad (2.28)$$

There are a number of reasons why this technique is not implemented in GAZE. Firstly, it requires much more memory. It is true that this will only become as problem when very large sequences are being analysed (of the order of Megabases long), but for small sequences, the run-time reduction afforded by the technique becomes negligible.

Secondly, and most importantly, it is only applicable for a specific kind of segment scoring, namely the "projected per-base" approach defined in 2.5. In addition, the technique requires the segments to be partitioned in advance into lists matching each segment qualifier referred to in the model. However, for some constraints (namely "completeness" and "exactness"), it is not known until the dynamic programming stage which segments will satisfy them. The cumulative arrays technique is therefore only available to segment qualifiers with no completeness or exactness constraints, scored by equation 2.4. Given its limited applicability, it was decided insufficiently worthwhile to implement.

Split-and-merge for whole genome analysis

Although the space requirements of GAZE are linear in the length of the sequence, the memory of a standard desktop computer is not likely to be sufficient to handle feature-sets from the complete genomes of eukaryotic organisms. To analyse whole genomes, it is necessary to design a method that is constant in its memory usage, regardless of the size of the input feature-set. Such an aim is not fanciful, especially

considering that the dynamic programming search-space pruning method (presented below) makes the algorithm effectively local. One possible strategy might therefore be to discard elements of the V, F and B vectors, as well as the features corresponding to these elements, when they are not needed any more. This idea is similar in essence to linear-space sequence alignment methods [80]. However, although it is possible to obtain the score of the optimal gene structure in this way, the gene structure itself is more difficult to obtain; the standard trace-back procedure is no longer possible.

A natural constant-memory method for obtaining both the highest scoring gene structure and posterior feature and region probabilities involves *off-lining*, where parts of the dynamic programming structures, and the features the themselves, are written to disk, discarded from memory, and read back in when required. I have implemented such a method, but approached the problem from a slightly different angle. The technique is based upon a split-and-merge strategy, which conceptually involves splitting the input into several manageable chunks, running GAZE separately on each one, and finally merging the results together. This has been implemented in a Perl script called GENOME_GAZE.

The first stage of GENOME_GAZE is the split, but GAZE itself provides an option to make this stage trivial. Specifically, it can be told to consider a specified subsequence window of the given arbitrary sized query sequence / feature-set; the input DNA sequence and features sets do not therefore have to be physically split at all. The "split" phase of GENOME_GAZE therefore involves running GAZE on windows $w_1 \cdots w_k$ to produce output files $o_1 \cdots o_k$, where the window size is chosen according to the available computational resources (the bigger machines available, the bigger the window size can be), and k is chosen to be large enough to cover the whole input sequence region with a specified overlap between w_i and w_{i+1}. The overlap allows the second, "merge" phase of GENOME_GAZE to be performed.

An overlap is necessary between subsequence windows w_i and w_{i+1} because there may be cases where a predicted gene structure straddles the boundary between

two windows. The final output gene prediction is formed generally by pushing the features in the output files onto a list in sequence order from o_1 to o_k. However, for the first feature ϕ_x in o_i that lies in a region that is also covered by the start of w_{i+1}, o_{i+1} is searched for the occurrence of that feature. If it is found, the rest of the features in o_i are ignored, and the pushing of features continues from that point in o_{i+1} at which ϕ_x was located. If it is not found, ϕ_x is pushed onto the list, and o_{i+1} is searched for the occurrence of ϕ_{x+1} in o_i. This continues until the appropriate cross-over point from o_i to o_{i+1} is identified.

When the output files o_i consist not of feature-lists representing predicted gene structures, but the complete input feature set ordered by sequence position, with posterior probabilities attached to each feature, a slightly simpler strategy suffices, where the midpoint of the overlapping region between o_i and o_{i+1} is chosen as the cross-over point between the two regions.

Split-and-merge offers a natural parallelisation strategy, because each window w_i can be analysed independently of other windows. Only the final stage of forming the consensus gene structure for all windows relies upon their order in the sequence, and therefore cannot be performed until all windows have been processed first by GAZE. This is one of the key advantages of post-processing approach via GENOME_GAZE over a split and merge algorithm in GAZE itself.

2.6.3 A novel pruning strategy

Earlier it was described how the region-score calculation is made less computationally expensive in order to reduce the overall run-time of the algorithm. An complementary approach is to reduce the number of region calculations that need to be performed, which is essentially $O(n^2)$ in the number of features. To this end, two pruning strategies have been implemented, one exact and one heuristic. Both rely upon the examination of the search-space in a directed manner.

In the calculation of the F vector outlined earlier (and likewise for the V and B vectors), the elements need to be computed in a directed way, starting with $F(0)$

and ending with $F(n+1)$. The calculation of $F(i)$ relies upon the values $F(j)$, $j < i$, which corresponds to examining the source features ϕ_j for a given target feature ϕ_i. These sources *need not* be visited in any directed way, but doing so provides opportunities for pruning. In particular, sources for a given target are examined firstly in order of type, and secondly (for sources of a given type) in order of proximity to the target, i.e. $\phi_{i-1}, \phi_{i-2} \dots \phi_0$.

A pruning strategy based upon model constraints

The first method makes use of the fact that certain constraints specified in the model can be used to prune away partial gene structures that cannot possibly be model-consistent. In particular, when scanning back through the sources for a given target ϕ_k, violation of an interruption or maximum distance constraint by the region $\phi_i \rightarrow \phi_k$ defined by a specific source ϕ_i means that all subsequent regions $\phi_j \rightarrow \phi_k$, $j < i$ will violate the same constraint. Therefore sources ϕ_j $(j < i)$ need not be considered for target ϕ_k.

Most models used in practice will not contain interruption or maximum distance constraints in many rules. In fact, for the models explained in chapter 3, only rules defining protein-coding regions make use of interruption constraints, to disallow in-frame stop codons. A more general pruning strategy is therefore necessary.

A pruning strategy based upon Dominance

The main idea of the strategy is based upon this aggressive assumption: when accumulating information for gene structures ending at a particular target feature ϕ_k, if the contribution made by source ϕ_j is insignificant when compared with that made by another source ϕ_i, then we can *ignore* ϕ_j as a potential source for all subsequent targets of the same type as ϕ_k. I formalise a general approach based upon this idea, and then consider cases where the main assumption might not hold, refining the strategy at each step. The method is presented in terms of the computation of the forward score $F(i)$, but the method applies equally to the backward $(B(i))$ and

Viterbi ($V(i)$) calculations.

To formalise the notion, I introduce the concept of *dominance*. For a given target feature ϕ_k, and two valid source features *of the same type*, ϕ_j and ϕ_i, where $j < i$, ϕ_i *dominates* ϕ_j if the contribution made to forward score for ϕ_k ($F(k)$) by the component involving ϕ_j is insignificant (given the limits of machine precision) compared to the contribution made by the component involving ϕ_i. More precisely:

$$\mathrm{Dom}(\phi_i, \phi_j, \phi_k) \qquad \text{if} \qquad \mathrm{Rdiff}(\phi_i, \phi_j, \phi_k) > X \tag{2.29}$$

The right-hand-side can be read as the relative difference between the contribution made to the forward score of ϕ_k by respectively ϕ_j and ϕ_i, and is defined thus:[2]

$$\begin{aligned} \mathrm{Rdiff}(\phi_i, \phi_j, \phi_k) \quad = \quad & Reg_{t(\phi_i) \to t(\phi_k)}(l(\phi_i), l(\phi_k)) + F(i) \\ & - Reg_{t(\phi_j) \to t(\phi_k)}(l(\phi_j), l(\phi_k)) - F(j) \end{aligned} \tag{2.30}$$

It is important to note that because the F vectors are computed in log-space, a *difference* of X between the ϕ_i and ϕ_j components means that the former is e^{X} times greater than the latter in probability space. A small value for X (20-30) is therefore sufficient for ϕ_i to dominate ϕ_j.

The pruning strategy relies on the fact that, at least under certain conditions, the dominance is *time-invariant*. That is, if ϕ_i dominates *all* sources of the same type ϕ_j (for $j < i$) with respect to a target ϕ_k, then it will dominate the same sources for subsequent downstream targets ϕ_q of the same type as ϕ_k. When considering potential sources for a ϕ_q, we need not therefore search back further than ϕ_i. The feature ϕ_i in this case is an *omnipotent* source of its type with respect to targets of the same type as ϕ_k.

[2]the contribution to the forward score also contains the given score of ϕ_k, but since this is the same for sources i and j, it cancels. In the calculation of the *backward* score however, the relative difference includes terms for the given scores of targets ϕ_i and ϕ_j.

In the implementation, a matrix $Onmi(src, tgt)$ is maintained which stores for each $src \rightarrow tgt$ rule, the index of the current omnipotent source of type src for targets of type tgt. As the dynamic programming progresses, the dominance condition is continually checked, and the $Omni$ matrix updated. The desired effect is that at any time in the algorithm k, $Omni(src, t(\phi_k))$ will not be much less than k for all source feature types src. This means that only a constant number of sources need to be examined for each target, rather than $O(n)$ sources.

The assumption of time invariance underlying this pruning technique is important for its soundness. In order to reason about the conditions under which time invariance holds, it is convenient to re-define the assumption in terms of the relative difference between the contribution made to the forward score of ϕ_k by respectively ϕ_j and ϕ_i (equation 2.30). The assumption is therefore as follows: this quantity will remain constant or increase when measured with respect to subsequent downstream targets ϕ_q of the same type as ϕ_k, preserving the domination condition. However, this assumption is *not* valid in the following circumstances:

1. Feature ϕ_q is in a different reading frame to ϕ_k, the relevant rule includes a phase constraint, and the region $\phi_i \rightarrow \phi_q$ violates this constraint; in this case, the omnipotent feature is not even a valid source for ϕ_q.

2. The region $\phi_j \rightarrow \phi_k$ violates a DNA constraint which is not violated for the region $\phi_j \rightarrow \phi_q$.

3. The region $\phi_i \rightarrow \phi_q$ violates a DNA constraint, even though $\phi_i \rightarrow \phi_k$ did not.

4. Feature ϕ_j is located at the same position as the upstream end of an "exact_match" segment; feature ϕ_k lies upstream from the location of the downstream end of this segment, which therefore does not contribute towards the score for region $\phi_j \rightarrow \phi_k$. Feature ϕ_q however lies at the same location as the downstream end of the segment, which *does* contribute to the score for region $\phi_j \rightarrow \phi_q$.

5. The *src* $\rightarrow$ *tgt* rule includes a length penalty function, and difference in the penalties for $\phi_j \rightarrow \phi_k$ and $\phi_i \rightarrow \phi_k$ is *greater* than the difference between penalties for $\phi_j \rightarrow \phi_q$ and $\phi_i \rightarrow \phi_q$.

In all these cases, it is wrong to consider ϕ_i to be an omnipotent source of that type with respect to features of type ϕ_k. It is therefore necessary to revise the ideas of dominance and omnipotence, addressing each of the problems in turn.

The first problem can easily be addressed by adding an extra "absolute-reading-frame" dimension to the *Omni* matrix. This allows us to represent a different set of omnipotent sources not only for each target type, but for each target-type in each absolute reading frame.

The second and third problems can be dealt with by revising the domination criterion with a $DNA_constraint_condition$ which states that ϕ_i does *not* dominate ϕ_j with respect to ϕ_k if (a) $\phi_j \rightarrow \phi_k$ is violated by a DNA constraint, or (b) ϕ_i has the potential to violate a DNA constraint for a future region $\phi_i \rightarrow \phi_q$.

The fourth problem is addressed by adding a $Exact_Segment_condition$ to the domination criterion which states that ϕ_i does *not* dominate ϕ_j with respect to ϕ_k, if (a) the relevant rule contains a segment qualifier with an "exact match" condition and (b) a segment of the appropriate type begins at the same location ϕ_j but extends *beyond* feature ϕ_k.

The final problem is very likely to occur, because many useful length penalty functions used in practice will have this behaviour. My approach is to remove the length-penalty component from the domination condition. Equation 2.30 then becomes:

$$\begin{aligned} \text{Rdiff}(\phi_i, \phi_j, \phi_k) &= Seg_{t(\phi_i)\rightarrow t(\phi_k)}(l(\phi_i), l(\phi_k)) + F(i) \\ &\quad - Seg_{t(\phi_j)\rightarrow t(\phi_k)}(l(\phi_j), l(\phi_k)) - F(j) \end{aligned} \tag{2.31}$$

In doing this we must now consider the possibility that ϕ_i dominates ϕ_j by the above criterion, but *only* when the length penalty component is ignored, i.e. if

we have inadvertently extended the criterion in trying to restrict it. If the length penalty for $\phi_j \to \phi_k$ is greater than that for $\phi_i \to \phi_k$, then ϕ_i will still dominate ϕ_j even if the penalties are included in the comparison. Although many length penalty functions used in practice would be expected to be monotonically increasing with distance, one of the strengths of GAZE is that it allows the definition of arbitrary length penalty functions. However, it is likely that this facility will be most useful in the early, small-distance portion of the functions. All functions used in practice will be *eventually* monotonically increasing (or at least monotonically constant); a length function for which this is not the case implies a kind of region that, in the limit, become *more* likely the longer it is. So, assuming that each function has a distance d at which it becomes monotonic, the relative difference *including* the length penalty (2.30) is at least as big as the relative difference *ignoring* it (2.31) if the distance from ϕ_i to ϕ_k is bigger than d. If this is the case, then $\phi_j \dots \phi_k$ will also be in the monotonic part of the function, and furthermore so will the regions $\phi_j \dots \phi_q$ and $\phi_i \dots \phi_q$, for all subsequent ϕ_q of the same type as ϕ_k.

It is straightforward to derive the monotonic point for length penalty function p, Mpoint(p):

$$\text{Mpoint}(p) = \min_{x=0}^{\infty} s.t.\forall yz\,[(x < y < z) \to p(x) \leq p(y) \leq p(z)] \qquad (2.32)$$

Given these considerations, I formulate a revised, and final, domination condition:

$$\text{Dom}(\phi_i, \phi_j, \phi_k) \qquad \text{if} \qquad \begin{array}{l} \text{Rdiff}(\phi_i, \phi_j, \phi_k) > X \quad \text{and} \\ Exact_Segment_Condition \quad \text{and} \\ DNA_Constraint_Condition \quad \text{and} \\ l(\phi_k) - l(\phi_i) \geq \text{Mpoint}(Len_{t(\phi_i)\to t(\phi_k)}) \end{array} \qquad (2.33)$$

The domination/omnipotence pruning strategy means that in practice, it is only ideally necessary to consider a constant number of sources for each target. This means that that effective number of pairwise feature comparisons (and therefore

region-score calculations) necessary is now effectively $O(n)$. When considering also the segment indexing strategy explained earlier (which makes the region-score calculation effectively $O(\log n)$), the run-time of complexity of GAZE can be described as pseudo log-linear. My experience shows log-linearity to be an upper bound, at least on my own library of GAZE models (see chapters 3 and 5); many models display linear growth in run-time with sequence length.

Linear-time dynamic programming algorithms for gene prediction have been published before. In particular, there is at least one example of a linear-time algorithm over an external model of gene structure ([52], and section 2.7). However, the flexibility of GAZE in allowing user-defined length-penalty functions and segments makes it difficult to design a linear time algorithm for obtaining both the highest scoring gene structure, and posterior feature and region probabilities.

2.7 Relationship to other similar systems

2.7.1 Other gene prediction toolkits

Although the signal and content detection techniques used by most existing integrated gene prediction systems are hard-coded in the software, their specific parameters are usually abstracted into an external file which is read at run-time. This allows the development of distinct parameter sets for different organisms for example. In this way, the majority of systems are configurable. Some systems however, like GAZE, take this idea further in attempt to provide a "toolkit" for the development of gene prediction methods.

Dong and Searls [33] for example represent the rules of gene structure as formal grammars [58]. They have constructed a toolkit for the graphical definition and computational parsing of certain kinds of grammars, based upon the Prolog programming language. They have used their toolkit to produce the GENLANG gene prediction program.

The work of Gelfand, Roytberg and co-workers is comparable to GAZE in the

that it forms the basis of a research tool for the investigation of gene prediction methods. Their technique, called *Vector Dynamic Programming*, identifies the *set* of gene structures that is guaranteed to contain the "optimal" structure with respect *all* scoring functions that adhere to certain mathematical properties [95]. The fact that this set is usually orders of magnitude smaller than the complete set of all possible gene structures allows for the rapid investigation of several different scoring functions. They have used their tool to design the scoring function that is used in the GREAT program [46].

The GENAMIC algorithm [52] lying at the heart of the GENEID system ([89], [84]) has many elements in common with GAZE. In particular, it accepts as input a list of scored candidate exons in GFF, and also a model for how the different types of exon fit together into complete gene structures. It then identifies the highest scoring exon assembly consistent with the model rules. Although similar in these respects, there are notable differences between GAZE and GENAMIC. Firstly, GAZE works with the signal and content data *before* it has been pre-processed to produce a set of candidate exons with pre-assigned frames and scores. This gives greater flexibility in the way in which external evidence is incorporated. Secondly, the model constructs offered by GENAMIC are more restricted than those offered by GAZE. In particular, GENAMIC allows the specification of a minimal and maximal distance between exons, but not arbitrary length penalty functions. Thirdly, GENAMIC does not compute posterior probabilities. These last two differences in particular however mean that the dynamic programming recursions of GENAMIC are less general and therefore more amenable to optimisation; the highest scoring gene structure is identified by means of a algorithm for which the run-time grows strictly linearly with the number of candidate exons, making it extremely fast.

Some of the motivations for the DYNAMITE system [9] were similar to those for GAZE. It is based upon the observation that many differing applications in bioinformatics have at their heart quite similar dynamic programming algorithms. Furthermore, implementation of these algorithms can be time consuming and error-

prone. DYNAMITE provides a simple language for the specification of such dynamic programming recursions, allowing large and complex models to be defined in an intuitive way. A compiler then generates code (in C) for the specified recursions that can be linked into a stand-alone application. DYNAMITE differs from GAZE primarily in the way that it is designed for *sequence alignment*, rather than feature selection. It is therefore particularly suitable for development of sequence-similarity based gene prediction applications; a large part of code in the GENEWISE program [10] for example was generated by the DYNAMITE compiler.

2.7.2 HMM methods

In outlining some of the common methods for the prediction of gene complete gene structures in chapter 1, I drew a distinction between gene fragment assembly techniques and Hidden Markov models. I have presented GAZE from the angle of gene fragment assembly, but it can also be viewed as a kind of Generalised Hidden Markov model. To see the correspondence, it is instructive to look at other systems based explicitly on GHMMs. GENSCAN [21] and GENIE [72] are examples of such.

Both GENIE and GENSCAN work in practice by first scanning the sequence for candidate state transitions. These are used as anchor points for a dynamic programming procedure to identify the state path with highest probability. In this way, they can be viewed feature-based methods (like GAZE), rather than classical single-base-at-a-time HMMs (for example HMMGENE [68]).

The advantage that GHMMs have over standard Hidden Markov models is that they allow the lengths of the regions to be modelled by arbitrary, non-geometric probability distributions. This is particularly useful because the lengths of protein-coding exons in particular are not geometrically distributed (see chapter 1). This aspect of GHMMs is reflected in GAZE by the length penalty component of the scoring function. For reasons of efficiency, GENSCAN in particular restricts the use of fully defined length probability distributions to alternating states (in practice those corresponding to protein-coding regions). The search-space pruning in GAZE means

that it is not limited in this way. It is not clear whether this is the case for GENIE.

In GENSCAN, the dynamic programming is performed over an assumed, fixed GHMM architecture. Like GAZE, GENIE is not subject to the same restriction. Signal and content sensors are treated as external modules which results in a "plug-and-play" architecture. Unlike GAZE however, it is necessary for GENIE to make specific assumptions about the scores reported by the components, namely that they are probabilities. The treatment of feature scores by GAZE as arbitrary "energies" makes it strictly more general.

Since the emission probabilities of a GHMM can be reflected by feature and segment scores, and the length probability distributions by length penalty functions, only the transition probabilities of a GHMM do not have a direct analogue in GAZE. They can however be represented by adjusting the appropriate length penalty function. Since it is possible to define a distinct penalty function for every pair of feature types, this is fully general. The disadvantage of such an approach is that the resulting models lack the intuitive appeal of a finite state automaton.

One of the key aspects of HMMs is that they are fully probabilistic, defining a joint probability distribution over gene structures *and* sequences. Careful model construction and scoring of features and segments can allow GAZE gene structure scores to correspond directly to (log) joint probabilities, although this will often not be possible when using data from external sources. Taking this idea to the extreme, the representation of a standard base-at-a-time HMM is also possible in GAZE. This could be done by having a feature type for each state, and a specific feature of each type for each position of the sequence, with emission probabilities represented by the feature scores and transition probabilities by length penalties[3]. This of course would take away one of the advantages of GAZE, namely the representation of a large number of DNA bases by a relatively small number of sequence features.

[3]Interruption constraints would need to be used to ensure that only the previous base is considered at each stage of the dynamic programming.