The Pennsylvania State University

The Graduate School

College of Engineering

IMPROVED PAIRWISE ALIGNMENT OF GENOMIC DNA

A Thesis in

Computer Science and Engineering

by

Robert S. Harris

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

December 2007

The thesis of Robert S. Harris was reviewed and approved* by the following:

Webb Miller
Professor of Biology and Computer Science and Engineering
Thesis Adviser
Chair of Committee

Padma Raghavan
Professor of Computer Science and Engineering

Francesca Chiaromonte
Associate Professor of Statistics and Health Evaluation Sciences

Ra j Acharya
Professor of Computer Science and Engineering
Head of the Department of Computer Science and Engineering

*Signatures are on file in the Graduate School

ABSTRACT

Advances in DNA sequencing technology have fueled a rapid increase in the number of sequenced vertebrate genomes, and we anticipate an explosion in the number of genomes sequenced in the near future. Detecting similarities between genomes is a valuable technique in discovering functional elements, and sequence alignment is the primary tool for discovering similarities. The quality of alignments is affected by several user-specified control parameters. The parameters are so little understood that most users simply use default settings. We seek to change that, to have the program automatically infer appropriate parameter choices from statistics derived automatically from the sequences.

We introduce a program, INFERZ, which addresses part of the inference problem, inferring substitution and gap scores according to a mathematically sound model. Further, we explore the usefulness of iterating inferred scores to convergence. We test this process on both simulated and actual genomic data, and show that iteration will converge in general, but found that converged scores were not a consistent improvement.

INFERZ has a synergistic relationship with LASTZ, our improved drop-in replacement for the widely used alignment program BLASTZ. INFERZ makes repeated calls to LASTZ to test score sets, and LASTZ provides the user an option to have INFERZ decide what scoring parameters to use. Compared to BLASTZ, LASTZ adds a richer set of seeding strategy choices, supports alignment to probabilistic sequences and reduces memory requirements. Additionally, disciplined software techniques make it a better platform for continued experimentation.

TABLE OF CONTENTS

List of Figures vi

List of Tables vii

Acknowledgements viii

Chapter 1 Introduction 1

1.1 Motivation 1

1.2 A Brief Introduction to Sequence Alignment 4

1.3 Quantum Alignment 7

1.4 Related Work 7

Chapter 2 BLASTZ-type Aligners 9

2.1 Seeding and Gap-free Extension 9

2.2 Chaining 12

2.3 Anchoring and Gapped Extension 12

2.4 Interpolation 12

Chapter 3 Spaced Seeds 14

Chapter 4 LASTZ 17

4.1 Seeding Strategies 17

4.2 Memory Requirements 19

4.3 Overweight Seeds 25

4.4 Quantum Sequence Support 25

Chapter 5 INFERZ 27

5.1 Finite State Automaton for Neutral DNA 27

5.2 Inferring Scores From Alignments 29

5.3 Empirical Agreement with the FSA Model 30

5.4 Inferring Scores from Sequences 32

5.5 Experimental Results on Simulated Genomic Sequences 34

5.6 Experimental Results on Actual Genomic Sequences 35

Chapter 6 LASTZ Experimental Results 43

6.1 Hashed Diagonal Extent Table 43

6.2 Z-step 43

6.3 Twin Hit Seeds 45

Chapter 7 Conclusions and Future Work 49

Bibliography 51

Appendix A Glossary 55

Appendix B Methods 56

B.1 Analysis of 28-Vertebrate Alignments 56

B.2 Simulated Sequence Pairs 56

B.3 ENCODE Data 58

B.4 Syntenic Chromosomal Data 59

B.5 Receiver Operating Characteristic 59

Appendix C Quantum DNA Techniques 62

C.1 Inferring Ancestral Quantum Sequence 62

C.2 Choosing a Quantum Alphabet 64

C.3 Quantum versus DNA Scoring 65

C.4 DNA Ball Generation 66

Appendix D Linear Inference Techniques 68

D.1 Linear Discriminator 68

D.2 Inverse Alignment 69

Appendix E Seed Packing 71

LIST OF FIGURES

1.1 Alignment and scoring example 5
2.1 Alignment stages 10
2.2 Y-drop alignment region 11
3.1 Seed sensitivity distribution 15
4.1 Seed-word position table, linked list implementation 23
4.2 Seed-word position table, last/previous implementation 24
5.1 Pair finite state automaton 28
5.2 Gap Lengths Distribution 31
5.3 Ungapped Run Lengths Distribution 32
5.4 Iterated Scoring Inference 33
5.5 Scores inference convergence on simulated sequence pair 36
5.6 Scores inference performance on simulated sequence pair 37
5.7 Scores inference performance on real sequence pair 39
5.8 Scores inference performance adjusted for true positive rate 40
5.9 Estimating false positive rate by alignment to backward sequence 41
5.10 False positive scoring distributions 42
6.1 Z-step experimental results 46
6.2 Twin hit seed 47
B.1 ROC example 61
C.1 Quantum inference example 64
C.2 Quantum scoring context 66
E.1 Seed packing 72
E.2 Hard-coded seed packing 73

LIST OF TABLES

6.1 Hash failures in the diagonal extent table 44
6.2 Statistics for single hits vs. twin hits 48
B.1 Genome-wide alignment probabilities observed in six vertebrates 57
B.2 Alignment scores derived from observations in six vertebrates 57
E.1 Comparison of greedy seed packing to optimal 73

ACKNOWLEDGEMENTS

I first would like to thank the people who started me on my road to research. Jens Gregor, my M.S. advisor some years ago at the University of Tennessee, was the first to introduce me to the concept of sequence alignment. And Michael Thomason, another member of my M.S. committee, who in 2001 advised me that if I was looking for a challenge, I should return to grad school.

Second, I would like to thank my current committee members and my advisor, Webb Miller, for giving me the opportunity, and the encouragement, to apply what I knew about alignment to what was an entirely new field for me.

Third, I would like to thank the members of our extended group at Penn State, the Center for Comparative Genomics and Bioinformatics. Owning little prior knowledge of Statistics, and next to none of Biology, it was invaluable to me to be in a group with people of mixed disciplines. Webb Miller, Francesca Chiaromonte, Wojciech Makalowski, Ross Hardison, Laura Elnitski, David King, James Taylor, Min Mei Hou and Kateryna Makova were especially helpful, but everybody in the group contributed to my education. Extending beyond Penn State, I thank Adam Siepel for his work in creating quantum ancestral sequences.

Lastly, I want to express my appreciation for the free interchange of data in this research field. The experiments in this paper were made possible by data from RefSeq and the ENCODE project, and the multitude of free resources at the UCSC Genome Browser website.

.

To all my ancestors—every bit of me is a little bit of you.

Chapter 1

Introduction

1.1 Motivation

Over the past decade, advances in DNA sequencing technology have produced whole genome sequences for an increasing number of species. This increase has spurred the field of comparative genomics, in which sequences from many species are compared *in silico* to reveal highly similar segments common to many species. A high similarity level is indicative of the presence of conserved elements from a common ancestor. As evolution operated along the different branches leading to the present-day species, these elements suffered fewer mutations, maintaining more of their similarity than the genome in general. The prevailing explanation is that such elements must be under selective pressure, and they are expected to have important biological function.

The basic tool for identifying inter-species similarities is a sequence aligner. Given sequences of DNA, an aligner identifies segments in one sequence that are similar to segments in another. Similarity is defined mathematically, reflecting evolution by rewarding nucleotide matches and penalizing mutations such as mismatches, insertions and deletions. An aligner seeks the segments of highest similarity.

What constitutes high similarity depends on the species being compared and the evolutionary distance between them. For example, identifiable conserved elements between human and macaque are, on average, 90-95% identical, while in the more distantly related chicken average identity is in the 65-70% range. In the latter case, the signal is weaker, and it is more difficult for an aligner to distinguish real homology (biological relatedness) from similarities that occur by chance. To achieve best performance—balancing sensitivity and specificity when aligning to a newly sequenced genome—the aligner's control parameters must be tweaked. Present aligners require the user to choose these parameters but offer little guidance on how the choices should be made. We seek to change this, to make alignment closer to a turnkey operation by building control choices into the aligner. We want the aligner to take a quick look at the sequences and infer good control choices before proceeding with alignment.

A method for inferring substitution scores from sequences was proposed in Chiaromonte et al. (2002), and was used to create the default scoring matrix for BLASTZ (Schwartz et al., 2000, 2003). Inferring gap scores that work well in practice has been more elusive, and in practice they are set by intuition. As an example of current scoring practices, we note the scoring used for the BLASTZ stage of multiple alignments for the UCSC genome browser. The default scores, which were originally chosen for human vs. mouse, are used to align human to species ranging (in distance from human) from rhesus to mouse. For species more distant from human, ranging from opossum to lizard, a second scoring matrix is used, reducing mismatch penalties by about 25% across the board, with no change in gap scores. The very close chimp species has its own scoring matrix, with much stiffer mismatch and gap penalties.

In addition to substitution and gap scores, a turnkey aligner must also make choices for seeding strategies and score thresholds, saving the user from uninformed choices.

In this thesis we deal only with pairwise alignment, in which only two sequences are involved. When we use the words “aligner” or “alignment”, we mean pairwise. Moreover, we are primarily interested in aligning DNA sequences, in which the alphabet consists only of the four characters A, C, G and T. However, we do generalize this to allow one of the sequences to be probabilistic profiles, which we call quantum DNA (section 4.4).

The author has incorporated the ideas presented here in INFERZ, a program to infer scoring matrices, and LASTZ, a replacement for the widely used BLASTZ. Further enhancements have been made to reduce memory usage, to increase the size of sequences that can be aligned on modern desktop workstations with 1G byte of memory. The practical limit, for the current implementation of LASTZ on a 1G machine, is 125Mbases, about half the length of the longest human chromosome. LASTZ *can* perform full chromosome-to-chromosome alignments in 2G of memory.

As a result of its generality, LASTZ, in its current implementation, is about 20% slower[1] than BLASTZ. The gapped alignment stage of LASTZ is nominally faster (about 2%) and for I/O. However, these advantages are overwhelmed by the per-bp cost of general seeding strategies, and of some memory-for-time tradeoffs. The reduced memory usage of LASTZ provides a window of speed improvement for large sequences. On a 1G machine BLASTZ hits the memory limit when sequences reach about a 100M bases; memory swapping reduces performance above this point. For LASTZ this occurs at around 125M bases.

Another goal of the author's work was to improve the state of the source code in BLASTZ. The original BLASTZ grew out of several years of work by three authors, Scott Schwartz, Zheng Zhang and Webb Miller, each with different coding styles. In addition, some of the source code was collected from parts of earlier programs, leading to inconsistent terminology within the code. LASTZ is nearly a complete rewrite, by a single author, with consistent style and terminology, and a greater emphasis on internal documentation. This makes it a better platform for continued experimentation.

The author expects to continue improving INFERZ and LASTZ, but current implementation includes the following major improvements:

INFERZ:

- Simple inference of substitution scores.
- Iterated inference of substitution scores.
- Iterated inference of gap scores.

[1] All timing tests were run on a 2GHz Intel Dual iMac with 1GB, lightly loaded.

LASTZ (compared to BLASTZ):

- A wider range of seeding choices, including twin hit seeds, transition-match seeds, multiple transitions and user-definable seed patterns.
- Reduced memory requirements, allowing processing of larger sequences on a 1G machine.
- Use of larger alphabets (up to 255 symbols) for one sequence, to support alignment of quantum DNA sequences.
- Multiple output formats, including the widely used MAF and AXT.
- Single-author rewrite, providing a better platform for future experimentation.

1.2 A Brief Introduction to Sequence Alignment

Sequence alignment involves finding similarities between two sequences. Given two sequences of symbols from some alphabet Σ, we can construct an alignment of a subsequence of each by first inserting spaces into the subsequences so that they have the same length, then arranging the modified subsequences as two rows, one above the other, subject to the constraint that no column contains only spaces. Columns without spaces are called matches or substitutions, while runs of spaces are called gaps. Figure 1.1(a) shows an example alignment.

Alignments provide a natural definition of the similarity between sequences. Given a scores set $\{s_{xy}, s_{\text{open}}, s_{\text{extend}}\}$, each match is scored as s_{xx}, each substitution as s_{xy}, each gap of length n as $s_{\text{open}} + ns_{\text{extend}}$ and the alignment as the sum of its component scores. Scores are usually positive to reward matches and negative to penalize substitutions and gaps. Charging separate penalties for the presence and length of a gap is called affine-gap scoring. Figure 1.1(b) shows typical scores for aligning DNA. The similarity between two sequences is the maximum score of any alignment between them, and alignments giving the maximum score are called optimal. It is often instructive to view an alignment as a dot-plot, which shows the positions of each aligned pair as a point in a Cartesian space indexed by one sequence along each axis.

Given two sequences X and Y, it is possible to find the similarity between two sequences using a dynamic programming algorithm (Smith and Waterman, 1981; Gotoh 1982). We use the notation $|X|$ to indicate the length of X, and $X_{[1..i]}$ to represent the length-i prefix of X, or an empty sequence when i is zero. Define $S_{i,j}$ to be the similarity between $X_{[1..i]}$ and $Y_{[1..j]}$. Then S has the recurrence relation (1.1) and the similarity of X and Y is $S_{|X|,|Y|}$.

$$\begin{aligned}
I_{i,0} &= s_{\text{open}} + i s_{\text{extend}} \\
D_{0,j} &= s_{\text{open}} + j s_{\text{extend}} \\
S_{i,0} &= I_{i,0} \\
S_{0,j} &= D_{0,j} \\
I_{i,j} &= \max \begin{cases} I_{i,j-1} + s_{\text{extend}} \\ S_{i,j-1} + s_{\text{open}} + s_{\text{extend}} \end{cases} \\
D_{i,j} &= \max \begin{cases} D_{i-1,j} + s_{\text{extend}} \\ S_{i-1,j} + s_{\text{open}} + s_{\text{extend}} \end{cases} \\
S_{i,j} &= \max \begin{cases} I_{i,j} \\ D_{i,j} \\ S_{i-1,j-1} + s_{X_i Y_j} \end{cases}
\end{aligned} \tag{1.1}$$

(a)

```
sequence 1:   ...GAAAACTCTGGTAAATCTTGAGGTGAAG-----GGGAGGCAC...
sequence 2:   ...GAAAAC----------CTTGAGGCAAAGATGGAGGGGGGCAC...
```

(b)

	A	C	G	T
A	91	-114	-31	-123
C	-114	100	-125	-31
G	-31	-125	100	-114
T	-123	-31	-114	91

open -400
extend -30

(c)

Features:	**AA**:8	**CC**:4	**GG**:11	**TT**:2	**AG**:1	**GA**:1	**TC**:1	O:2	E:15
Scores:	728	400	1100	182	-31	-31	-31	-800	-450
Total:	1067								

Figure 1.1 Alignment and scoring example. (a) An alignment of the DNA sequences **GAAAACTCTGGTAAATCTTGAGGTGAAGGGGAGGCAC** and **GAAAACCTTGAGGCAAAGATGGAGGGGGGCAC**. (b) Scoring parameters. The matrix entry at row x column y is the score for aligning an x in the first sequence with a y in the second. (c) Score calculation. The features from the alignment are counted and multiplied by the corresponding score.

A simple dynamic programming algorithm follows directly from recurrence (1.1), with $O(|X|\ |Y|)$ complexity in time and memory. Keeping traceback information for each i,j allows the identification of an optimal alignment, with no increase in complexity. Excluding negative values of $S_{i,j}$, as in (1.2), allows us to identify similar *sub*sequences.

$$\begin{aligned}
I_{i,0} &= s_{\text{open}} + i s_{\text{extend}} \\
D_{0,j} &= s_{\text{open}} + j s_{\text{extend}} \\
S_{i,0} &= S_{0,j} = 0 \\
I_{i,j} &= \max \begin{cases} I_{i,j-1} + s_{\text{extend}} \\ S_{i,j-1} + s_{\text{open}} + s_{\text{extend}} \end{cases} \\
D_{i,j} &= \max \begin{cases} D_{i-1,j} + s_{\text{extend}} \\ S_{i-1,j} + s_{\text{open}} + s_{\text{extend}} \end{cases} \\
S_{i,j} &= \max \begin{cases} I_{i,j} \\ D_{i,j} \\ S_{i-1,j-1} + s_{X_i Y_j} \\ 0 \end{cases}
\end{aligned} \tag{1.2}$$

Memory complexity can be reduced to O(|X|) by scanning row by row and only keeping the latest value computed in each column, along with one extra ‘lag’ variable to keep track of the diagonal cell from the previous row. Time is still quadratic, however, and for large problems heuristic methods are necessary to reduce time requirements.

Rather than compute values for the entire dynamic programming matrix (DP matrix), heuristics are used to reduce the search to small regions of the matrix where the highest similarities are more likely. While algorithms based on recurrences (1.1) and (1.2) are guaranteed to find optimal alignments, they are not feasible for large sequences due to the quadratic time complexity. Alignment programs must use heuristic methods to guide the search to the small fraction of the DP matrix that contains high scoring alignments. Heuristics incur a loss of sensitivity, thus sensitivity becomes an important consideration in choice of heuristics.

The anchored alignments that will be discussed in section 2.3 are found using a variation of recurrence (1.1). The alignment is required to include a fixed starting point, but for the other endpoint we choose the maximum score anywhere in the DP matrix. The portion of the DP matrix actually computed is also reduced.

1.3 Quantum Alignment

In many problems the sequences being aligned contain a certain amount of uncertainty at each position. For example, suppose we have the present day DNA sequence of the same gene in several species. Though the ancestral species no longer exists, the ancestral sequence of this gene can be inferred from the present day sequence. One can establish the probability that each position in the ancestral sequence contained a particular nucleotide.

One solution is to discard the probabilities after inference and project the ancestral sequence to the most likely bases. Instead, we permit sequences that represent a base as a probability distribution over the symbols A, C, G, and T. To simplify discussion, we have coined the term *quantum nucleotide* (shorthand *q-DNA*) for such a distribution, in analogy with the field of quantum mechanics[2], and *quantum sequence* (shorthand *q-sequence*) for a sequence composed of q-DNA[3].

With an appropriate scoring scheme, quantum sequences can be aligned to DNA sequences or to other quantum sequences. Recurrence (1.1) is still applicable. However, the heuristic methods used for large problems often take advantage of the small alphabet of DNA, and must be altered to accommodate the infinite quantum alphabet. A truly infinite alphabet presents additional challenges; for the sake of efficiency we project quantum bases onto a finite alphabet that allows us to implement s_{xy} of recurrence (1.1) by a table lookup. This introduces an additional problem of how best to choose the alphabet.

1.4 Related Work

Sequence alignment has been studied since the 1960s, originally motivated by document comparison and text queries, with eventual adoption for comparison of protein and DNA sequences. Early seminal work is due to Levenstein (1966), Needleman and

[2] In quantum mechanics, measurable properties such as energy or position (whether continuous or discrete) are represented by probability distributions rather than definite values.

[3] The author realizes that there are many names in the literature for the same concept, among them profiles, weighted sequences, and probabilistic DNA. But no term has won exclusive use, and many suffer from easy confusion with other concepts.

Wunsch (1970), Smith and Waterman (1981) and Gotoh (1982). A recent survey of the state of the field is provided by Batzoglou (2005).

Alignment of probabilistic DNA sequences has recently become a topic of interest in the field. Hudek (2005) aligns sequences of ambiguous DNA inferred from multiple alignments, but discards probabilities from the sequences. Flannick and Batzoglou (2005) reduce a multiple alignment to a sequence of probabilistic profiles over {A,C,G,T}, but in contrast to our research also include a probabilistic gap at each position, and align to the sequence of most-probable bases rather than the probabilistic sequence. In the MAVID multiple aligner, Bray and Pachter (2004) infer probabilistic sequences similar to ours, but reduce them to sequences of most-probable bases prior to alignment.

Chapter 2

BLASTZ-type Aligners

BLASTZ (Schwartz et al. 2003) is a pairwise DNA sequence aligner originally patterned after Gapped Blast (Altschul et al. 1997; Zhang et al. 1998). Initially designed as a piece of the PipMaker server (Schwartz et al. 2000), it has received widespread use in the scientific community, serving, for example, as the first stage in generating whole genome alignments for the UCSC Genome Browser (http://genome.ucsc.edu). A major contribution of BLASTZ was a reduction in memory requirements, allowing sequences of a few million base pairs to be aligned. As longer sequences have become more prevalent, BLASTZ has again reached the point of being constrained by memory.

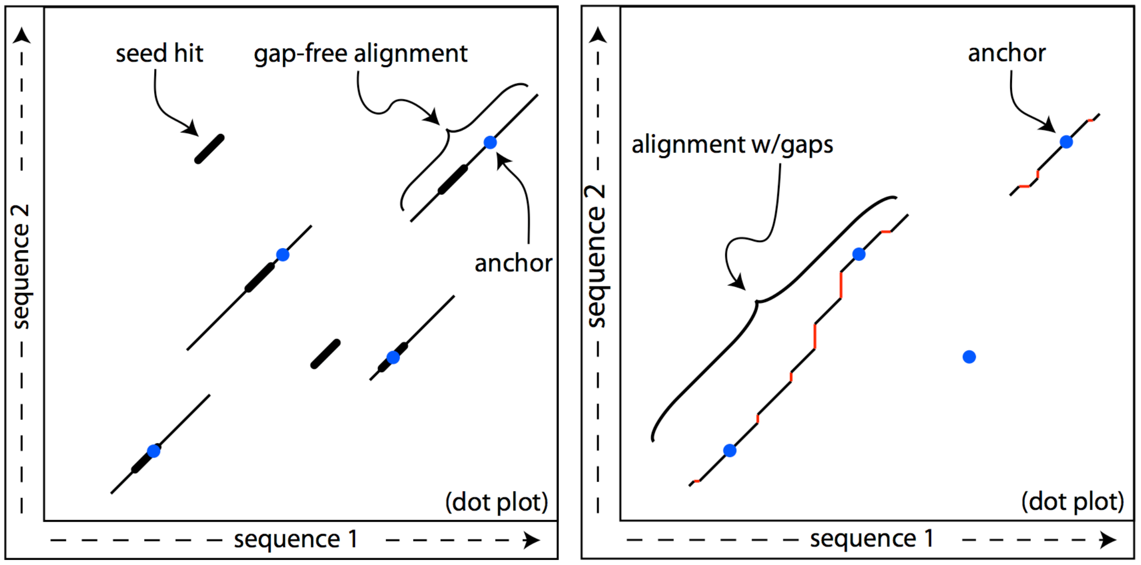
What follows is a simplified presentation of the program. The actual program has many parametric choices, which we will discuss in later sections. BLASTZ is optimized to preprocess one sequence (which we call *sequence 1*) and then align several queries to it (we will use *sequence 2* or *query* interchangeably). The algorithm consists primarily of the following stages: seeding, gap-free extension, chaining, anchoring, gapped extension and interpolation. Both BLASTZ and LASTZ include a few other features, such as dynamic masking, which will not be discussed in this thesis.

2.1 Seeding and Gap-free Extension

The seeding stage identifies short near-matches (seed hits) between sequence 1 and 2. In general, a seed pattern/rule determines what constitutes a near-match of some length L, but for this discussion it is enough to think of a seed hit as a perfect match of two L-mers. More sophisticated seeding strategies are used in practice, discussed in Chapter 3 and section 4.1.

A preprocessing pass parses sequence 1 into overlapping *seed words* of length L. Each word is converted to a value, called the *packed seed word* (usually requiring fewer bits than the seed word) according to the seed pattern (discussed in more detail in Chapter 3). These (seed, position) pairs are collected in a table. Conceptually, the table is a mapping from a packed seed value to a list of the sequence 1 positions where that seed occurs. The *seed word position table* is one the major space requirements of the program, and we discuss design choices in section 4.2. Both time and memory required for seeding can be decreased by using *sparse spacing*. Instead of storing a seed word for every position, positions are stored only for multiples of z (the *z-step*). Large values of z (e.g. z=100) incur a loss of sensitivity, at least at the level of seed hits. However, to discover any gapped alignment we only need to discover one seed hit in that alignment (of many), so the actual sensitivity loss is small in most cases. Section 6.2 discusses the effect of z-step on the end result.

To locate seed hits, the query sequence is then similarly parsed. Each query seed is used as an index into the position table to find the sequence 1 positions that 'hit' that



(a) (b)

Figure 2.1 Alignment stages. (a) Seed hits and HSPs. Heavy lines are seed hits, short gap-free near-matches. Seed hits are extended to create HSPs (thin lines). Seed hits with no HSP had low scoring extensions. (b) Anchors and gapped alignment. Anchors (blue dots) are single points in highest scoring window of each HSP. Anchors are extended to form gapped alignments (gaps in red). Anchor shown without alignment had low scoring extensions which were discarded.

seed. As each seed hit is found, it is extended without allowing gaps to determine whether it is part of a *high-scoring segment pair* (HSP). The hit is extended along the diagonal[4] in both directions, using the score values s_{xy} to accumulate the score of the extended match. In each direction, the extension is stopped whenever a segment with a large negative score is encountered (negative of the 'x-drop' threshold). These negative scoring ends are then trimmed. If the resulting score meets the *ungapped alignment score threshold* (K) it is an HSP and is kept for further processing. Matches at do not meet the score threshold are discarded. An additional filtering step eliminates hits with low entropy. An example of this process is shown in figure 2.1(a).

Usually an HSP will contain several seed hits. Extending each of these hits would result in the same HSP several times. This is prevented by rejecting seed hits that overlap previous extensions (even extensions that failed to produce an HSP). Hits along any diagonal are processed in increasing order (see the discussion of the seed word position table in section 4.2). Thus we only need to keep track of how far we have progressed

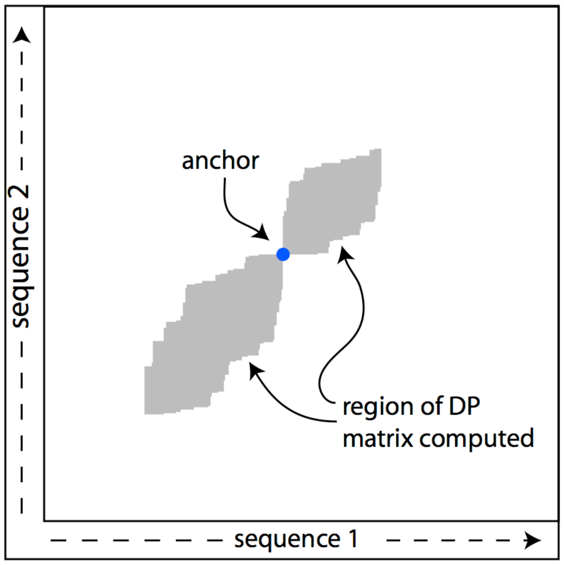


Figure 2.2 Y-drop alignment region. The boundaries of the region are points scoring much lower than the possible maximum.

[4] A *diagonal* is a set of DP cells (i,j) that have a constant difference i-j. The diagonal is often referred to by this difference.

along each diagonal; if a new hit occurs to the left of progress on its diagonal, we can quickly discard it. The storage of this *diagonal extent table* is also a major space consideration, and will be discussed further in section 4.2.

2.2 Chaining

The chaining stage finds the highest scoring series of HSPs in which each HSP begins strictly before the start of the next. All HSPs not on this chain are discarded. This is useful when elements are known to be in the same relative order in the query as in sequence 1. Briefly, the chaining algorithm is an example of sparse dynamic programming. It processes the HSPs in order along sequence 1, building chains by adding the next HSP to the best previous viable chain.

2.3 Anchoring and Gapped Extension

Every remaining HSP is reduced to a single point to be used as an anchor for gapped alignment. A constant-width window is slid across the HSP and the midpoint of the highest-scoring window is chosen as the anchor.

The anchors are then processed in order of the score of their HSP (highest score first). One-sided extension is performed in both directions from the anchor point, the two resulting alignments are joined at the anchor, and if the score meets the *gapped alignment score threshold* (L) it becomes an alignment in the output file. One-sided extension is computed per recurrence (1.1), beginning at the anchor and ending at the highest scoring point. The portion of the DP matrix considered is reduced by disallowing low-scoring segments (Zhang et al., 1998); wherever the score drops further below the known least possible maximum than the *y-drop* threshold, the DP matrix is truncated and no further cells are computed along that row. An example of the resulting search range is shown in figure 2.2, while figure 2.1(b) shows an example of the overall gapped alignment results.

2.4 Interpolation

Once gapped extension has been performed, it is not uncommon to have regions leftover in which no alignment has been found. In the interpolation stage we repeat all previous stages, in these leftover regions, at a higher sensitivity. For example we could use a lower weight seed or a lower scoring threshold. Using such high sensitivity from the outset would be computationally prohibitive, but is feasible on the smaller, leftover regions.

Chapter 3

Spaced Seeds

As noted earlier a common heuristic is to focus alignment search in the vicinity of seed hits, short matches or near matches. The seed hit is evidence of a larger similarity between the sequences. Early aligners commonly required exact matches for seed hits. However, allowing some mismatches in seed hits can increase sensitivity with no loss in specificity (Ma et al. 2002).

To understand how sensitivity is improved, consider the following example. Suppose we have a 20 bp homologous sequence that has undergone substitution mutations, but no insertions or deletions, to the extent that each base has only a 70% chance of matching in the present day sequences ($p_{\text{match}} = 70\%$). If we require an exact match of 5 consecutive bases for a seed hit, then we must have such a match somewhere among the 20 bp or we will fail to discover this homology. The chance of having at least one 5-bp match is only 73%.

Now suppose instead that our seed hit requires 5 matching bp, but allows one mismatch between the 3rd and 4th match. We describe this seed pattern as `111011`, with the `1`s representing required *match-positions* and the `0` representing a *don't-care position*. This pattern has the same specificity as the exact match (the expected number of hits is same), but the chance of it occurring at least once among the 20 bp is 80%. The weakness of the first seed is due to the fact that, as we advance along the sequence with the first seed, any mismatch knocks us all the way back to the beginning of our pattern, 5 steps from success. With the second seed, a mismatch after 3 or 4 matches simply bumps us back a couple steps; we're still only two steps away from success.

We call a seed pattern containing don't-care positions a *spaced* seed[5]. The number of positions in the seed is called its *length* L, and the number of required matches is its *weight* W; we call such as seed a *W-of-L* seed. Any spaced seed will have higher

[5] In the literature, the term "seed" is often used interchangeably for a seed pattern, seed word or seed hit.

sensitivity than an exact match seed of the same weight, provided the homology is long enough and/or p_{match} is high enough. However, for a given homologous length h, the weight-W exact match can ‘hit’ in h+1-W positions, while the (L,W) spaced seed can only hit in h+1-L. For h shorter than some cutoff, the exact match is more sensitive. This makes ‘light’ seeds, with a large proportion of spaces, less useful.

The sensitivity of a seed has a complicated relationship to its pattern. Two similar seeds, having the same length and weight but disagreeing only in the position of one don’t-care position, can have much different sensitivity. Further, sensitivity depends on the evolutionary substitution rate of the sequences. One seed may be better for a 95% rate while another may be better for 70%.

A seed’s sensitivity *can* be computed by transforming its pattern into a deterministic finite automaton that accepts strings containing that pattern, then computing the probability that a random string (according to some evolutionary model) will be accepted (Buhler et al., 2003). Some seed(s) will be optimal, having the highest possible

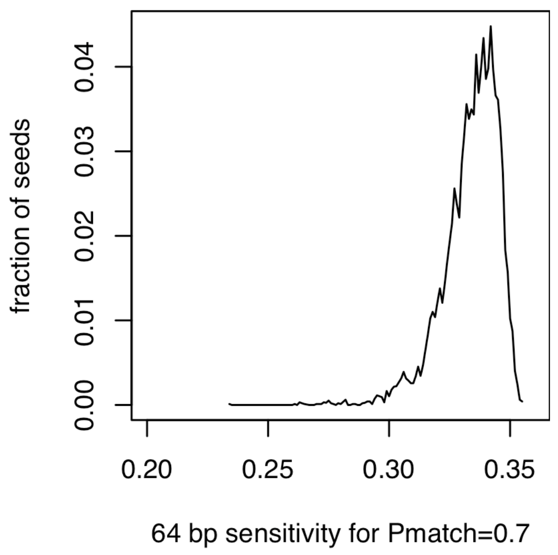


Figure 3.1 Seed sensitivity distribution. Distribution of the probability of discovering a 64 bp homology with 70% identity for all 12-of-19 seeds. The median seed has sensitivity at least 94% of the optimal (0.355), and the 90th percentile is at least 97% as sensitive.

sensitivity for a particular model; Buhler has studied optimality extensively under a variety of evolutionary models.

Computation of a single seed's sensitivity appears to be exponential in the number of don't-care positions, and the number of patterns grows exponentially with the length. Finding optimal seeds by exhaustive evaluation is computationally impractical, and the result is a seed that is only known to be optimal for a particular evolutionary model. But many seeds are close to optimal (see figure 3.1). So the more computationally efficient strategy of trying several random seeds and picking the best may be adequate for most uses.

BLASTZ makes use of a specific 12-of-19 seed based on seed shown to be optimal for 64 bp homologies with 70% identity (Ma et al., 2002; Schwartz et al., 2003). The user may also choose a 14 of 22 seed or an exact match seed of any length. Both seeds also allow a transition mismatch in any one of the seed's match positions, increasing sensitivity.

We discuss additional seeding strategies in section 4.1.

Chapter 4

LASTZ

With the rapid acceleration of sequencing technologies, the effort required to tune BLASTZ for each new genome sequence has become more of a nuisance. Further, choosing appropriate tuning parameters still seems more art than science, and the necessary insight is in the hands (or minds) of a few. A major goal of LASTZ is to change that, to have the program derive suitable tuning parameters from the sequences themselves. Toward that end, LASTZ acts as a tool bench for experimenting with alignment strategies.

LASTZ supports all the capabilities of BLASTZ, but extends them, providing a richer variety of seeding strategies, reducing memory requirements and aligning quantum sequences.

4.1 Seeding Strategies

The run time of BLASTZ-type aligners is greatly affected by seed specificity. Low specificity equates to a high number of false seed hits, requiring extra computation during gap-free extension. Further, the number of false seed hits scoring high enough to become HSPs increases the run time during gapped extension. Many seeding strategies have been proposed in the literature, and LASTZ adds support for user-specified seed patterns, transition-match positions, half-weight seed patterns, double transitions, and twin hit seeds.

User-specified seed patterns. To facilitate experimentation with seed patterns, LASTZ allows the user to directly specify the pattern as a string of 1 (match), 0 (don't-care) and T (transition-match) positions.

Transition-match positions. A *transition-match* or *T-position* allows a match or transition mismatch, but not a transversion[6]. Transition-matches were introduced in the literature by Sun and Buhler (2006)[7] and were further studied by Zhou and Florea (2007). Because of its effect on specificity, a T-position contributes the value of 1/2 position to seed weight.

Half-weight seed patterns. LASTZ adapts the idea of Hou et al. (2007), allowing patterns restricted to transition-match and don't-care positions. An additional filtering stage is added between seed hits and gap-free extension. The number of matches over the length of the seed word must meet a specified lower bound, and the number of transversions must obey a specified upper bound. Note that hits for a half-weight seed without don't-care positions will not contain any transversions.

The main advantage of a half-weight seed is that it allows a longer seed length without using additional memory (see Appendix E). Further, it allows more general mismatch cases than simple spaced seeds. For example, the 12-of-19 seed 1110100110010101111 allows up to 7 mismatches but only in specific positions. The 19-of-19 half-weight seed can be specified to allow 7 mismatches in the seed word, regardless of position, and uses less memory than the 12-of-19 seed.

The mathematical properties of half-weight seeds have yet to be fully explored. It appears that a length 2L half-weight seed, with no spaces, should be more sensitive than a length L exact match seed, and this is supported by some experimental results in Hou et al. (2007). Half-weight seeds were implemented to support those experiments, but have not been investigated further. Ideally, we would like to be able to show that a spaced half-weight seed is more sensitive than the equivalent seed with 1-positions replacing the Ts.

The name *half-weight* comes from the fact that, since the seed contains only T-positions and don't-cares, its weight is half that of the equivalent seed with 1s instead of Ts.

---

[6] A T-position should not be confused with the allowance of a single transition in a match-position. The former allows any number of transitions, but only in specific positions. The latter allows only one, but in any match-position.

[7] The author's own implementation of T-positions dates to late 2004, the same time frame in which Sun and Buhler first submitted their 2006 paper.

Double transitions. LASTZ can allow up to two transitions among the match-positions in a seed. The computational cost is a factor of about W/2 when scanning the query during seeding.

Twin hit seeds. For sequences with high similarity, BLASTZ seeding strategies are too sensitive, resulting in too many seed hits. LASTZ allows the requirement of two nearby hits on the same diagonal before gap-free extension is performed. The user can specify the range of the gap length between the seeds (the gap may also be negative, indicating overlapping seed hits).

Section 6.3 describes experimental results for twin hit seeds.

Floating-point scoring. A separate build of LASTZ treats scores as floating-point values. This was useful in studying iterative scoring inference, without having to worry about truncation effects.

4.2 Memory Requirements

One of the author's goals for LASTZ was to reduce memory requirements to increase the size of sequences, and the weight of seeds, that could be used within the memory constraints of a desktop workstation with 1 GB memory. The primary components with potential for consuming memory are the two sequences, the seed word position table, the diagonal extent table, the dynamic programming array, and traceback.

For convenience in this discussion, we define the following terms, and consider a *large chromosome* to be 250M bases.

L_1,L_2	= length of sequence 1 and 2.
W	= seed weight
Z	= z-step
V	= fraction of 'viable' seed words in sequence 1.

A seed word is *viable* if it contains no masked or ambiguous bases[8]. It is not uncommon for V to be around 50%. For example, in the 44 ENCODE regions (ENCODE

[8] Genomic DNA contains many repeat elements—segments that have been replicated at some point in the past. A common practice in the preparation of genomic DNA prior to wide scale alignment is to identify repeats and "mask" them by using lowercase a, c, g and t. In addition, sequences often contain bases for

Project Consortium 2004), V ranges from 38 to 63% in human, with only one region below 45%.

Sequences. (memory required: $2L_1+2\max(L_2)$). Both sequences are stored internally at one byte per base. Of multiple queries, only the current query is resident. A second copy of each sequence is stored, giving the sequence in reverse order to save gapped alignment the concern of directionality. For two large chromosomes, these total 1G. There is opportunity for savings here, at the expense of speed. First, the reversed sequence copies could be eliminated by having four different copies of the gapped alignment routine (one for each combination of sequence direction). This would require no run-time cost but would incur a cost in source code maintenance. Second, DNA sequences could be packed four bases per byte after the seeding stage. While building the position table, we need access to masking information in the sequence but if sequence input parsing was more tightly coupled with position table building this could be accomplished with a small constant sized buffer, building the position table and packed sequence at the same time. Similarly, when scanning sequence 2 for seed hits, we also require masking information, but could build the packed sequence during the seed hit scan.

Both sequences are needed for gap-free and gapped extension, and having them in packed form would slow down both processes. Additionally, sequence 2 may require 8 bits per symbol if it is a quantum sequence. For these reasons, we chose not to pack the sequences.

However, for overweight seeds (section 4.3) we do construct a packed version of sequence 1, requiring an additional $L_1/4$ bytes. This is another tradeoff of memory for speed. Having the packed version resident saves us the time of repacking each seed word to resolve a seed hit.

Seed word position table. $(4(4^W+(L_1/Z))$. The position table must provide a mapping from a seed word to a list of the positions where that word occurs in sequence 1. For large seeds and/or long sequences, the table can be very large. Several schemes were

hard-to-sequence regions where the actual nucleotides are not known. These ambiguous bases are represented by N. LASTZ does not allow masked or ambiguous bases as part of seed hits.

considered, but in the end we chose between two options, which we call *linked list* and *last/previous*. Last/previous is the scheme used in LASTZ, but we will describe both here.

The linked list scheme is shown in figure 4.1. A pointer table indexed by seed word contains pointers to linked lists. Each list element contains an index into sequence 1 and a pointer to the next element. Elements are allocated on an as-needed basis (in blocks[9]) and only seed words positioned at multiple of z are stored, so only VL_1/Z list elements are needed. On a 32-bit machine, the pointer table requires 4×4^W and the list elements $8VL_1/Z$. On a 64-bit machine, pointers are 8 bytes. List elements could conceivably be stored in 12 bytes, but many compilers (including the widely used gcc) store them in 16 bytes. So the requirements double, to $8 \times 4^W + 16VL_1/Z$.

The last/previous scheme is shown in figure 4.2. A position table (`Last`) indexed by seed word contains the index into sequence 1 of the last position containing that seed word. A second table (`Previous`), indexed by sequence position, provides the position of the 'previous' occurrence of the same seed word[10]. A zero is used to terminate the chain[11]. Since only seed words positioned at multiples of z are stored, the table contains space only for those positions, and contains L_1/Z entries. Since the tables store sequence positions instead of pointers, memory requirements are the same regardless of processor word size. `Last` requires 4×4^W and `Previous` $4VL_1/Z$.

For highly masked sequences the linked list version will use less memory. On a 32-bit machine, the break-even point is V=50%, which is a typical value for human sequences. On a 64-bit machine the linked list version will use less memory only when $V <= (1/4) - (4^{\wedge W-1})(Z/L_1) <= 25\%$.

Unfortunately, neither method has good cache behavior. As we parse each sequence into seed words, the accesses to the pointer table, or to `Last`, have no locality.

---

[9] The overhead (both memory and time) for allocating list elements one-at-a-time can be extremely detrimental to performance.

[10] More formally, `previous`[i] = max j such that seedword(sequence1[j]) = seedword(sequence1[i]).

[11] Internally, we actually store the position of the *end* of the seed word rather than the start, so zero is never used for a valid seed word.

Nor do accesses to `Previous` exhibit locality. For the linked list scheme, a post processing step on the list elements, following construction of the table, could organize them to be more cache friendly while processing the second sequence.

A useful side effect of both schemes is that, since the query sequence is scanned in increasing order, hits along any diagonal are also processed in increasing order. This facilitates the implementation of the diagonal extent table.

Other data structures for locating seed hits have been presented in the literature. Gusfield (1997) applies a suffix tree to the problem of finding exact substring matches between two strings. While a suffix tree leads to an O(n) solution for exact matches, the space requirements can be daunting. Typical implementations require 20 bytes per sequence character, according to Lippert (2005)[12]. The Burrows-Wheeler transform (Burrows and Wheeler, 1994) can reduce space to 2.5 *bits* per sequence character (Lippert, 2005), and provide exact matches in O(n log n) time. Neither structure appears suitable for finding all partial matches similar to spaced seeds. While Gusfield (1997) presents O(n) solutions to several partial match problems, these all involve finding only a single partial match. Lippert (2005) gives a partial match method, but it involves searching for $O(2^k)$ matches, where k is the number of mismatches allowed.

Cameron (2006) stores the equivalent of the seed word position table in a deterministic finite automaton (DFA). The DFA has space comparable to our implementation but has much better cache behavior. However, the DFA concept does not work well for spaced seeds. It takes advantage of the fact that number of different 'next' seed words as we scan the sequence is small. For exact matches the number of possible next words is only four. For spaced seeds it is much higher; as we move to the next position, many positions that were spaces in the previous seed word are now filled with characters, and each of these has four possibilities. Z-steps aren't DFA friendly, for similar reasons.

Diagonal extent table. (fixed 0.75M). As part of the gap-free extension stage (section 2.3) we maintain an array of the extent of expansion along each diagonal. This

[12] Suffix tree space is O(n log n), so Lippert's claim of 20 bytes per character, without mention of a specific sequence length, is curious.

allows us to quickly filter out seed hits that have been covered by the expansion of a previous hit. If twin hits are required, a second array gives the start position of the latest series of nearby hits on each diagonal. A third array is used to resolve hash collisions, as will be explained shortly.

A direct implementation of this data structure (for which collision resolution would be unnecessary) would require $8(L_1+L_2)$ bytes, or 4Gbytes for large chromosomes[13]. Instead, we hash diagonals to 16-bit values[14]. This reduces memory requirements to 12×2^{16} = 0.75Mbytes (after we add the third array to detect hash collisions).

Experimental results regarding the rate of hash collisions and their effect on resulting alignments are given in section 6.1.

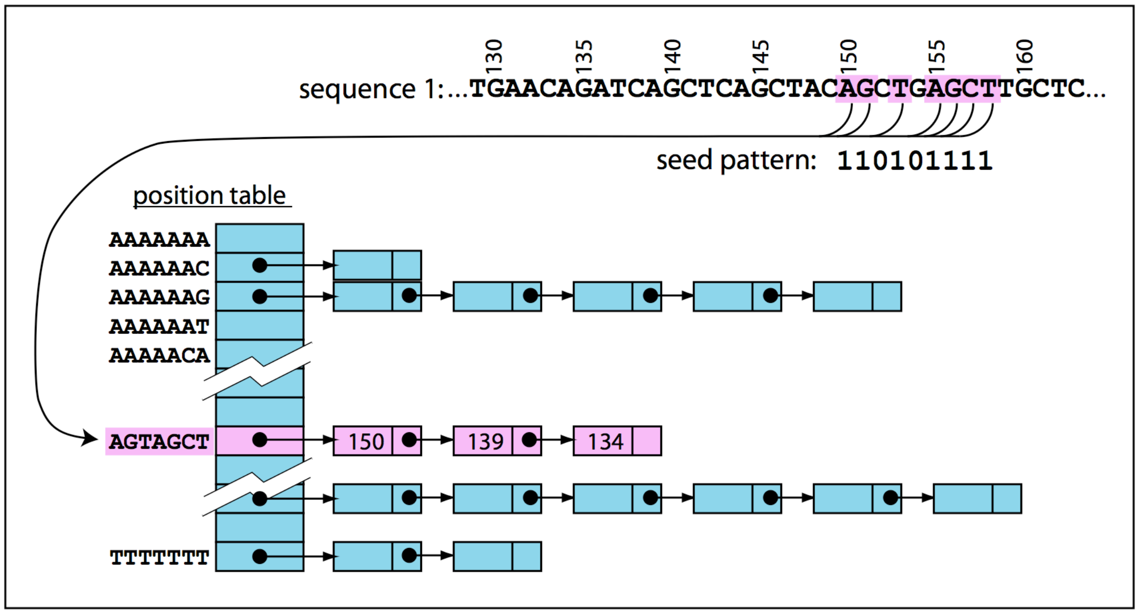


Figure 4.1 Seed-word position table, linked list implementation. Packed seed word selects linked list containing positions of matches in sequence 1.

[13] If twin seeds are sacrificed, only 2G bytes would be required, still well above our target budget.

[14] The hash function simply uses the least significant bits of the diagonal.

Dynamic programming array and traceback memory (80M). The DP array memory used during gapped extension is greatly reduced, compared to a straightforward implementation of (1.1), by constraining to the y-drop region. Further, scores are only saved for the current row being processed, so the number of DP cells needed is the longest row slice through some y-drop region. Memory is allocated on an as-needed basis, in increments of about 16K. This is not a significant memory consumer because, for default settings, the longest row slice is typically ≈ 2,000 cells, or about 32K.

A larger concern is traceback memory. In order to reconstruct an optimal alignment, we need to store traceback information over the entire y-drop region for a one-sided gapped extension. These are stored as one byte per cell. For default settings, the average y-drop region is less than 1 million cells, and the largest less than 25 million. The user can specify the amount allocated. By default 80M bytes are allocated, which will cover 80 million cells. A traceback memory shortfall causes truncation of a gapped

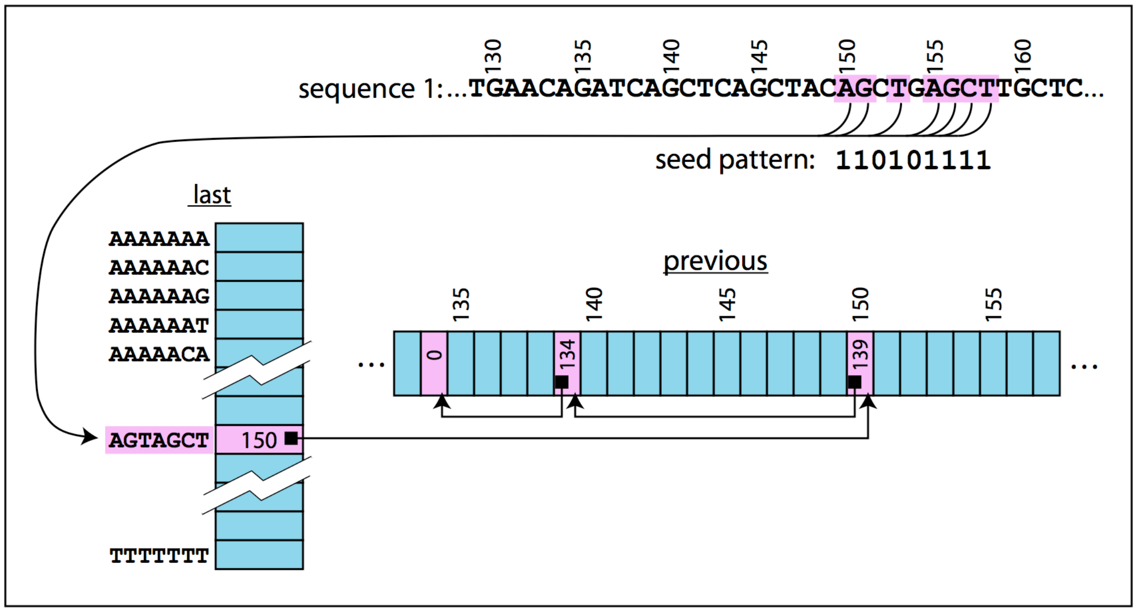


Figure 4.2 Seed-word position table, last/previous implementation. Packed seed word selects last position of a match in sequence 1. That position indexes position of previous match, additional matches are found by following the series of positions.

extension. This happens only when sequences are very similar, in which case it is likely that another anchor will allow us to pick up the rest of that alignment.

Additionally, we maintain an array of indexes to where each row begins (in the traceback array). Indexes into the traceback array are used rather than pointers to save on 64-bit machines. The index array is allocated as needed, expanding in leaps of 512K (128 thousand rows). For default settings, fewer than 20,000 rows are usually required.

4.3 Overweight Seeds

Memory requirements for the seed word position table depend on two factors, the number of positions stored in the table and the weight of the seed pattern. An increase of 1 in seed weight quadruples both the number of possible packed seed words and the bytes needed for the table indexed by them. On a machine with 1G of memory, a practical weight limit is 13. A weight-14 seed requires 1G just for this table, leaving no resident memory for anything else. Even on a machine with 8G, a weight-15 seed uses half the resident memory.

As another tradeoff of memory for time, LASTZ allows the user to specify a limit W_{max} on the portion of the seed used to index the table. Seeds heavier than that limit are considered *overweight*, and require additional processing to resolve seed hits. In effect, the seed packing function becomes a hash function. The 2L bits of the seed word are packed to $2W_{max}$ bits instead of 2W. Matches in the seed word position table are no longer guaranteed to be seed hits. Each hit in the seed word position table is then resolved by comparing the seed word in sequence1 with that in sequence 2. To facilitate this comparison, we construct a packed version of sequence 1.

4.4 Quantum Sequence Support

As described in section 1.3, it is often desirable to deal with DNA sequences that contain uncertainty, which we call quantum sequences. LASTZ can align a quantum query sequence to a DNA sequence. Externally, the quantum sequence must be reduced to a finite alphabet Σ_Q (maximum 255 symbols) and an appropriate $|\Sigma_Q|$x4 substitution scoring matrix must be provided. Techniques for performing the reduction and creating the scoring matrix are discussed in Appendix C.

The quantum sequence and scoring matrix comprises everything LASTZ needs to know about the sequence. Specifically, it does not know anything about the probabilities represented by any symbol[15]. The standard alignment recurrence (1.1) is unaffected by the presence of quantum DNA on one side of the equation. All that is required is a scheme that provides scores for substitution and gaps. Further, any algorithm that computes alignment scores based on that recurrence will still work for quantum alignment. Thus LASTZ is able to use the same algorithms that it uses for gap-free extension, chaining, anchoring, gapped extension and interpolation when aligning DNA. However, the seeding technique used in DNA alignment takes advantage of the small 4-character alphabet, creating a table indexed by 4^W seed words. A table of all $|\Sigma_Q|^W$ seed words would be prohibitive. We can still make use of a table of words from the DNA sequence, but we won't have words in the quantum sequence that directly match them.

For quantum sequences, LASTZ modifies the anchor finding stage as follows. It builds the usual table of seed words for (DNA) sequence 1. The (quantum) query sequence is parsed into overlapping q-words of length L. Each q-word is collapsed to W symbols (removing spaces if a spaced seed is being used). For each q-word it generates the ball of DNA words that score above some threshold (irrespective of whether the words exist in the DNA sequence). Then it looks up the locations of those words in the DNA sequence and proceeds as for DNA-to-DNA alignment. The ball generation algorithm is described in C.4.

[15] The mapping from symbol to probability can be provided, to enhance certain output formats.

Chapter 5

INFERZ

Given two long sequences of DNA, how can we assign 'appropriate' scores for matches and penalties for substitutions, gap open and gap extend? To answer this, we must have a model of the evolutionary process that mutated a common ancestor into the two sequences, and we must have a means of evaluating the inferred scores. We discuss the model in the rest of this chapter and from it derive a method for inferring scores. This method is encapsulated in the INFERZ program.

For evaluation, we treat the aligner as a classifier, classifying aligned bases, and use statistics based on the receiver operating characteristic (ROC, section B.5). ROC in turn requires values for true (TP) and false positive (FP) rates. For alignments of real data the correct alignment is rarely (if ever) known, so exact TP and FP rates aren't known. We addressed this is two ways. First, we test our inference method on simulated data for which the correct alignments are known. Second, for real data we use an estimate of FP derived from the alignments themselves, discussed in section 5.6.

5.1 Finite State Automaton for Neutral DNA

As per Durbin et al. (1998) we model a homologous sequence pair of neutral DNA by a three state Finite State Automaton (FSA) shown in figure 5.1(a). The FSA emits homologous pairs of bases in state H, and gaps in states I_X and I_Y. It is equivalent to an evolutionary model in which a common ancestral sequence evolved into a pair of sequences by means of independent mutations. Substitutions occur according to p_{xy}. Insertions occur with the same probability p_{open} at any position in either sequence, and do not overlap. The insertion process can stop independently at each base, continuing with probability p_{extend}, resulting in a geometric distribution of lengths. Deletions are treated as insertions in the other sequence. We address the validity of this model in section 5.3.

The model relates directly to the affine-gap alignment algorithm. Given observations from alignments we can infer alignment scores under this model. Inferring directly from the model, we can use log-odds[16] scores from each observed probability. We first consider the score of a length n gap and ignore base content, reducing the FSA to the gap-only model in figure 5.1(b). Since the model requires every gap to be followed by at least one step in H, we also include that step. The score for this elongated gap, consisting of n steps in I and one in H, should be

$$\begin{aligned} s_{\mathrm{nI+H}} &= log\left(p_{\mathrm{open}}(p_{\mathrm{extend}})^{n-1}(1-p_{\mathrm{extend}})\right) \\ &= log\ p_{\mathrm{open}} + (n-1) log\ p_{\mathrm{extend}} + log\ (1-p_{\mathrm{extend}}) \end{aligned} \tag{5.1}$$

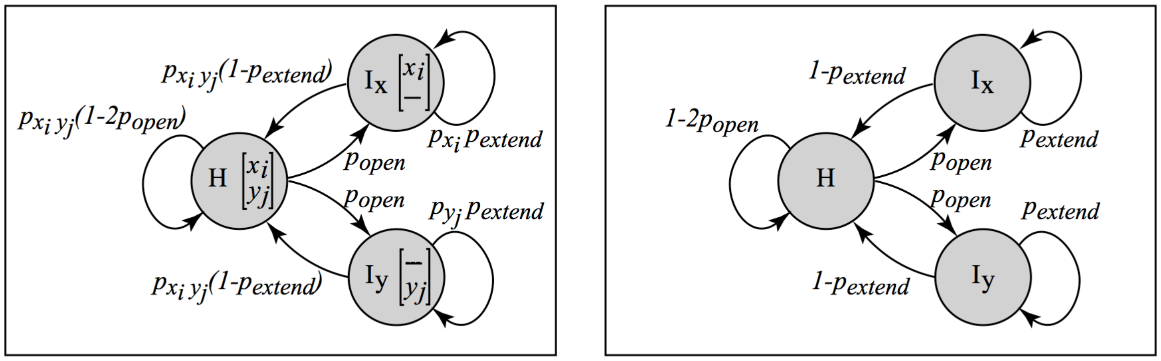
while the score for any other H event should be

$$s_{\mathrm{H}} = log\ (1-2p_{\mathrm{open}}) \tag{5.2}$$

For scores suitable for recurrence (1.1) we must have

$$s_{\mathrm{nI+H}} = s_{\mathrm{open}} + ns_{\mathrm{extend}} + s_{\mathrm{H}} \tag{5.3}$$

From this we see that recurrence (1.1) charges a penalty for one extra extend, and scores the gap-terminating event the same as any other H-event. Making appropriate adjustments to s_{open}, the proper inferred scores are



$p_{x_i y_j}(1\text{-}p_{extend})$
$p_{x_i y_j}(1\text{-}2p_{open})$
$I_x \begin{bmatrix} x_i \\ - \end{bmatrix}$
$H \begin{bmatrix} x_i \\ y_j \end{bmatrix}$
$I_y \begin{bmatrix} - \\ y_j \end{bmatrix}$
$p_{x_i} p_{extend}$
p_{open}
p_{open}
$p_{y_j} p_{extend}$
$p_{x_i y_j}(1\text{-}p_{extend})$
$1\text{-}p_{extend}$
$1\text{-}2p_{open}$
I_x
H
I_y
p_{extend}
p_{open}
p_{open}
p_{extend}
$1\text{-}p_{extend}$

(a) (b)

Figure 5.1 Pair finite state automaton. (a) Full model emitting pairs. (b) Simplified gap-only model.

[16] When we give specific log values we use base 2 logarithms.

$$\begin{aligned} s_{\text{open}} &= log\ p_{\text{open}} - log\ p_{\text{extend}} + log\ (1 - p_{\text{extend}}) - log\ (1 - 2p_{\text{open}}) \\ s_{\text{extend}} &= log\ p_{\text{extend}} \\ s_{\text{H}} &= log\ (1 - 2p_{\text{open}}) \end{aligned} \tag{5.4}$$

Incorporating base content, we include log odds scores for the pair emitted in the H event. Using the approach of Chiaromonte et al. (2002), we have

$$\begin{aligned} s_{\text{open}} &= log\ p_{\text{open}} - log\ p_{\text{extend}} + log\ (1 - p_{\text{extend}}) - log\ (1 - 2p_{\text{open}}) \\ s_{\text{extend}} &= log\ p_{\text{extend}} \\ s_{\text{xy}} &= log\left(\frac{p_{\text{xy}}}{p_{\text{x}\bullet}\ p_{\bullet\text{y}}}\right) + log\ (1 - 2p_{\text{open}}) \end{aligned} \tag{5.5}$$

Note that the formula for s_{xy} in (5.5) differs from the formula in Chiaromonte et al. by the addition of a score for remaining in state H. Since Chiaromonte et al. did not consider gaps, $p_{\text{open}} = 0$ in their model, $\log(1 - 2p_{\text{open}}) = 0$, and (5.5) is consistent with their result.

We could also incorporate base content in gap scoring. However, there is no accommodation for doing so in recurrence (1.1). Effectively we are modeling base distribution in gaps as uniform.

5.2 Inferring Scores From Alignments

Given a collection of alignments $\mathcal{A}$, we can estimate event probabilities and apply (5.5) to infer alignment scores. Defining these alignment statistics

$\mathcal{A}_{\#}$	= number of alignments.
$\mathcal{A}_{\text{H}}$	= number of ungapped columns (number of steps in FSA state H).
$\mathcal{A}_{\text{I}}$	= number of gapped columns (number of steps in FSA state I_X or I_Y).
$\mathcal{A}_{\text{gaps}}$	= number of gaps.
$\mathcal{A}_{\text{xy}}$	= number of ungapped columns with x for sequence 1 and y for sequence 2.
$\mathcal{A}_{\text{x}\bullet}$	= number of ungapped columns with x for sequence 1.
$\mathcal{A}_{\bullet\text{y}}$	= number of ungapped columns with y for sequence 2.

We can estimate the necessary quantities for (5.5) thus:

$$\begin{aligned}
\hat{p}_{\text{open}} &= \frac{1}{2}\left(\frac{\mathcal{A}_{\text{gaps}}}{\mathcal{A}_{\text{H}} - \mathcal{A}_{\#}}\right) \\
&\approx \frac{1}{2}\left(\frac{\mathcal{A}_{\text{gaps}}}{\mathcal{A}_{\text{H}} - \mathcal{A}_{\#}} + \frac{\mathcal{A}_{\#}}{\mathcal{A}_{\text{H}}}\left(1 - \frac{\mathcal{A}_{\text{gaps}}}{\mathcal{A}_{\text{H}} - \mathcal{A}_{\#}}\right)\right) \\
&= \frac{1}{2}\left(\frac{\mathcal{A}_{\#} + \mathcal{A}_{\text{gaps}}}{\mathcal{A}_{\text{H}}}\right) \\
&= \frac{1}{2}\left(\frac{1}{\text{average ungapped run}}\right)
\end{aligned} \tag{5.6}$$

$$\begin{aligned}
\hat{p}_{\text{extend}} &= \frac{\mathcal{A}_{\text{I}} - \mathcal{A}_{\text{gaps}}}{\mathcal{A}_{\text{I}}} \\
&= 1 - \frac{\mathcal{A}_{\text{gaps}}}{\mathcal{A}_{\text{I}}} \\
&= 1 - \frac{1}{\text{average gap length}}
\end{aligned} \tag{5.7}$$

$$\begin{aligned}
\hat{p}_{\text{x}\bullet} &= \frac{\mathcal{A}_{\text{x}\bullet} + \mathcal{A}_{\tilde{\text{x}}\bullet}}{\mathcal{A}_{\text{H}}} \\
\hat{p}_{\bullet\text{y}} &= \frac{\mathcal{A}_{\bullet\text{y}} + \mathcal{A}_{\bullet\tilde{\text{y}}}}{\mathcal{A}_{\text{H}}} \\
\hat{p}_{\text{xy}} &= \frac{\mathcal{A}_{\text{xy}} + \mathcal{A}_{\text{x}\tilde{\text{y}}} + \mathcal{A}_{\tilde{\text{x}}\text{y}} + \mathcal{A}_{\tilde{\text{x}}\tilde{\text{y}}}}{\mathcal{A}_{\text{H}}}
\end{aligned} \tag{5.8}$$

5.3 Empirical Agreement with the FSA Model

Under the simplified gap-only FSA, the distribution of gap lengths should be geometric. This is a shortcoming of the model, as there is much empirical evidence that suggests gap lengths *in vivo* follow a power law distribution (Zhang and Gerstein, 2003). Genome-wide observations of *in silico* alignments between human and six vertebrates (section B.1) show gap length distributions somewhere between power-law and geometric, as is apparent in figures 5.2(a) and (b). The underlying alignment algorithm used affine-gap scoring, and it appears it has pushed the observed distribution toward geometric.

Though this is a serious deficiency, affine-gap scoring is much faster to compute than more general schemes, and this is especially important for large-scale alignments. The deficiency says nothing about the algorithm's ability to locate homology on a broad scale, only that its ability to accurately place gaps is questionable. The program could be augmented by post-processing the discovered alignments using a more realistic gap model (Cartwright, 2006).

The model also predicts that ungapped run lengths (inter-gap distances) should be geometric. Figure 5.3 shows two disagreements with observed run lengths *in silico*. First, there is an abrupt drop-off in the number of very short runs (in dog, this occurs for runs shorter than 10 bp, for macaque 30 bp). This is an alignment artifact; recurrence (1.1) discourages nearby gaps and will find a higher scoring alternative. Second, macaque has a greater number of short runs (other than very short), suggesting the true distribution is closer to a power-law.

The scarcity of very short runs demonstrates a failing of the premise of basing alignment on maximum parsimony, finding an optimum under some model. Parsimony is imperfect. With a large enough sample, the actual evolutionary history will contain

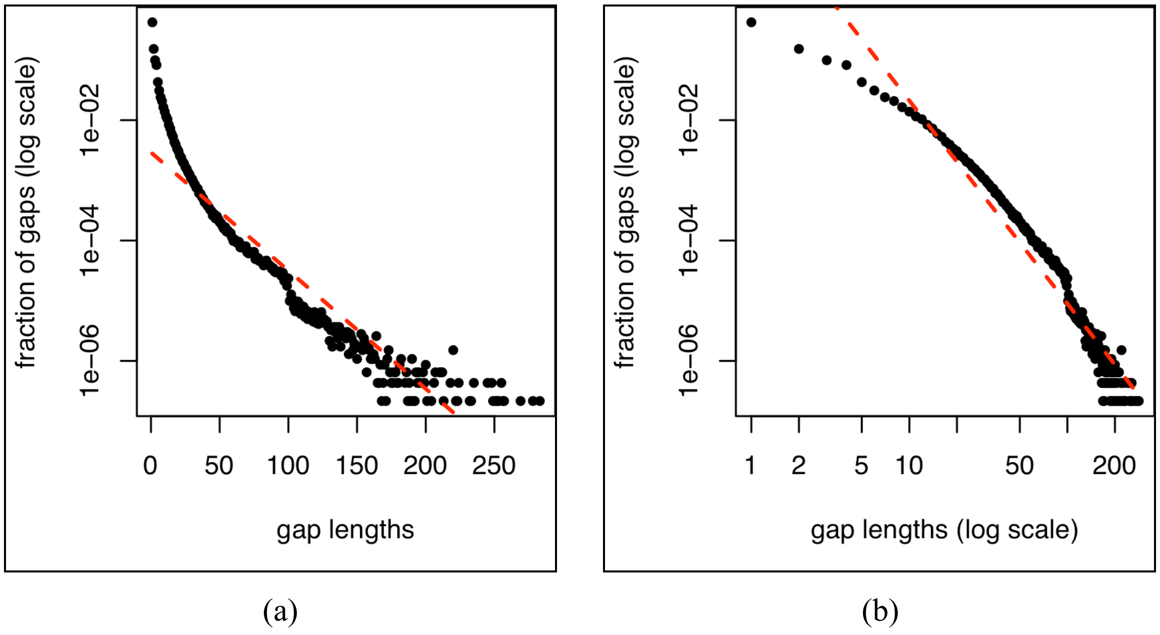


Figure 5.2 Gap Lengths Distribution. Gap lengths in medium G+C content regions of human aligned to dog. (a) Comparison to geometric distribution (dashed red line) shows the presence of more short gaps than would be expected. (b) Comparison to power law distribution (dashed red line) shows the presence of fewer short gaps and long gaps than would be expected.

events that are not the most probable. Holmes and Durbin (1998) and Lunter et al. (2007) have done seminal work measuring the expected incorrectness of maximum parsimony aligners, but this line of research is still limited.

5.4 Inferring Scores from Sequences

Given two sequences (as opposed to alignment examples), how should we apply (5.5) to infer scores? The solution in Chiaromonte et al. (2002) is to align using ±1 substitution scores and infer from the resulting alignments. Gaps were not addressed, so alignment was halted after gap-free extension. Inferring scores from alignments created by a recurrence (1.1) aligner raises questions of circularity. The process can be viewed as a function that maps one set of *starting scores* (chosen to create the alignments) to

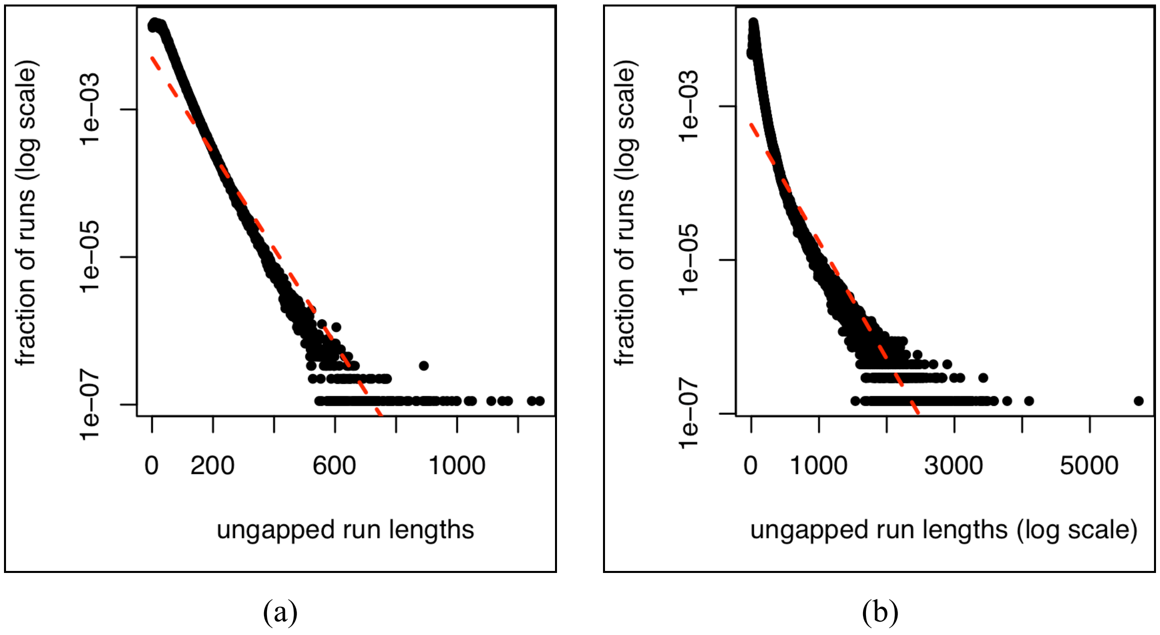


Figure 5.3 Ungapped Run Lengths Distribution. Lengths of ungapped runs in medium G+C content regions of human aligned to (a) dog and (b) macaque, compared to geometric distribution (dashed red lines). Both distributions show a abrupt drop-off of very short runs. Disregarding short runs, the distribution for dog is a close fit to a geometric, while for macaque there are still more short runs than would be expected.

another (inferred from the alignments), with the sequences as a fixed control variable. To what extent are the inferred scores affected by the choice of starting scores?

Chiaromonte et al. (2002) suggested that inferred scores might be improved by iterating the process, but left this as an open question. In fact the idea raises several questions. It is not immediately clear whether the iterated process will converge. If it does, will it converge to the same answer independent of the starting scores? Will the converged scores be the 'correct' scores (according to some model), and do the correct scores provide the best alignment results?

We tested iteration on simulated DNA data and on real data. Simulated data was generated according to the pair FSA model, and for a variation on that model that includes power law gap distributions (see section 5.3). For real data we used data from the ENCODE project.

We included gap score inference but chose a two-phase approach. We first iterated substitution scores inference until convergence, then used those substitution scores while iterating gap scores inference. The process is shown in figure 5.4. In order to avoid round off effects, we used floating-point scoring.

1	Start with substitution scores of ± 1
2	Find high scoring gap-free alignments, low identity
3	Infer substitution scores
4	Repeat (2 and 3) to convergence, orbit or divergence
5	Assume initial open and extend scores based on max substitution score
6	Find gapped alignments
7	Infer gap scores
8	Repeat (6 and 7) to convergence, orbit or divergence

Figure 5.4 Iterated Scoring Inference. Substitution scores are iterated from a starting point of ±1 match/mismatch to convergence. Initial gap scores are assigned relative to s_{max}, and iterated to convergence.

5.5 Experimental Results on Simulated Genomic Sequences

We applied the iterated inference procedure of section 5.4 to 135 simulated sequence pairs (section B.2). Of interest was whether the process would converge and if so whether the resulting scores matched the model scores.

Figure 5.5 shows typical gap score convergence for one of the simulated pairs. Twelve starting points were tried. Using the converged substitution scores, iteration for gaps scores was started at one of the 12 starting points and run to convergence. In all twelve cases convergence (to within 4 digits) occurred within 6 iterations, and all converged to the same point (to 6 digits). However, the convergence results did not match the model scores. s_{extend} was essentially correct, but s_{open} was about 20% too large.

These results are typical for the 54 sequence pairs with gaps generated per the pair FSA model. All but one of the pairs converged to a single attractor and did so in less than 10 iterations. One pair had two attractors, with nearly the same s_{open} but s_{extend} values differing by 7%. For all pairs (including the double attractor), the attractor's s_{open} is larger than the model, the error was as much as 30% but was usually within 10%. For all but two, s_{extend} was smaller than the model, off by as much as 10%; for most it was within 5%. Of the other two, one overestimated s_{extend} by 1%; the other was the double attractor case, in which both attractors overestimated s_{extend} by 10-20%. The double attractor case had model parameters p_{G+C}=31%, P_{match}=.65, P_{open}=.024 and P_{extend}=.81, all of which are extremes over the parameter space. Results are similar for the 81 power-law gap sequence pairs. Convergence results[17] for s_{open} are also too large but as a rule are much closer to the model, all cases being within 7%. Of the 81 sequence pairs, two lead to multiple attractors.

To evaluate whether the inference process improves scores—improves the alignments produced with those scores—we evaluated the alignments using each score set along the iteration path. Iteration was performed according to the process in figure 5.4 with only one starting point for gap scores. For comparison, the default BLASTZ score set was also tested. Since the true alignments for these sequence pairs are known, true

[17] While s_{extend} participated in convergence, we cannot interpret the accuracy of s_{extend}; there is no model value to compare to.

(TP) and false positives (FP) can computed, allowing us to evaluate by ROC, both visually and numerically.

Figure 5.6 shows the performance of each score set for the same sequence pair used in the convergence discussion. Iteration produced 15 score sets. The best set was gaps0, which combines the converged substitution scores with default gap scores (s_{open}=-3.5s_{max} and s_{extend}=-0.20s_{max}). Several score sets had a higher ratio of TP to FP for high scoring alignments, but found less total TP. For example, gaps2, gaps3, and gaps4 have fewer FP than gaps0 up about 45,000 TP but don't find much more TP than that. Gaps1 exhibits the same effect, but with a smaller FP advantage and a higher TP limit. For this sequence pair, each iteration of gap scores made things worse. Iteration of substitution scores seems to have made little difference. Other than subs0, which is a simple ±1 match/mismatch set, the substitution sets are so close that many of them are obscured in this plot. It is interesting to note that the BLASTZ default scoring set finds more TP than any of the other sets tested, 1.3% more than gaps0 (65,908 bp vs. 65,052). However, it also has a lower TP/FP ratio. Also worth noting is the relatively low TP for all score sets. The best set only found 82% of the homology. This sequence pair has the lowest identity and highest gap rates of the simulated sets.

Disappointingly, these results are not typical of all the simulated sequence pairs. We have noticed no trend as to which of the score sets produced by iteration is the best.

5.6 Experimental Results on Actual Genomic Sequences

We applied the iterated inference procedure to 35 modified ENCODE sequence pairs, with coding and repeat regions removed from human and repeats unmasked for the other species (section B.3). For these sequences we do not know the correct answers and have to use a different means to evaluate the quality of results. For this purpose, we estimate the false positive rate by aligning one sequence to the other sequence, backwards. The backwards sequence has no known biological meaning, as it is reversed but not complemented. As such, it is much like a random sequence with the same base composition as the original, but it also maintains some local base variations from the

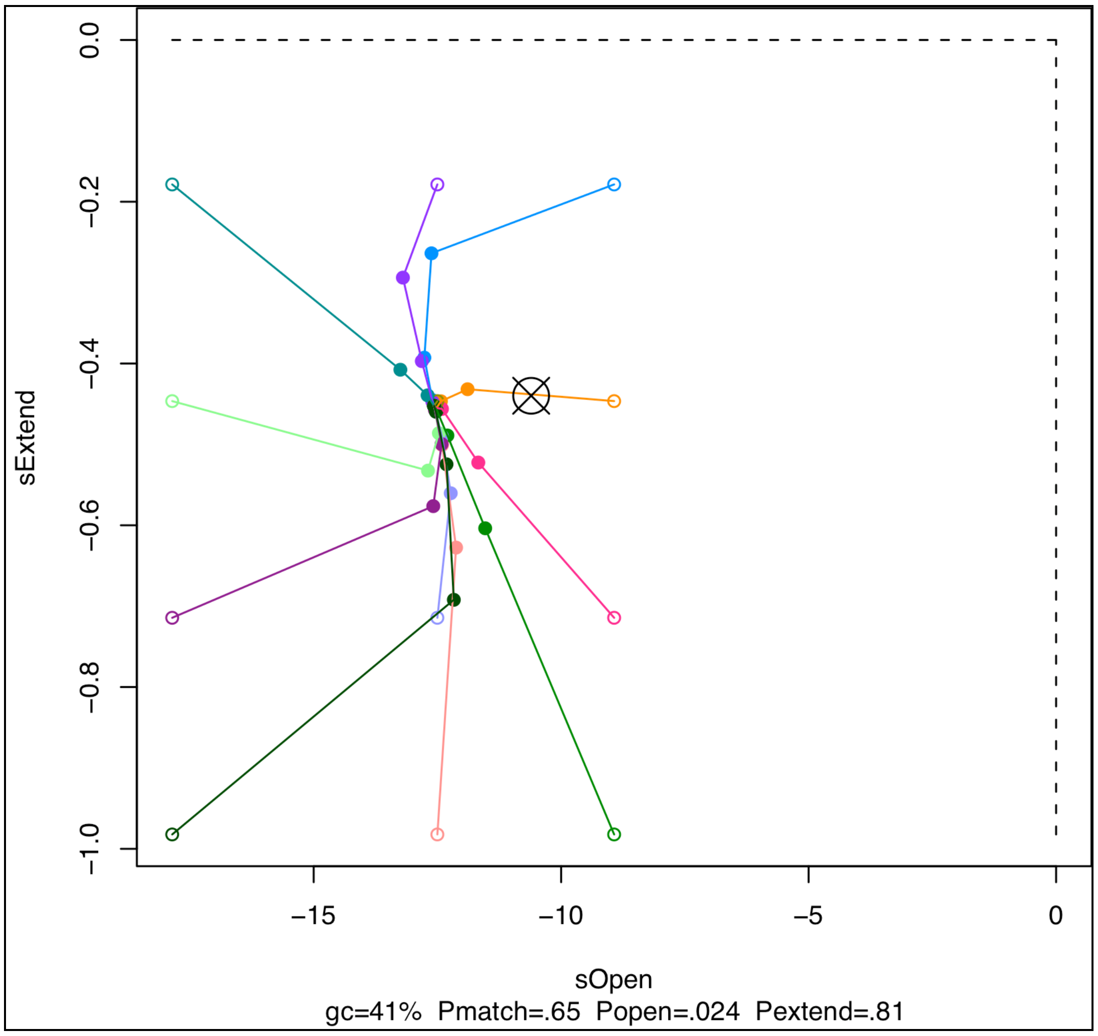


Figure 5.5 Scores inference convergence on simulated sequence pair. Open circles indicate twelve different starting points for gap scores; lines and solid circles show the progression of iterated scores. Each path converges to s_{open}=-12.5 s_{extend}=-0.45 (O=700 E=25 if scaled so that s_{max}=100). Target shows score expected from model, to s_{open}=-10.6 s_{extend}=-0.44.

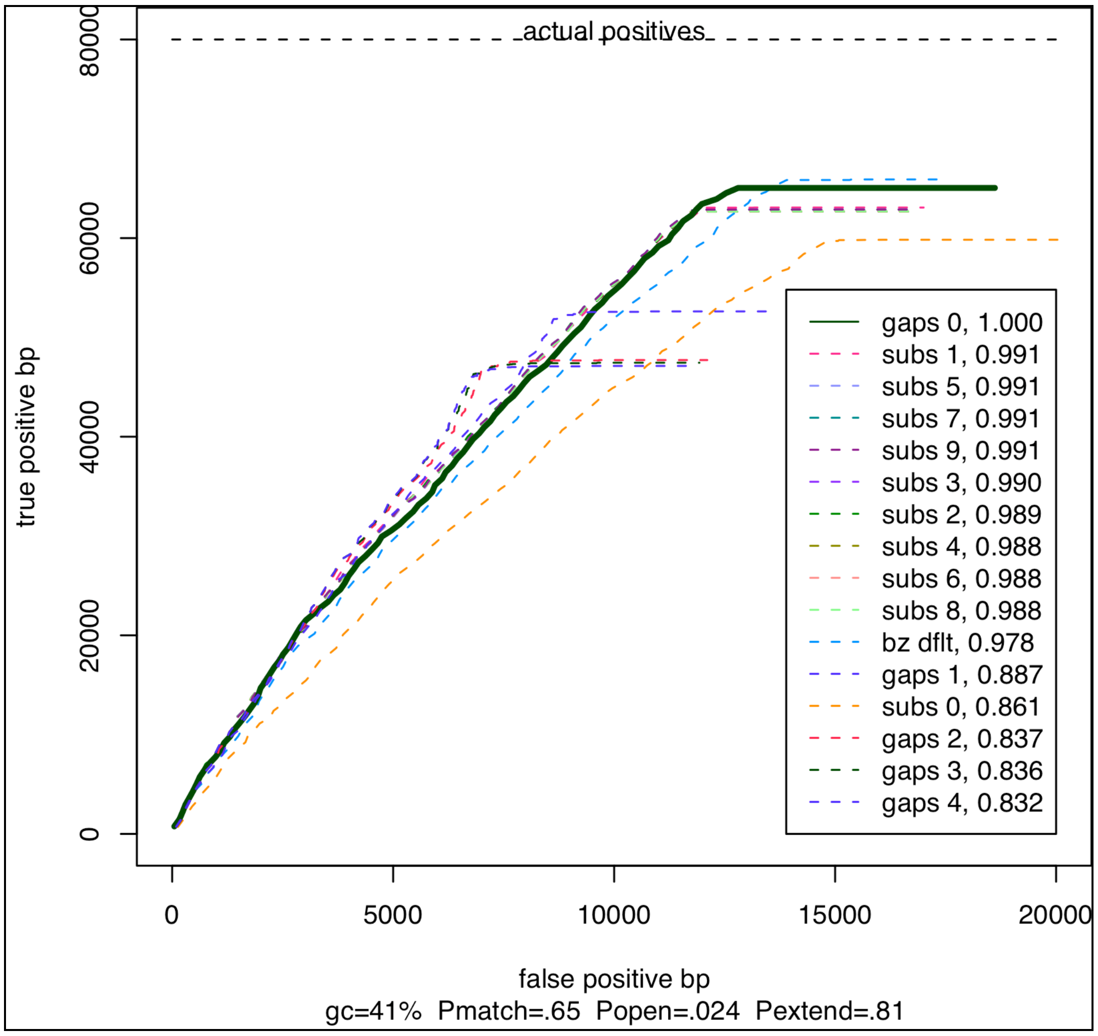


Figure 5.6 Scores inference performance on simulated sequence pair. Dashed lines are ROC plots of iterated score sets. Solid line is for best score set. Legend shows score sets ranked by ROC_{20000}. Names indicate position in iteration sequence (subs 0 is initial substitution score set, subs 1 is after one iteration, etc.). Value shown is ratio of the score set's ROC_{20000} to that of the best. Score sets were scaled so that s_{max}=100 and rounded to nearest integers. For each curve, the highest scoring alignments occur in the lower left, and score decreases along the curve. The actual alignment contains 80,000 homologous base pairs.

original that a simple random sequence would not[18]. Using this estimate of FP, we can evaluate alignments visually by comparing total aligned bases to FP. This is similar to ROC, with the major difference that aligned bases equals TP+FP. This evaluation technique has shortcomings, which we will address shortly.

As in section 5.5, we evaluate the score sets along the inference iteration path. Figure 5.7 shows the performance of each score set for region ENm001, human vs. mouse. The gaps0 set stands out as the best, aligning significantly more bases for any value of FP. For example, at FP=0 it identifies 12% more homology than the next best score set (730K bp vs. 660K).

While 5.7 shows very nice results with a clear winner, using alignments to backwards sequence as an estimate of FP leads to a logical inconsistency which we are not able to explain. If the estimate were exactly the FP rate, then we could compute TP by subtracting FP from the total bases aligned. This would enable us to plot a true ROC plot; such a plot is shown in figure 5.8. Unfortunately, this leads us to the nonsensical conclusion that the total number of TP can *decrease* as we discover more FP. This is evident from the negative slope of the top of every curve[19].

The crux of the inconsistency is demonstrated in figure 5.9. No FP bases are found above a certain score. As the score decreases, the FP rate increases, but initially is increasing slower than the total bases aligned. Eventually the FP rate exceeds the total rate, and the number of TP decreases. We are hard-pressed to explain this paradox; possibly it is an artifact of random FP having quadratic growth[20] (relative to sequence length) while TP is essentially linear. We briefly investigated using alignments on random sequences for an FP estimate but it appeared the paradox still existed. Figure 5.10 illustrates why—the scoring distributions for both FP estimates have similar shapes. The

[18] For example, promoter regions usually contain an elevated rate of G+C compared to the rest of a genome.

[19] This is true even for gaps0, but it is less noticeable.

[20] Since our unit of comparison is an alignment *column*, rather than an aligned *base*, the expected number of FP contributed by a particular base in sequence 1 grows with the length of sequence 2. This is in contrast to the expected number of TP contributed, which, in the absence of repeats in sequence 1 in this test, is constant.

only apparent difference is that alignments of the random sequences tended to score higher.

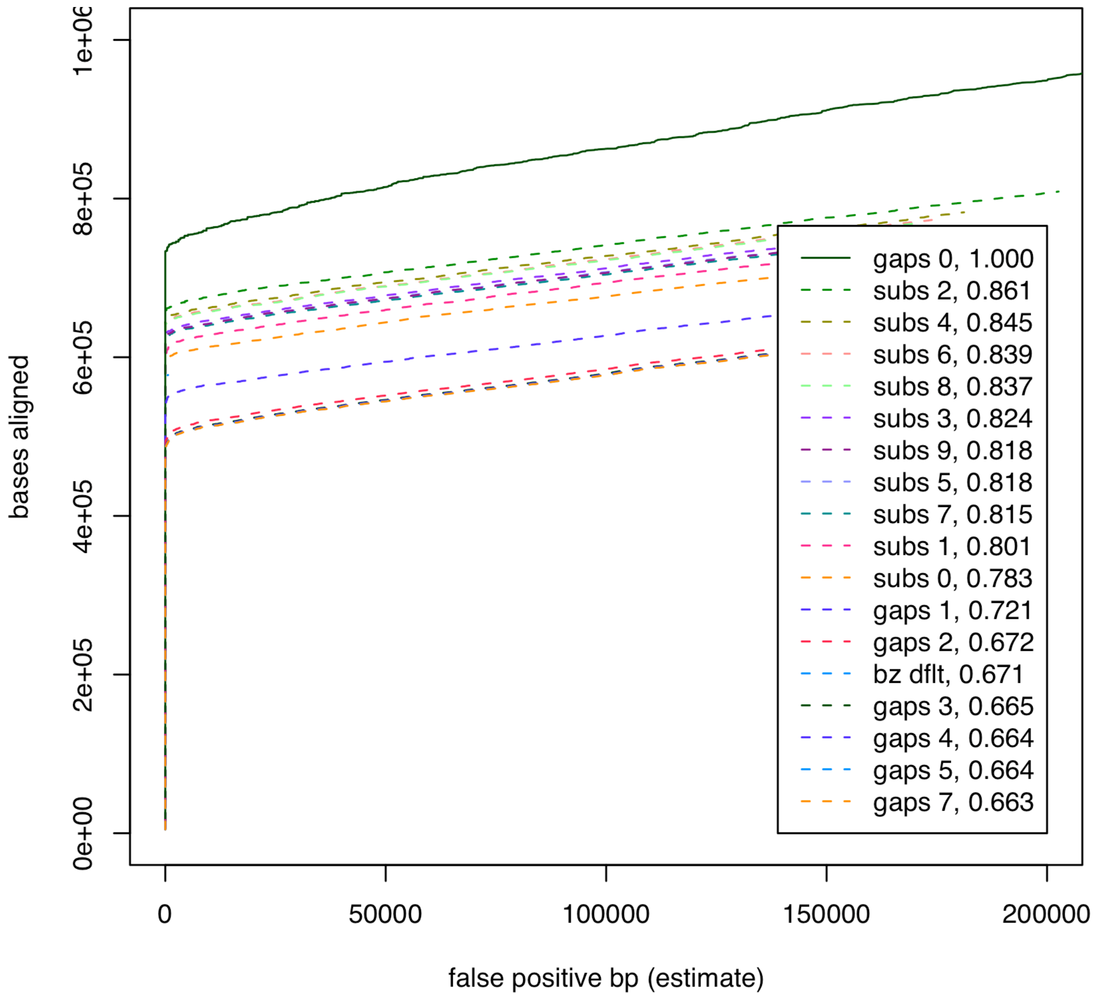


Figure 5.7 Scores inference performance on real sequence pair. Vertical axis is all bases aligned for ENm001 human vs. mouse with K=0. Horizontal axis is false positive rate estimated as bases aligned to backward mouse sequence. Solid line is for best score set. Legend shows score sets and ROC ratios as in figure 5.6.

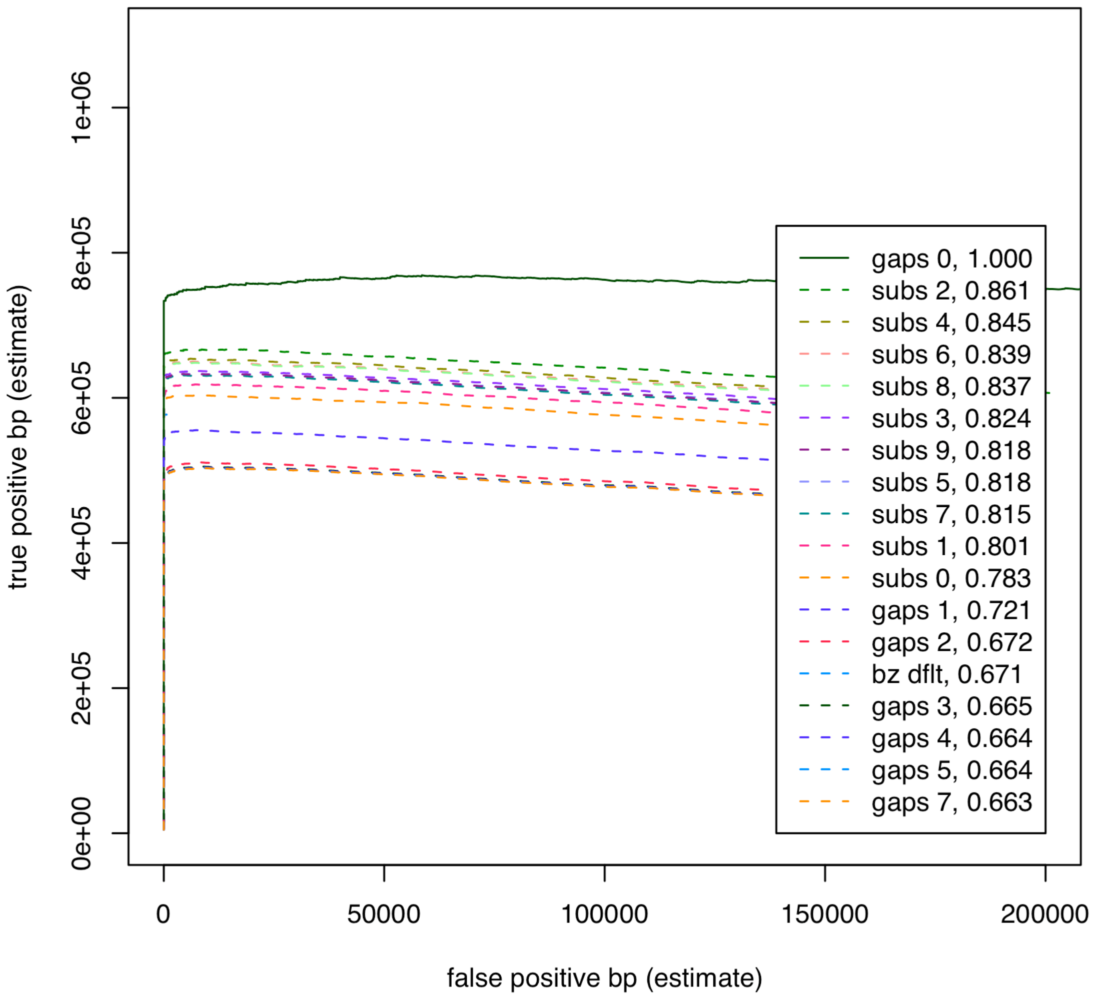


Figure 5.8 Scores inference performance adjusted for true positive rate. The same data is shown as in figure 5.7, but the vertical axis has been replaced by an estimate of the true positive rate (true positive = bases aligned – false positive). All curves contain a portion with negative slope, indicating a logical inconsistency.

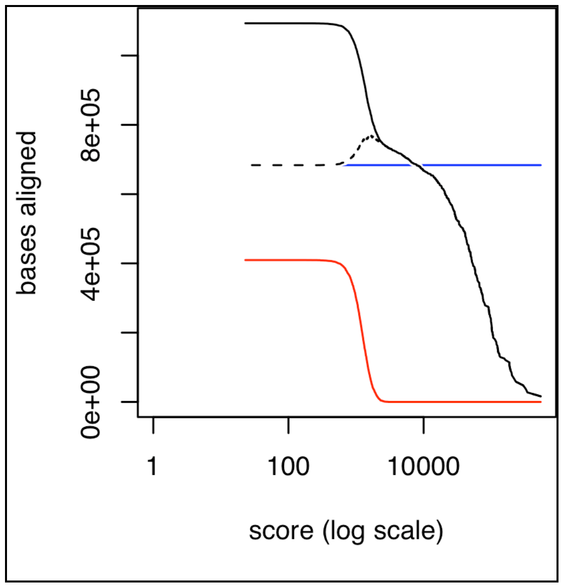
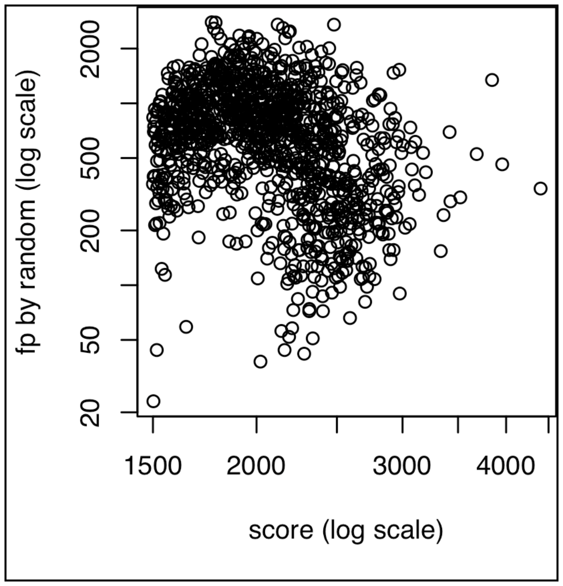
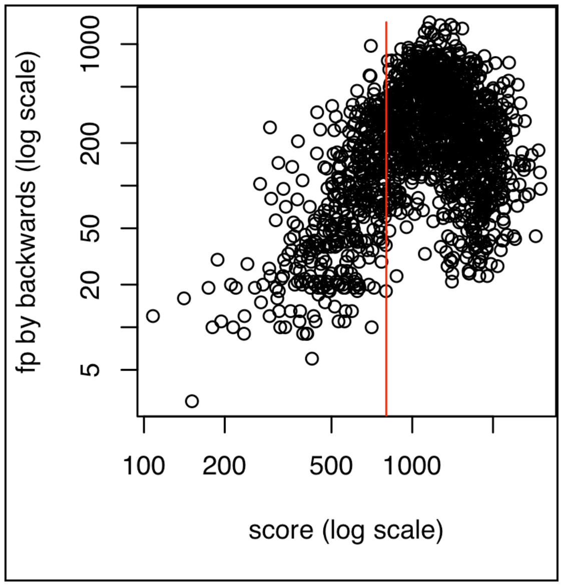


Figure 5.9 Estimating false positive rate by alignment to backward sequence. Bases aligned for ENm001 human vs. mouse with K=0 using the best score set found (gaps0). (Solid black) count of all aligned bases. (Red) bases aligned to backward mouse sequence. (Dashed) true positives, estimated by subtracting backward count from total count. (Blue) total true positives estimated. Logical inconsistency is revealed where estimated true positive curve is above total true positives estimate.



(a)



(b)

Figure 5.10 False positive scoring distributions. Distribution of scores for two estimates of false positive alignments using the gaps0 score set. (a) Estimated by alignment of random sequences, with K=1500. (b) Estimated by alignment of ENm001 human vs. backward mouse sequence with K=0. Shape of distribution to the right of the red line is similar to shape for random sequences.

Chapter 6

LASTZ Experimental Results

6.1 Hashed Diagonal Extent Table

To filter out redundant gap-free extension of seed hits we keep track of the extent of expansion along each diagonal. To reduce memory requirements for large sequences, we track only by a 16-bit hash value of the diagonal. Hash collisions will cause us to reject some seed hits and cause us to miss some homology.

To evaluate the effect of hash collisions, we ran alignments on 35 pairs of ENCODE sequences using LASTZ with default settings and a special version of LASTZ without diagonal hashing. The number of hash failures was counted, and the difference between the resulting alignments was measured (dividing alignment columns into true and false positives and false negatives, as described in section B.5).

It should be noted that not all collisions are hash failures. A collision occurs whenever the current seed hit's diagonal differs from the most recent diagonal with the same hash value. A failure occurs only when the extent stored for the diagonal's hash equivalent is beyond the seed hit. In other words, when some other diagonal with the same hash value contains an HSP that extends beyond the current seed hit, we have a failure. This is rare; the majority of collisions are not failures,

Table 6.1 shows the results. The failure rate is small but increases with sequence length. Though the rate is small, due to the large number of seed hits the number of failures is seemingly large, in some cases more than ten thousand. Regardless, there is no disagreement in the resulting alignments. In spite of the failures, there are still enough seed hits to identify the same alignments.

6.2 Z-step

The amount of memory used for seeding can be reduced by the use of sparse spacing. Seed words are only stored for positions that are multiples of the z-step. In

Table 6.1 Hash failures in the diagonal extent table.

region	human bp	species	bp	seed hits	hash failures	failure rate	alignment coverage
ENm001	1.11M	baboon	1.95M	5.10M	10,309	0.20%	1,019,525
		mouse	1.49M	3.28M	6,762	0.21%	529,724
		dog	1.51M	3.15M	7,069	0.22%	721,254
		opossum	1.83M	3.90M	7,042	0.18%	78,682
		chicken	744K	1.48M	2,511	0.17%	14,303
ENm012	647K	baboon	1.16M	2.47M	3,367	0.14%	645,347
		mouse	1.25M	2.03M	2,803	0.14%	405,847
		dog	919K	1.63M	2,283	0.14%	564,253
		opossum	1.26M	2.18M	2,608	0.12%	190,504
		chicken	528K	849K	1,073	0.13%	89,438
ENm014	646K	baboon	1.36M	2.26M	2,385	0.11%	620,571
		mouse	1.16M	1.61M	1,698	0.11%	394,012
		dog	990K	1.56M	1,859	0.12%	551,268
		opossum	1.43M	2,14M	2,053	0.10%	78,422
		chicken	513K	670K	864	0.13%	9,510
ENr114	266K	baboon	584K	637K	281	0.04%	258,549
		mouse	595K	374K	190	0.05%	73,265
		dog	427K	339K	233	0.07%	203,912
		opossum	506K	378K	157	0.04%	11,800
		chicken	182K	95K	37	0.04%	1,529
ENr132	318K	baboon	498K	373K	66	0.02%	269,169
		mouse	424K	174	31	0.02%	38,203
		dog	318K	154K	32	0.02%	90,074
		opossum	991K	373K	65	0.02%	7,610
		chicken	328K	120K	21	0.02%	2,237
ENr221	289K	baboon	819K	609K	156	0.03%	285,148
		mouse	568K	277K	74	0.03%	140,615
		dog	498K	329K	148	0.05%	238,898
		opossum	591K	327K	97	0.03%	33,227
		chicken	251K	119K	58	0.05%	8,672
ENr323	236K	baboon	1.02M	650K	120	0.02%	227,477
		mouse	516K	232	73	0.03%	86,066
		dog	437K	259K	83	0.03%	184,341
		opossum	611K	312K	80	0.03%	48,763
		chicken	136K	56K	9	0.02%	10,607

addition to memory savings, z-step can speed up the seeding stage by reducing the number of seed hits and the number of gap-free extensions performed[21]. Subsequent stages are also sped up due to a reduction in the number of anchors. Z-step can

[21] The reduction in HSPs is not linear, though. Since most HSPs contain more than one seed hit, an HSP will only be completely missed if all of its seed hits are at non-multiples of z.

potentially cause loss of sensitivity in the resulting alignments, in return for a gain in speed.

To compare the speed gain to sensitivity loss, we ran alignments on 35 pairs of ENCODE sequences using z-step values of 2, 5, 10, 20, 50 and 100, with default settings otherwise. For timing comparisons we aligned without a z-step. For alignment differences we compared to a BLASTZ alignment. Figure 6.1 shows results for two of seven regions. In 6.1(a) it is apparent for ENm014 that the time saved by using a z-step always exceeds the loss in sensitivity, even for large z-steps. The miss rate is remarkably low for baboon; for Z=100 the miss rate was slightly less than 1%, suggesting that long z-steps are a viable strategy for closely related sequences. Five of the seven regions had loss of around 1% at Z=100, while the other two had loss of 1.6% and 2.5%.

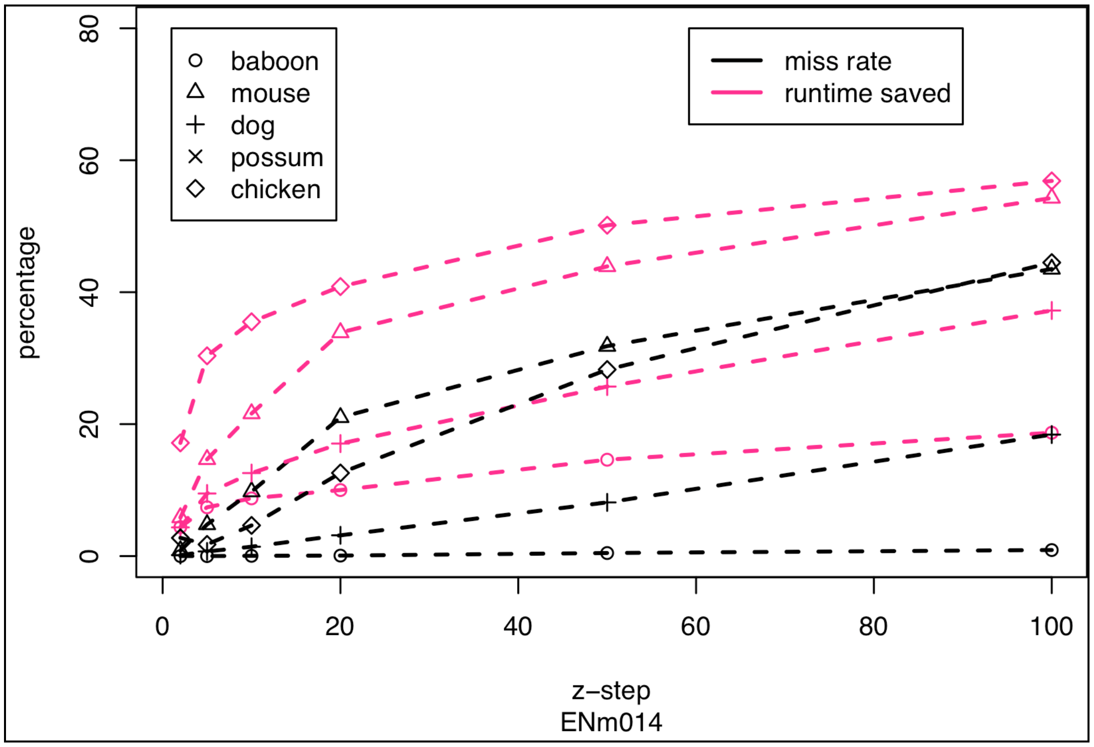
The results for ENm014 are typical of the other regions, but there are exceptions. 6.1(b) shows some odd results for ENr114. For chicken Z=20 performs worse than Z=50, and the miss rate actually exceeds time savings at Z=100. Inspection of table 6.1 reveals that chicken ENr114 has a low number of seed hits compared to other species and regions. The only alignment with fewer seeds hits, chicken ENr323 (not shown), exhibits a similar anomaly, with runtime for Z=20 exceeding that for Z=10, even though there is negligible loss in sensitivity for either.

It should be noted that the sensitivity losses for large Z, while lower than the time savings, are still unreasonably large for most applications. They are of interest here as a means of reducing the alignment time during inference.

6.3 Twin Hit Seeds

LASTZ supports twin hit seeds, in which two nearby hits on the same diagonal are required before gap-free extension is performed. Figure 6.2 shows an example of how they can increase specificity. Comparing 500K bp regions of human and mouse with K=1500. Without requiring twin hits, as in 6.2(a) and (b), we get a lot of alignment 'noise'. In 6.2(c) and (d) twin hits were required, overlapping by as much as 10 bp or separated by up to 10 bp. The twin hit seed removes nearly all the noise in the HSP stage. Gapped alignment speed is greatly affected by the number of HSPs. The noisy alignment in 6.2(b) took 40.4 seconds, the cleaner alignment in hits 6.2(d) took only 7.6 seconds.

(a)



(b)

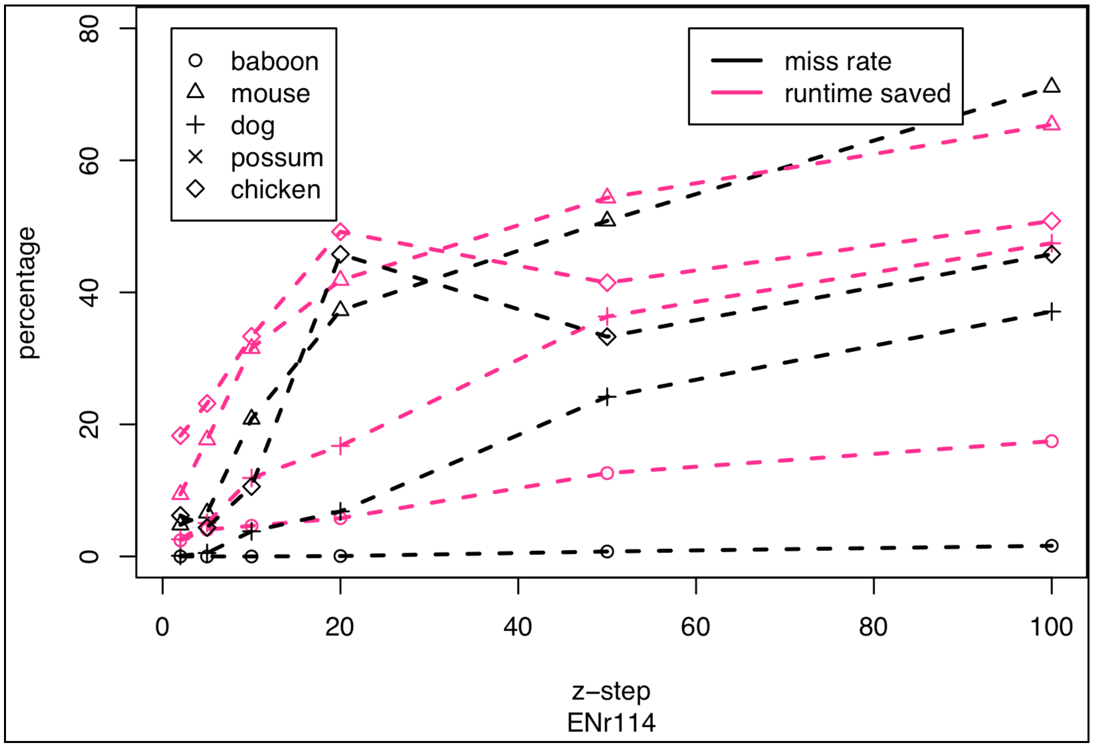


Figure 6.1 Z-step experimental results. (a) ENCODE region ENm014 exhibits regular behavior. (b) ENCODE region ENr114 shows strange behavior.

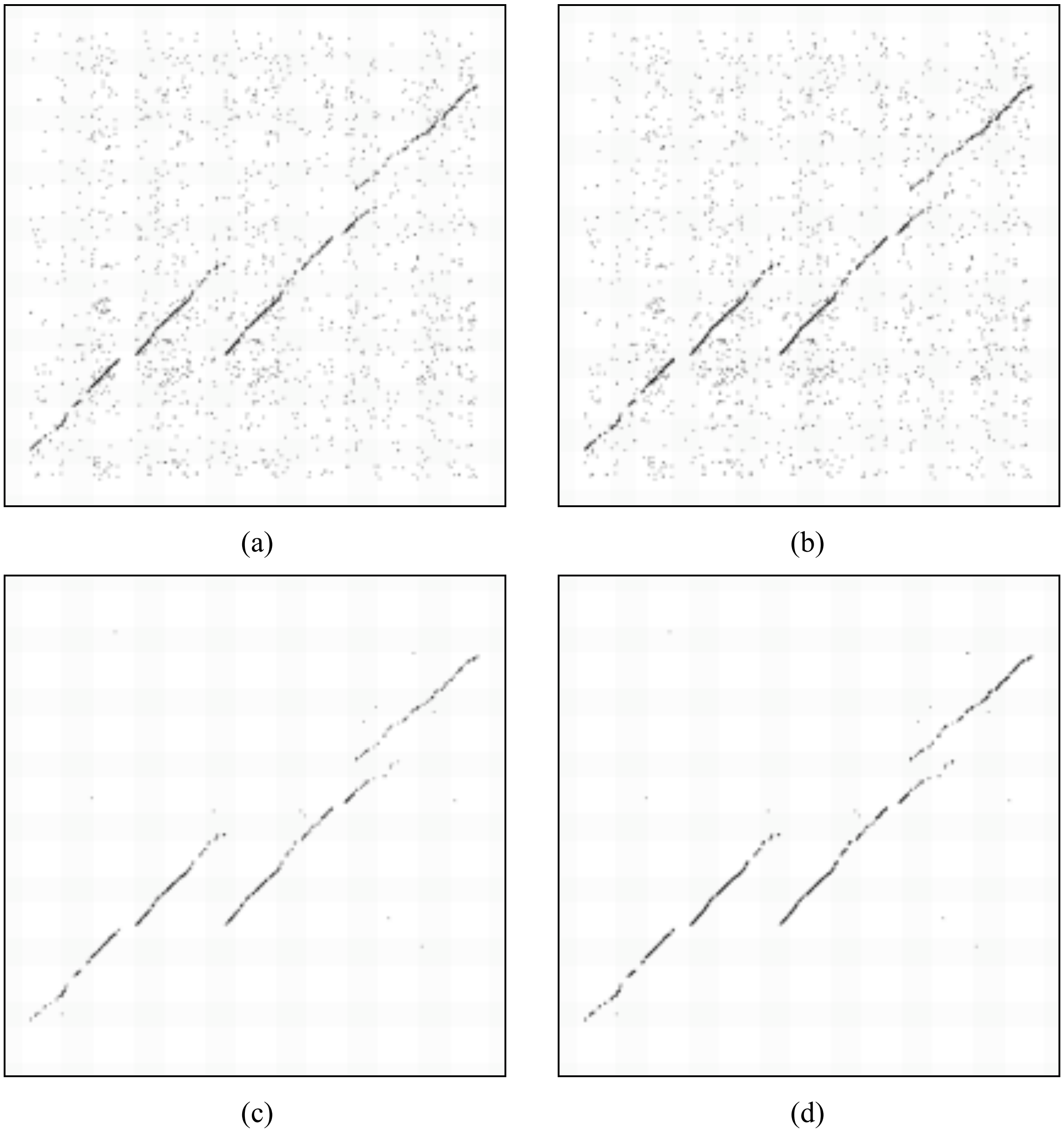


Figure 6.2 Twin hit seed. Dot plots for alignments of ENCODE region ENr233 human (horizontal axis, 500K bp) to mouse (vertical axis, 466K bp). (a) HSPs for single hit seed. (b) Gapped alignments from single seed hits. (c) HSPs for twin hit seed. (d) Gapped alignments from twin seed hits. The same spaced seed was used for all cases. All four plots have undergone identical contrast adjustment to emphasize the presence of extra alignments in (a) and (b).

Table 6.2 Statistics for single hits vs. twin hits.

	single hits	twin hits
gap-free extensions	220,402	1,025
bp extended (gap-free)	13,087,680	147,060
bp per gap-free extension	59	143
HSPs	3,069	885
anchors extended	1,814	175
gapped extensions	3,628	350
DP cells visited	1,670,368,460	295,283,732
fraction of DP matrix visited	1/140th	1/789th
DP cells per gapped extension	460,410	843,668
run time (seconds)	40.4	7.6

Chapter 7

Conclusions and Future Work

We have set in place a platform for automated alignment parameterization, LASTZ, and used that platform to create and evaluate a program, INFERZ, to infer scoring sets based on a well-understood mathematical model. We have found that multi-step inference of scores converges to an approximation of the correct scores but tends to overestimate the gap open penalty and underestimate the gap extend penalty. Further, we have found that this method is unpredictable as a means of finding an optimal score set. The convergence point is no more likely to be the best score set than any other set along the convergence path.

We view this work as a starting point toward the goal of completely automating alignment parameterization. In addition to inferring score sets, we will also need to automatically choose seeding strategies and thresholds. Ideally the user should only have to choose one parameter, ranging from zero (fast run time, lower sensitivity) to one (high sensitivity, slower run time).

The improvements in LASTZ, in comparison to BLASTZ, represent the next step in the evolution of BLASTZ. The primary guiding factor was to give INFERZ more optimization choices, most of which have yet to be explored.

We found that what appeared to be a reasonable scheme for automatically evaluating score sets—using alignment to backwards sequences—suffers from a logical inconsistency. Though this is a disappointing result, it does not render the scheme entirely useless. There remains a score region outside of the inconsistency, specifically for alignments scoring high enough that the false positive estimate is low, and we can make use of this to compare alignment results and score sets.

Being able to quickly evaluate score sets is important in light of the inconsistency of iterated inference. The promise of iterated inference was that it would converge to an optimal or near optimal score set, eliminating the need for evaluation feedback. As this is not true, other optimization techniques should be explored. Convergence plots suggest the score space terrain is smooth enough that hill climbing techniques would work. This requires quick evaluation of many scores sets. This might be accomplished by aligning a coarse subsample of the sequences with a very large z-step, but the effect of doing so has not been explored.

The additional versatility of LASTZ allowed us to test several new seeding strategies, but there are still more strategies that could be tried, for example the multiple seeds of Buhler et al. (2003) and Li and Ma (2003). Half-weight seeds have only been evaluated empirically, by Hou et al. (2007), and deserve a mathematical analysis, such as was done for (full weight) spaced seeds in Buhler et al. (2003), to determine their expected effect on sensitivity. The current implementation of LASTZ is slower than BLASTZ during seed hit processing. There is no intrinsic reason why this should be the case. This aspect of the program has received little attention to date, and the author believes this can be brought into line with the speed of BLASTZ when they are performing identical alignments. In addition, speed gains may be accomplished by using more than two stages of interpolation.

Because affine gap scoring is in conflict with empirical evidence of gap length distributions, LASTZ could be improved by incorporating a slower but more biologically realistic gap scoring model as a post processing step. Cartwright (2006) has shown the viability of post processing to improve alignment quality, but incorporating feedback from the post processed alignments could improve the scores used during the main alignment phase. Even if we achieve the optimal score set for affine gap alignment by itself, we may need a suboptimal score set to achieve the best alignments after post processing.

The quantum alignment capability incorporated in LASTZ also deserves further exploration. With appropriate scoring matrices and short queries (<20 bp), gap-free quantum-to-DNA alignment is equivalent to a position-weight-matrix motif finder. The ability to allow gaps and efficiently handle longer queries could be useful for long-motif applications, such as repeat finding. Adding support for quantum-to-quantum alignment would allow it to be used as the basis for a progressive multiple aligner, maintaining probabilistic base information for every internal ancestor.

BIBLIOGRAPHY

Altschul, S.F., Madden, T.L., Schaffer, A.A., Zhang, J., Zhang, Z., Miller, W. and Lipman, D.J. 1997. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Research* **25**:3389–3402.

Batzoglou S. 2005. The many faces of sequence alignment. *Briefings in Bioinformatics* **6**:6-22.

Bray N. and Pachter L. 2004. MAVID: Constrained ancestral alignment of multiple Sequences. *Genome Res*earch **14**:693-699.

Buhler J., Keich U. and Sun Y. 2003. Designing seeds for similarity search in genomic DNA. *Proc. 7th Annual International Conference on Research in Computational Molecular Biology (RECOMB'03),* 67–75.

Burrows, M. and Wheeler, D.J. 1994. A block-sorting lossless data compression algorithm. Technical report, Digital SRC, 1994. Research Report 124.

Cameron, M., Williams, H.E. and Cannane, A. 2006. A deterministic finite automaton for faster protein hit detection in BLAST. *Journal of Computational Biology* **13**:965-978.

Cartwright, R.A. 2006. Logarithmic gap costs decrease alignment accuracy. *BMC Bioinformatics* **7**:527.

Chiaromonte, F., Yap, V.-B. and Miller, W. 2002. Scoring pairwise genomic sequence alignments. *Pacific Symp. Biocomputing* 115–126.

Durbin, R., Eddy, S., Krogh, A. and Mitchison, G. 1998 *Biological Sequence Analysis*, Cambridge University Press, New York.

ENCODE Project Consortium 2004. The ENCODE (ENCyclopedia Of DNA Elements) Project. *Science* **306**:636–640.

Farris J. 1977. Phylogenetic analysis under Dollo's law. *Systematic Zoology* **26**:77-88.

Felsenstein, J. 1981. Evolutionary trees from DNA sequences. *Journal of Molecular Evolution* **17**:368–376.

Flannick J. and Batzoglou S. 2005. Using multiple alignments to improve seeded local alignment algorithms. *Nucleic Acids Research* **33**:4563-4577.

Gotoh O. 1982. An improved algorithm for matching biological sequences. *Journal of Molecular Biology* **162**:705-708.

Gribskov, M. and Robinson, N. 1996. Use of receiver operating characteristic (ROC) analysis to evaluate sequence matching. *Computers and Chemistry* **20**:25-33.

Gusfield, D. 1997. *Algorithms on Strings, Trees and Sequences*, Cambridge University Press, New York.

Haussler, D. 2005. Personal communication.

Holmes, I. and Durbin, R. 1998. Dynamic programming alignment accuracy. Journal of *Computational Biology* **5**:493-504.

Hou, M., Harris, R.S. and Miller, W. 2007. A new seeding strategy for DNA alignment. *RECOMB 2007 poster proceedings*.

Hudek A. and Brown D. 2005. Ancestral sequence alignment under optimal conditions. *BMC Bioinformatics*. **6**:273.

Joachims, T. 2003. Learning to align sequences: a maximum-margin approach (Technical Report). Cornell University.

Kececioglu, J. and Kim, E. 2006. Simple and fast inverse alignment. In *Proceedings of the 10th ACM Conference on Research in Computational Molecular Biology (RECOMB)*, 441–455.

Levenstein V. 1966. Binary codes capable of correcting insertions and reversals. *Sov. Phys. Dokl.* **10**:707-710.

Li, M. and Ma, B. 2003. PatternHunter II: highly sensitive and fast homology. *Genome Informatics* **14**:164–175.

Lippert, R. 2005. Space-efficient whole genome comparisons with Burrows–Wheeler transforms. *Journal of Computational Biology* **12**:407-415.

Lunter, G., Rocco, A., Mimouni, N., Caldiera, A and Hein, J. 2007. Uncertainty in homology inferences: assessing and improving genomic sequence alignment. (submitted).

Ma B., Tromp J. and Li M 2002. PatternHunter—faster and more sensitive homology search. *Bioinformatics* **18**:440-445.

Miller, W. et al. 2007. 28-Way vertebrate alignment and conservation track in the UCSC Genome Browser. *Genome Research* in press.

Needleman S. and Wunsch C. 1970. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology* **48**:443-453.

Pruitt, K. D. and Maglott, D. R. 2001. RefSeq and LocusLink: NCBI gene-centered resources. *Nucleic Acids Res*earch **29**:137–140.

Schwartz, S., Zhang, Z., Frazer, K., Smit, A., Riemer, C., Bouck, J., Gibbs, R., Hardison, R., and Miller, W. 2000. PipMaker: A web server for aligning two genomic DNA sequences. *Genome Research* **10**:577–586.

Schwartz, S., Kent, W.J., Smit, A., Zhang, Z., Baertsch, R., Hardison, R.C., Haussler, D., and Miller, W. 2003. Human–mouse alignments with BLASTZ. *Genome Research* **13**:103–107.

Siepel, A. and Haussler, D. 2003. Phylogenetic estimation of context-dependent substitution rates by maximum likelihood. *Molecular Biology and Evolution* **21**:468-88.

Siepel, A. 2005. Personal communication.

Smith T. and Waterman M. 1981. Identification of common molecular subsequences. *Journal of Molecular Biology* **147**:195-197.

Sun Y. and Buhler J., 2005. Designing Multiple Simultaneous Seeds for DNA. *Journal of Computational Biology* **12**: 847–861.

Sun Y. and Buhler J., 2006. Choosing the best heuristic for seeded alignment of DNA sequences. *BMC Bioinformatics* **7**:133.

Tamura, K. 1992. Estimation of the number of nucleotide substitutions when there are strong transition-transversion and G+C content biases. *Molecular Biology and Evolution* **9**:678-687.

Yang Z. 1994 Estimating the pattern of nucleotide substitution. *Journal of Molecular Evolution* **39**:105-11.

Zhang, Z., Berman, P. and Miller, W. 1998. Alignments without low-scoring regions. *Journal of Computational Biology* **5**:197–210.

Zhang, Z. and Gerstein, M. 2003. Patterns of nucleotide substitutions, insertions and deletions in the human genome as inferred from human pseudogenes. *Nucleic Acids Research* **31**:5338–5348.

Zhou, L., and Florea, L. 2007. Designing sensitive and specific spaced seeds for cross-species mRNA-to-genome alignment. *Journal of Computational Biology* **14**:113-130.

Appendix A

Glossary

HSP	High-scoring Segment Pair. An alignment between two sequences, containing no gaps.
p_{match}	Probability that an alignment column contains identical nucleotides.
p_{xition}	Probability that an alignment column contains a transition; either both are purines (A or G) or both are pyrimidines (T or C)
p_{xversion}	Probability that an alignment column contains a transversion (one purine and one pyrimidine)
p_{xy}	Probability of an alignment column with x for sequence 1 and y for sequence 2.
$p_{\mathrm{x}\bullet}$	Probability of an alignment column with x for sequence 1.
$p_{\bullet\mathrm{y}}$	Probability of an alignment column with y for sequence 2.
$p_{\mathrm{G+C}}$	Probability of a G or C.
$q_{\mathrm{A}}, q_{\mathrm{C}}, q_{\mathrm{G}}, q_{\mathrm{T}}$	For quantum base q, the probability that q=A (or C, G or T).
s_{xy}	Log-odds score for an alignment column with x for sequence 1 and y for sequence 2.
s_{open}	Log-odds score for opening a gap.
s_{extend}	Log-odds score for extending a gap.
s_{max}	Maximum log-odds score for a score set.
$\tilde{\mathrm{x}}$	Nucleotide complement. $\tilde{\mathrm{A}} = \mathrm{T}, \tilde{\mathrm{C}} = \mathrm{G}, \tilde{\mathrm{G}} = \mathrm{C}$ and $\tilde{\mathrm{T}} = \mathrm{A}$.
$\lvert X \rvert$	Length of (number of bases in) sequence X.
$X_{[i..j]}$	Subsequence of X consisting of bases i through j, inclusive.
x-drop	Limit on negative scoring segments allowed in an gap-free extension.
y-drop	Limit on negative scoring sub-alignments allowed in an gapped extension.
z-step	Seed word position granularity. With a z-step of 5 only every 5th position in sequence 1 is used.

Appendix B

Methods

B.1 Analysis of 28-Vertebrate Alignments

In order to have typical genome-wide parameters for simulation data, we measured statistics over pairwise alignments of human to six vertebrates—macaque, mouse, dog, opossum, platypus and chicken (assemblies hg18, rheMac2, mm8, canFam2, monDom4, ornAna1 and galGal3). We projected pairwise alignments from whole genome alignments of 28 species (Miller et al., 2007) downloaded from the UCSC Genome Browser (http://genome.ucsc.edu). Alignment blocks were partitioned into three sets by human G+C content, divided into low (less than 37.5%), medium (37.5% to 45.5%) and high (more than 45.5%).

Table B.1 shows estimated genome-wide probabilities of nucleotide match, transition, transversion, gap open and gap extend. These provide guidance for parameter choices of the simulated data sets of section B.2.

Table B.2 shows alignment scores inferred from observed probability estimates as per (5.5). These provide a base point for comparing gap scores to those used in BLASTZ. Typically BLASTZ scores are scaled so that the maximum substitution score (s_{max} here) is 100. Thus the ratios s_{open}/s_{max} and s_{extend}/s_{max} correspond to BLASTZ's O and E parameters, divided by -100. BLASTZ's defaults are O=400 and E=30. The table shows open scores that would range from 347 to 569 and extend scores from 15 to 23. This suggests that BLASTZ slightly under penalizes gap open and over penalizes gap extend.

B.2 Simulated Sequence Pairs

We generated 135 simulated neutrally evolved sequence pairs covering the ranges of statistics observed in the 28-vertebrate alignments (section B.1). Substitution rates were modeled with the T92 evolution model (Tamura, 1992). T92 is a three-parameter

Table B.1 Genome-wide alignment probabilities observed in six vertebrates.

human GC	species	GC	p_{match}	p_{xition}	p_{xversion}	$\frac{p_{\text{xition}}}{p_{\text{xversion}}}$	p_{open}	p_{extend}	$\frac{p_{\text{mismatch}}}{p_{\text{open}}}$
low	macaque	31%	0.942	0.037	0.021	1.81	0.008	0.775	7.2
low	mouse	31%	0.673	0.185	0.142	1.30	0.023	0.791	14.0
low	dog	31%	0.758	0.146	0.096	1.52	0.019	0.766	12.8
low	opossum	31%	0.635	0.187	0.178	1.05	0.023	0.787	16.2
low	platypus	31%	0.640	0.188	0.172	1.09	0.021	0.778	16.8
low	chicken	31%	0.641	0.182	0.177	1.03	0.023	0.786	15.5
medium	macaque	41%	0.938	0.041	0.020	2.05	0.006	0.789	10.0
medium	mouse	41%	0.665	0.195	0.139	1.40	0.022	0.797	15.5
medium	dog	41%	0.744	0.159	0.097	1.63	0.017	0.768	15.0
medium	opossum	41%	0.654	0.189	0.157	1.20	0.019	0.787	18.4
medium	platypus	41%	0.658	0.186	0.155	1.20	0.017	0.781	19.7
medium	chicken	41%	0.681	0.172	0.147	1.17	0.016	0.780	19.7
high	macaque	53%	0.928	0.050	0.022	2.32	0.007	0.812	10.3
high	mouse	54%	0.664	0.196	0.141	1.39	0.023	0.812	14.4
high	dog	53%	0.717	0.170	0.113	1.51	0.020	0.793	14.0
high	opossum	56%	0.643	0.193	0.164	1.17	0.020	0.813	17.7
high	platypus	58%	0.624	0.192	0.184	1.04	0.021	0.813	18.2
high	chicken	57%	0.659	0.173	0.169	1.03	0.018	0.809	19.5

Table B.2 Alignment scores derived from observations in six vertebrates.

human GC	species	s_{AA}	s_{CC}	s_{max}	s_{open}	s_{extend}	$\frac{s_{\text{open}}}{s_{\text{max}}}$	$\frac{s_{\text{extend}}}{s_{\text{max}}}$	average non-gap	average gap
low	macaque	1.46	2.51	2.51	-8.72	-0.37	-3.47	-0.15	62.2 bp	4.4 bp
low	mouse	1.06	1.67	1.67	-7.27	-0.34	-4.35	-0.20	21.4	4.8
low	dog	1.19	1.97	1.97	-7.38	-0.38	-3.75	-0.20	26.4	4.3
low	opossum	0.96	1.60	1.60	-7.29	-0.35	-4.55	-0.22	22.2	4.7
low	platypus	1.01	1.58	1.58	-7.29	-0.36	-4.61	-0.23	23.3	4.5
low	chicken	0.99	1.61	1.61	-7.25	-0.35	-4.51	-0.22	21.7	4.7
medium	macaque	1.67	2.15	2.15	-9.23	-0.34	-4.30	-0.16	81.5	4.7
medium	mouse	1.21	1.49	1.49	-7.45	-0.33	-4.99	-0.22	23.2	4.9
medium	dog	1.36	1.70	1.70	-7.55	-0.38	-4.45	-0.22	29.4	4.3
medium	opossum	1.15	1.52	1.52	-7.57	-0.35	-4.96	-0.23	26.7	4.7
medium	platypus	1.21	1.49	1.49	-7.64	-0.36	-5.13	-0.24	28.8	4.6
medium	chicken	1.23	1.58	1.58	-7.72	-0.36	-4.90	-0.23	30.9	4.5
high	macaque	1.95	1.80	1.95	-9.26	-0.30	-4.75	-0.15	71.8	5.3
high	mouse	1.38	1.29	1.38	-7.45	-0.30	-5.39	-0.22	21.3	5.3
high	dog	1.54	1.38	1.54	-7.51	-0.33	-4.88	-0.22	24.7	4.8
high	opossum	1.35	1.26	1.35	-7.69	-0.30	-5.69	-0.22	24.7	5.3
high	platypus	1.41	1.11	1.41	-7.66	-0.30	-5.43	-0.21	24.3	5.4
high	chicken	1.47	1.23	1.47	-7.86	-0.31	-5.36	-0.21	28.5	5.2

model, fixing the G+C content distribution ($\pi_G = \pi_C = p, \pi_A = \pi_T = 1 - p$) and the instantaneous ratio of transitions to transversions (κ=transition rate / transversion rate). (B.1) shows the instantaneous rate matrix given p and κ. The third parameter is time t (equivalently, branch length), with the transition matrix $T = \exp(Qt)$. While this is the underlying model, we chose a different parameterization based on p, p_{match} and observed $p_{\text{transition}}$ / $p_{\text{transversion}}$ ratio.

$$Q = \begin{pmatrix} * & p & \kappa p & 1-p \\ 1-p & * & p & \kappa(1-p) \\ \kappa(1-p) & p & * & 1-p \\ 1-p & \kappa p & p & * \end{pmatrix} \qquad \text{(B.1)}$$

We allowed nine different sets of substitution parameters. G+C content was 31, 41 or 55% and p_{match} was 65, 80, and 95%. $p_{\text{transition}}$ / $p_{\text{transversion}}$ ratio was fixed at 1.5. For each of the nine sets we chose an appropriate rate and rate matrix of form (B.1) to produce a substitution matrix with the desired expected observed statistics.

We generated two collections of sequence pairs for each substitution set. One set uses the pair FSA model of section 5.1. The other generates gaps with a power law distribution. For the FSA model we allowed p_{open} to be 0.008, 0.016 and 0.024, and allowed p_{extend} to be 0.77 or 0.81. For power law gaps, we used the same values for p_{open} and allowed the power law exponent to be 1.5, 1.6 or 1.7. These latter values match the range observed in the 28-vertebrate alignment, but which are not shown in section B.1. Thus we have 54 sets for the pair FSA model, and 81 for the power law gaps model.

For each set, we generated 100 pairs, each with 800 homologous bp sandwiched between two independent 100 bp segments. The pairs were then concatenated into two sequences with 200 Ns acting as separators.

B.3 ENCODE Data

We constructed 35 test data sets from real genomic data, extracting data from seven encode regions (ENm001, ENm012, ENm014, ENr114, ENr132, ENr221 and ENr323) for five species (baboon, mouse, dog, opossum and chicken). The specific regions were chosen for the property that they are free of rearrangements when aligned with human for all five species. Following the methods of Chiaromonte et al. (2002), coding and repeat regions, as identified by RefSeq (Pruitt and Maglott 2001) annotations, were stripped from the human sequences. Repeat regions in the other five species were unmasked (made indistinguishable from other bases). Species aligning to human negative strand were reverse-complemented.

B.4 Syntenic Chromosomal Data

To facilitate testing on chromosome-to-chromosome alignments, we constructed a data set consisting of one 247Mbase sequence (human chromosome 1) and one 200Mbase sequence. The latter was constructed from six segments of mouse chromosomes 1, 3, and 4 that aligned well to human chromosome 1. The result simulates a pair of chromosome-length sequences with a common ancestral sequence, without rearrangements, at human-mouse evolutionary distance. Repeat masking information was retained.

B.5 Receiver Operating Characteristic

To measure the accuracy of a discovered alignment when the correct answer (reference alignment) is known, we use the Receiver Operating Characteristic (ROC). We follow Gribskov and Robinson (1996), in brief, plotting true positives against false positives and measuring the area under the curve. ROC gives a score between 0 and 1, with higher values indicating higher accuracy.

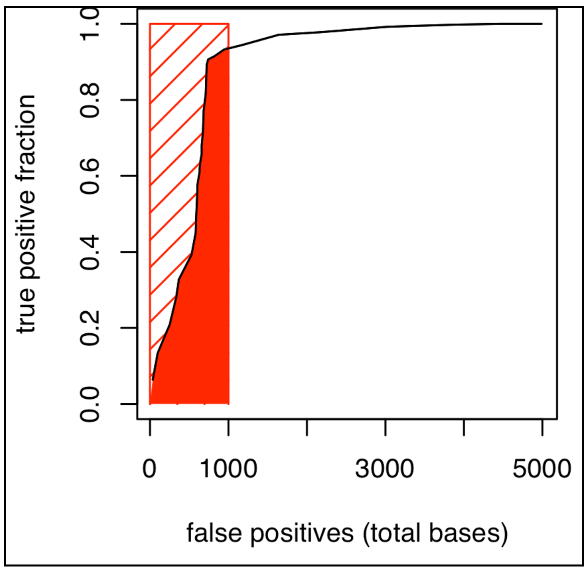
To compute the ROC score, we perform alignment with the alignment score threshold set low enough to assure we will discover enough false alignments—aligned segments that are not in the reference. We then compare discovered alignments to the reference and treat the aligner as a classifier with an adjustable score threshold. All ungapped columns in an alignment are assigned the score of that alignment. Columns that are also in the reference are true positives; columns that aren't are false positives. With a high enough score the classifier will identify nothing. As the score decreases, alignments will be 'discovered' and the true and false positive totals will increase.

Plotting true and false positives produces the ROC curve, giving a visual indication of how well the hypothetical classifier is performing. See figure B.1. A perfect classifier would discover all true positives before any false positives; the curve would hug the left and top edges of the unit square. A perfectly bad classifier would hug the bottom and right edges, discovering all false positives before any true positives. Thus the area under the ROC curve gives a measure of the quality of the classifier.

As pointed out in Gribskov and Robinson (1996), it is more meaningful to ignore any true positives that score lower than a certain number n of false positives. They define ROC_n to be the area under the curve when only the n highest scoring false positives are considered. Figure B.1(b) demonstrates this measure.

score	FP	TP	total FP	total TP
103	36	267	36	267
81	63	294	99	561
77	152	311	251	872
73	84	303	335	1175
66	33	195	368	1370
52	162	272	530	1642
51	53	244	583	1886
39	6	186	589	2072
38	8	118	597	2190
37	4	93	601	2283
36	2	60	603	2343
35	0	53	603	2396
34	10	60	613	2456
33	19	92	632	2548
31	1	46	633	2594
30	11	85	644	2679
29	15	68	659	2747
26	0	48	659	2795
24	0	50	659	2845
23	16	147	675	2992
22	8	196	683	3188
21	0	29	683	3217
20	5	39	688	3256
19	22	153	710	3409
18	5	99	715	3508
16	0	20	715	3528
15	7	173	722	3701
14	0	28	722	3729
13	19	62	741	3791
12	79	35	820	3826
11	126	75	946	3901
10	**224**	**48**	**1170**	**3949**
9	470	114	1640	4063
8	539	29	2179	4092
7	840	59	3019	4151
6	837	23	3856	4174
5	603	9	4459	4183
4	416	0	4875	4183
3	87	0	4962	4183
2	26	0	4988	4183

(a)



(b)

Figure B.1 ROC example. (a) Alignment results relative to a known reference alignment, shown as true (TP) and false positives (FP) by decreasing score. Row in bold shows cutoff for computation of ROC_{1000}. (b) Black line is ROC curve, plotting total TP vs. total FP. Red box shows calculation of ROC_{1000} (≈ 0.52), the ratio of the solid red area to the area of the red box.

Appendix C

Quantum DNA Techniques

One application of quantum alignment is to align a present day DNA sequence to a reconstructed ancestral sequence. The process breaks down into four parts— inferring a quantum sequence for the ancestor, reducing the sequence to an alphabet of 255 representative q-bases, creating a scoring scheme reflecting the similarity of any of these q-bases with DNA, and aligning the quantum sequence with a DNA sequence. Choosing an alphabet is of practical importance, reducing the space needed to represent the sequence to a byte per base, and allowing alignment scoring to be implemented with a small lookup table.

In sections C.1, C.2 and C.3 we synopsize solutions for ancestral inference, alphabet selection and scoring, due to Siepel (2005). Quantum alignment has already been discussed in section 4.4; here (section C.4) we describe the DNA ball generation algorithm required for seeding quantum vs. DNA alignment.

C.1 Inferring Ancestral Quantum Sequence

We are given a multiple alignment of DNA sequences for several species along with a phylogenetic tree topology and are to infer a quantum sequence for the ancestor.

The first step is to decide which bases were present in the ancestral sequence and which were not. The latter represent indels in the hypothetical alignment of the ancestor to the multiple alignment. We do this using Dollo parsimony (Farris, 1977), which, as applied here, says that the most parsimonious explanation for a particular column has no more than one insertion event. The single insertion rule implies that the insertion occurred at the branch leading into the last common ancestor of all non-gap leaves. In the special case that this is a child of the root, we cannot distinguish whether we had an insertion before the root, followed by a deletion between the root and one child, or an insertion between the root and the other child. We take the 'safe' approach and infer a base.

The second step is to estimate the nucleotide substitution model that best fits the data. It is assumed that all the locations have undergone the same evolutionary process, and that substitutions can be modeled by a time-reversible Markov process. The REV model (Yang, 1994) is used. The result is a substitution rate matrix Q, its stationary distribution π, and a length for each edge of the tree, measured in substitutions per site.

$$Q = \begin{pmatrix} * & \pi_C\alpha_1 & \pi_G & \pi_T\alpha_2 \\ \pi_A\alpha_1 & * & \pi_G\alpha_3 & \pi_T\alpha_4 \\ \pi_A & \pi_C\alpha_3 & * & \pi_T\alpha_5 \\ \pi_A\alpha_2 & \pi_C\alpha_4 & \pi_G\alpha_5 & * \end{pmatrix} \qquad \text{(C.1)}$$

The third step is to infer the ancestral sequence; to estimate the probability of each nucleotide at each non-gap position. For any column of the alignment, the rate matrix and branch lengths allow us to infer the nucleotides at ancestral nodes in the tree. This is performed with an algorithm due to Felsenstein (1981), modified by Siepel and Haussler (2003) to allow for gaps. For each node in the tree, it computes the probability of observing the leaves if that node contained a given nucleotide. More formally, let *u* be a node in the tree, b_u be the length of the branch from *u*'s parent to *u*, and L_u be the observations in the leaf nodes descended from *u*. Observations include the nucleotides A, C, G, and T as well as gaps. We can compute $\Pr(L_u \mid u = x)$ recursively by formula (C.2):

$$Pr(L_u|u=x) = \begin{cases} 1 & \text{if } u \text{ is a gap leaf} \\ 1 & \text{if } u \text{ is a non-gap leaf} = x \\ 0 & \text{if } u \text{ is a non-gap leaf} \neq x \\ \sum_{y,z} P(y|x,b_v)P(L_v|y)P(z|x,b_w)P(L_w|z) & \text{if } u \text{ has children } v \text{ and } w \end{cases} \qquad \text{(C.2)}$$

To get the desired quantum base for this column, we adjust for the background distribution and normalize:

$$q_x = \frac{\pi_x Pr(L_{\text{root}}|\text{root}=x)}{\sum_y \pi_y Pr(L_{\text{root}}|\text{root}=y)} \qquad \text{(C.3)}$$

When computed for each column of the alignment corresponding to an inferred non-gap, the result is a sequence of quantum DNA. Figure C.1 shows an example of the computation for a single column.

C.2 Choosing a Quantum Alphabet

The set of possible quantum bases is the probability simplex over four variables, where we have $q_A+q_C+q_G+q_T = 1$. It is advantageous to represent q-bases by a relatively small alphabet, and to do so we choose 255 points from the simplex and assign each q-base to a symbol representing the nearest such point. While the number of different q-bases that appear in any sequence must be finite, in practice there is a different q-base for every distinct column in the multiple alignment, the number of which grows with the number and distances between species. These are not evenly distributed about the simplex—most bases falling near a simplex corner (indicating near certainty for a specific nucleotide) or edge (indicating a choice primarily between two nucleotides).

$\Pr(L_u\|A)$	$\Pr(L_u\|C)$	$\Pr(L_u\|G)$	$\Pr(L_u\|T)$	obs'd	species
0	1	0	0	C	human
0	1	0	0	C	chimp
0	1	0	0	C	mouse
0	1	0	0	C	rat
1	0	0	0	A	rabbit
0	0	0	1	T	cow
0	0	0	1	T	cat
1	1	1	1	-	dog
0	0	0	1	T	hedgehog
0.000001	0.985579	0.000001	0.000009		human-chimp ancestor
0.000148	0.826509	0.000160	0.001461		mouse-rat ancestor
0.023284	0.024585	0.003395	0.002099		mouse-rabbit ancestor
0.011891	0.057052	0.017836	0.940401		cat-dog ancestor
0.000442	0.009154	0.000947	0.806805		cow-cat ancestor
0.000357	0.020443	0.000071	0.000138		human-mouse ancestor
0.000065	0.002450	0.000153	0.657229		cow-hedgehog ancestor
0.000001	0.000345	0.000001	0.000281		human-cow ancestor

(a)

Pr(A)	Pr(C)	Pr(G)	Pr(T)	species
0.002652	0.438488	0.001063	0.557797	human-cow ancestor

(b)

Figure C.1. Quantum inference example. (a) Probability of observed subtree given specific nucleotide at each node. (b) Probability of each nucleotide at root, adjusted for background distribution.

The method used is a clustering scheme that attempts to minimize the overall error in encoding the quantum sequence with the chosen alphabet[22]. The simplex is first carved into 175 cells, and an initial alphabet is made consisting of the centroids of each of these cells. Each observed q-base would be assigned to the single point in its box. The total error in each box is measured and whichever box has the largest error is granted a second point. K-means is used to determine a good placement of the two points in that box so as to minimize error. The process is repeated—the box with the largest error is granted an extra code—until all 255 codes have been assigned.

The above method has some potential drawbacks. Every box is assigned a code even if there are no q-bases observed in that box. These codes could be assigned to other boxes to reduce error. Further, the method does not guarantee that a q-base will be assigned to the nearest code.

The importance of the alphabet selection to the results is an open. While it is likely that some alphabets would perform poorly, it is not yet clear to what extent it is necessary to tune the alphabet to the specific alignment problem, and whether the gains in alignment from optimizing the alphabet are worth the computation spent on the task.

C.3 Quantum versus DNA Scoring

The following derivation is due to Haussler (2005). Suppose we observe the situation shown in figure C.2. *d* is a base in a DNA sequence, *q* a base in a quantum sequence, and *t* the evolutionary distance between them. Assume we know *t*, and that we know *q* (i.e. we know $q_x = \Pr(q = x), x \in \{A,C,G,T\}$). We also assume we have some model that estimates $\Pr(d \mid q = x, t)$ and the stationary distribution $\Pr(d = x)$.

We have two hypotheses, $H1$, that d and q are related, and $H0$, that they are not. We can compute the probability of this observation for both hypotheses:

$$\begin{aligned} P(d,q|H1) &= P(d|q,t)P(q) && \text{(Bayes' rule)} \\ P(d,q|H0) &= P(d)P(q) && \text{(independence, stationary distribution)} \end{aligned} \tag{C.4}$$

and from this we can compute a log odds score:

[22] The error measure used is symmetric relative entropy (symmetric Kullback-Leibler distance), but other measures might perform as well.

$$\begin{aligned} S(d,q) &= log\frac{P(d,q|H1)}{P(d,q|H0)} \\ &= log\frac{P(d|q,t)P(q)}{P(d)P(q)} \\ &= log\frac{\sum_{x\in\{\mathrm{A,C,G,T}\}} q_x P(d|q=x,t)}{P(d)} \end{aligned} \tag{C.5}$$

When q is in a quantum sequence inferred from a multiple alignment of its descendants, the model is a substitution matrix estimated from the alignment (the shaded triangle in figure C.2), and the stationary distribution is from the same model. We're making the implicit assumption that substitution rates in the shaded triangle are the same as those in the larger evolutionary context that contains *d* and q.

C.4 DNA Ball Generation

The ball generation algorithm, Generate-DNA-Ball, emits all DNA words that score at least T_{score} when aligned, without gaps, to quantum word q of length w. It is a simple depth-first search with pruning of low-scoring prefixes. In lieu of recursion, the generated word is used as a stack. Performance characteristics of this algorithm have not been measured. In practice the performance has not been noticeably detrimental for word sizes up to 13. Run time could potentially be improved by changing the order in which the word is scanned, visiting locations with a wider score discrepancy first.

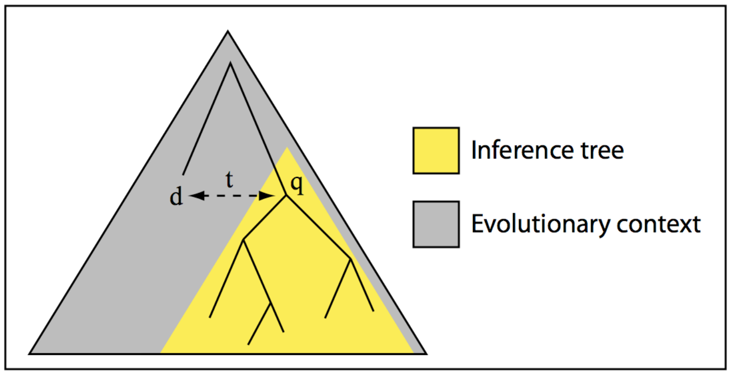


Figure C.2. Quantum scoring context. *t* is the evolutionary distance between DNA base *d* and quantum base q. q is inferred from descendent tree in yellow.

```
Generate-DNA-Ball(q,w,$T_{score}$)
9   $lower_w = T_{score}$
10  for i = w-1 downto 1
11      $lower_i = lower_{i+1} - \max_{x \in \{A,C,G,T\}} s(x, q_{i+1})$
12  i = 1
13  $dword_1$ = "$"
14  score = 0
15  while i > 0                          while we haven't considered all words…
16      if $dword_i$ ≠ "$"               subtract score for previous symbol
17          score = score – s($dword_i$, $q_i$)
18      $dword_i$ = Next($dword_i$)        try next symbol
19      if $dword_i$ = "$"               all symbols tried; backtrack
20          i = i-1
21          continue
22      score = score + s($dword_i$, $q_i$)   add score for this symbol
23      if score < $lower_i$             score too low—prune (go undo this symbol)
24          continue
25      if i < wordLen                   word incomplete, move to next position
26          i = i+1
27          $dword_i$ = "$"
28          continue
29      emit dword
```

$$\text{Next}(x) = \begin{cases} \text{“A”} & \text{if } x = \text{“\$”} \\ \text{“C”} & \text{if } x = \text{“}A\text{”} \\ \text{“G”} & \text{if } x = \text{“}C\text{”} \\ \text{“T”} & \text{if } x = \text{“}G\text{”} \\ \text{“\$”} & \text{if } x = \text{“}T\text{”} \end{cases}$$

Appendix D

Linear Inference Techniques

While it is not often mentioned in discussion of (1.1)-based alignment, computing the score for a given alignment is a linear operation. The score is the product of a feature vector, giving the count of each feature (the sixteen nucleotide pairs, gap open and gap extend), and the score vector (s_{AA} through s_{TT}, s_{open} and s_{extend}). (see figure 1.1(c) for an example). From this viewpoint it is natural to think of linear supervised learning techniques to find a good score vector from sample alignments.

While the author had little success using these techniques for local alignments in large sequences, others have had success using them for global alignments on relatively short sequences, such as proteins. We include a cursory description of these techniques here.

D.1 Linear Discriminator

If we have samples of both positive ($\mathcal{A}^+$) and negative ($\mathcal{A}^-$) alignments we would like to find a score vector S that will discriminate between the two sets. That is, for all $A \in \mathcal{A}^+$ and $B \in \mathcal{A}^-$, $A \cdot S > B \cdot S$. Linear discrimination (LD) is a well-studied problem, and we will not go into details here; we only mention that among the difficulties in applying LD to alignment scoring are the size of the problem and the fact that complete discrimination is unlikely in practice. Joachims (2003) presents an algorithm to address these issues, with some constraints.

The author's own efforts were unsuccessful. Positive samples were generated by aligning two sequences from the HOXD region of human and mouse (about 1 Mbase), using a starting scoring model. Negative samples were generated by aligning human to the reverse (not reverse-complement) of mouse. Using libsvm (http://www.csie.ntu.edu.tw/~cjlin/libsvm), an off-the-shelf LD learning program (support vector machine with a linear kernel), two major problems were encountered. First, gap scores were not properly constrained; often the 'best' solution (from the standpoint of SVM) rewarded gaps rather than penalize them. Second, the size of the problem (≈60K positives, 35K negatives) was vastly larger than the program could keep track of in memory.

D.2 Inverse Alignment

Another approach is to treat the problem as a linear constraint problem, an idea due to Kececioglu and Kim (2006). Though the solution could be extended to include negatives, here we will describe it when only positive samples are available. The goal is to find a score vector that (1) makes each sample's alignment optimal (or as close as possible), (2) properly constrains gap scores, and (3) maximizes the margin.

The basic idea is that a particular alignment's *optimality region*, the region of scoring space in which it is optimal, is convex with linear boundaries. We do not, however, have to identify all the boundaries. We only have to find a point in the optimality region. To find a score vector that makes all alignments optimal, we must find a point in the intersection of all optimality regions. In practice the intersection will be empty. So an additional sub-optimality control $\epsilon \geq 0$ is added that requires each alignment only score within some fraction of optimal. More formally, for any alignment A in $\mathcal{A}^+$ and any other alignment B of the same two sequences, we require that

$$(1+\epsilon)\text{score}(A) \geq \text{score}(B) \qquad \text{(D.1)}$$

Kececioglu and Kim (2006) give an ingenious algorithm that solves the problem quickly, finding the smallest possible ϵ value (to any desired accuracy) as well as a point that meets all three requirements. The algorithm makes use of linear programming, using an off-the-shelf linear programming package such as the GNU Linear Programming Kit (GLPK, http://www.gnu.org /software/glpk). A key realization is that we can determine whether we satisfy (D.1) for all B by determining whether (D.1) holds for an optimal alignment B^*.

Here we describe the algorithm as it applies to finding the two gap scores, open and extend, with substitution scores fixed. We assume that ϵ is fixed (initially at 0). We begin with a set of (at most) $|\mathcal{A}^+|+2$ linear constraints. Open and extend scores must be negative, constraining us to the lower left quadrant of the two-dimensional score space. We can then add one linear constraint for each alignment in $\mathcal{A}^+$, to force it to score at least zero[23]. Using linear programming, we find a point that satisfies all constraints and maximizes margin (or some other desirable feature). If no such point can be found, the intersection of the constraints is empty; this means there is no solution for this value of ϵ.

Otherwise, the solution point is a score vector potentially satisfying criterion (1). We use this vector with each sample, in turn, finding an optimal alignment (the *alternative*) of its two sequences. If for every sample the original alignment scores as well as the alternative found (with consideration for ϵ), then we have a solution. If some sample fails, it is easily converted into an additional linear constraint that will make the original score as well as the alternative (again, with consideration for ϵ). We add this constraint to the set and solve again. Eventually we will either find a solution or reach a point where the constrained space is empty, indicating that ϵ is not viable. The minimum value of ϵ can be found by a simple binary search over a reasonable search interval, such as (0,1)[24].

The author attempted to use this as part of an iterated score inference scheme, using substitution scores inferred as per Chiaromonte et al. (2002). However, as a function mapping one set of gap scores to another, this process behaves badly. Small changes in the input gap scores led to vastly different output scores, seemingly without rhyme or reason. This would produce very poor convergence behavior.

We feel the method has promise, though, and deserves further study. Kececioglu and Kim (2006) have shown some success in inferring protein alignment scores.

---

[23] Allowing the possibility that an optimal alignment has a negative score would disallow sub-optimality. If score(optimal alignment B)<0, (D.1) could be satisfied only when A is also optimal and $\epsilon=0$.

[24] A solution for $\epsilon=1$ means all alignments score at least half as high as an optimal alignment.

Appendix E

Seed Packing

In this section we discuss how redundancy is removed from the seed word, allowing it to be used as an index for the seed word position table (section 4.2, figures 4.1 and 4.2). The seed word covers L nucleotides, which must be reduced to a 2W-bit index. The scheme presented here reduces the number of operations necessary for packing seeds. This has a negligible effect on the overall performance of LASTZ—seed packing is only used during the seeding stage, which is overwhelmed by the gapped alignment stage. Nonetheless, we include this discussion because in other computational contexts this reduction may be effective.

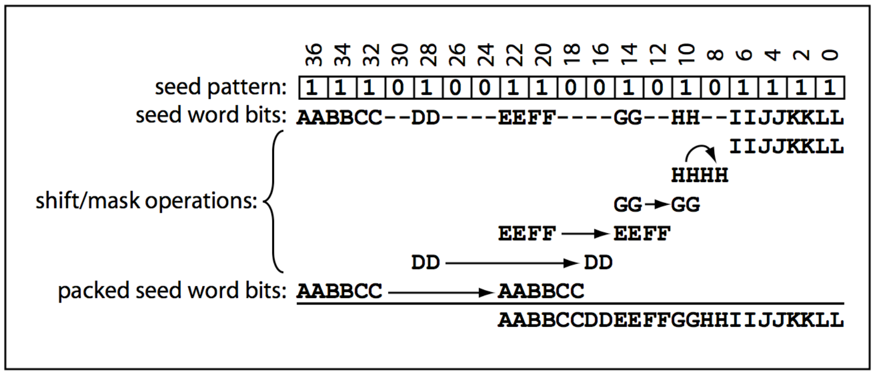
Nucleotides are encoded in the seed word as a concatenation of two bit fields, encoding A as `00`, C as `01`, G as `10` and T as `11`. This particular encoding was chosen so that it is easy to determine if a base is a purine (A or G, second bit `0`) or pyrimidine (C or T, second bit `1`). Moreover, when comparing two aligned bases we can quickly determine whether they are a transversion (second bit different) or not (second bit the same). The seed word's L nucleotides are thus encoded as a 2L-bit value. Similarly, we encode the seed pattern two bits per position; matches are encoded as `11`, don't-cares as `00`, and transition-matches as `01`.

The simplest scheme for packing the seed word would simply remove the bit positions that are zero in the pattern, shifting bits to the right to fill the holes while retaining the order of bits in the unpacked word (figure E.1(a)). Maintaining bit order is unimportant, though, and by sacrificing it we can (in nearly all cases) reduce the number of operations needed for packing. Figure E.1(b) gives an example.

This packing method can support T-matches at no additional cost. By encoding a T-match as bit pair `01`, the packed seed retains for that position only the bit distinguishing between purine and pyrimidine. If two seed words have the same bit in that position, they either match or are a transition. Figure E.1(c) shows an example of packing a seed containing a T-match.

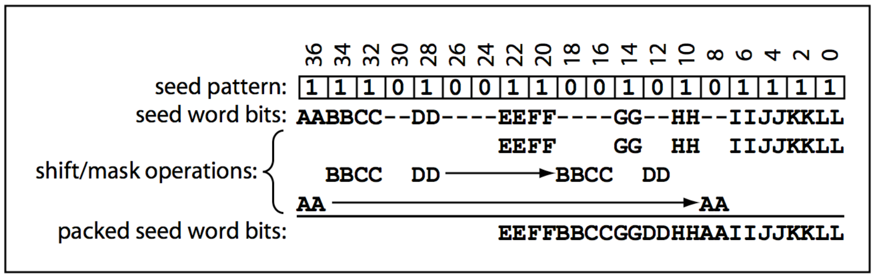
The packing of any seed pattern can be performed by a series of shift-and-mask operations. Given a seed, how can we find the best packing? Since LASTZ allows the user to specify any seed, it is imperative that it can quickly find an optimal or near-optimal packing. LASTZ includes a greedy algorithm for this purpose. Starting with the encoded seed pattern and initial goal pattern (the least significant 2w bits), it finds the shift-and-mask operation that will cover the most bits in the goal. It then removes those from both the pattern and the goal, and tries again. The result is never worse than the simple scheme, and is usually much better. For all but nine of the ≈20,000 possible 12-of-19 seeds, the greedy algorithm finds a packing with fewer than five operations.

(a)



36 34 32 30 28 26 24 22 20 18 16 14 12 10 8 6 4 2 0
seed pattern: 1 1 1 0 1 0 0 1 1 0 0 1 0 1 0 1 1 1 1
seed word bits: AABBCC--DD----EEFF----GG--HH--IIJJKKLL
shift/mask operations:
IIJJKKLL
HHHH
GG GG
EEFF EEFF
DD DD
packed seed word bits: AABBCC AABBCC
AABBCCDDEEFFGGHHIIJJKKLL

(b)



(c)

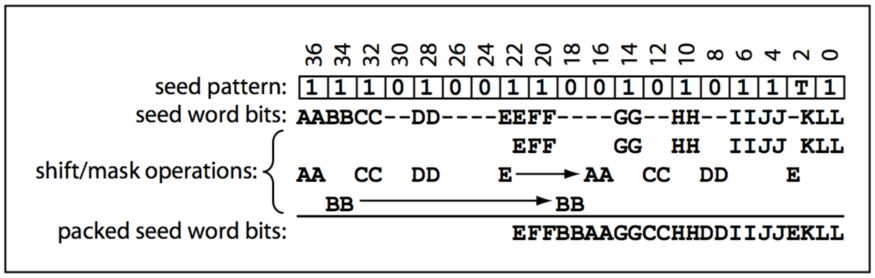


Figure E.1 Seed packing. (a) Straightforward packing of 12-of-19 seed pattern's 38 bits to 24 bits requires six shift-and-mask operations. (b) Same seed can be packed with only three shift-and-masks. (c) Changing one position to a T makes an $11\frac{1}{2}$-of-19 pattern; three shift-and-masks are still sufficient to pack to 23 bits.

Table E.1 Comparison of greedy seed packing to optimal for all 12-of-19 seeds.

optimal operation count	greedy operation count	number of seeds
2	2	383
	3	463
	4	**18**
3	3	9865
	4	8465
	5	**8**
4	4	244
	5	2

Calculation of an optimal packing is a more computationally expensive undertaking, searching for a shortest path of operations mapping the pattern to the goal[25]. A meet in the middle search reduces the time and space drastically, but for many seeds, especially those containing T-matches, it is still very costly. Table E.1 shows how well the greedy algorithm performs, relative to optimal, for all 12-of-19 seeds containing only matches and don't-cares. For 54% of the seeds it finds an optimal packing, and for all but 0.13% of the seeds it is within one of optimal (26 seeds, shown in bold).

Unfortunately, allowing a general seed pattern incurs a run-time cost. A table driven loop, performing one shift-and-mask each time through the loop, is no match for the same series of operations hard-coded and optimized by the compiler. Figure E.2

```
uint32_t pack_seed (uint64_t word) {
    return ( word        & 0x00F0CCFF)
         | ((word >> 16) & 0x000F3000)
         | ((word >> 28) & 0x00000300);
}
```

Figure E.2 Hard-coded seed packing. Machine-written C routine implementing the seed packing of figure E.1(b).

25 Care must be taken to ensure that no bit is "moved" more than once; multiple moves would prevent parallelization at runtime.

shows a C routine implementing the packing of the default 12-of-19 seed using the operations discovered by the greedy algorithm (i.e. these are the same operations used in LASTZ for this seed). Creating the seed position table for a 100Mbase sequence took 9.0 seconds when the operations in figure E.1(b) were performed by a general loop. Using the hard-coded routine of figure E.2 reduced this to 7.7 seconds. For comparison, a hard-coded routine implementing the operations in figure E.1(a) took 8.3 seconds. While the reduced operation packing is about a 7% improvement, it represents a miniscule portion of the overall alignment time.

The LASTZ distribution includes a program to convert a seed pattern to such a routine, which can then be compiled as part of LASTZ. A better solution would be to compile it to a dynamically linked module, and allow specification of the module on the LASTZ command line.

Vita

Robert S. Harris is a Ph.D. candidate in Computer Science and Engineering at the Pennsylvania State University. He received his Bachelor's degree in Applied Math from Carnegie-Mellon University in 1979 and his Master's degree in Computer Science from the University of Tennessee in 1993. Following a 20-year career in industry as a software engineer and game designer, he entered the Ph.D. program at Penn State in 2003 and shortly thereafter joined Webb Miller's lab at the Penn State Center for Comparative Genomics.