A Fast Genomic Dictionary

by

Valentin I. Spitkovsky

S.B. in Mathematics

Massachusetts Institute of Technology (1999)

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Computer Science and Engineering and

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 22, 2000

Author . . . .
Department of Electrical Engineering and Computer Science
May 22, 2000

Certified by . . . .
Bonnie A. Berger
Samuel A. Goldblith Associate Professor of Applied Mathematics
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

A Fast Genomic Dictionary
by
Valentin I. Spitkovsky[1]

Submitted to the
Department of Electrical Engineering and Computer Science

May 22, 2000

In Partial Fulfillment of the Requirements for the Degrees of
Bachelor of Science in Computer Science and Engineering and
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

With the completion of this thesis comes a new tool for rapidly finding common regions between strings. The dictionary provides a fast and fully automated approach to locating the longest matches at every position in a set of input sequences to the sequences stored in a database. While the applications discussed in this work all stem from the realm of computational biology, we place no restrictions on the sequences at hand: all algorithms described continue to function as intended while the dictionary operates on strings drawn from an arbitrary alphabet. We have been actively using the fast genomic dictionary in our lab even as the tool was being written, tested, revamped, and optimized. In practice, the dictionary has already proved itself in several tasks worthy of note: 1) gene annotation / recognition and exon prediction — the matches provide useful information pertaining to locating common segments between exons, alternative splice sites, and frequency data of long tuples, establishing the basis for both homology determination and statistical approaches; 2) identification of related sequences — the dictionary approach dodges many shortcomings inherent in the de-facto standard alignment-based algorithms like `BLAST`, emerging as a valuable complementary tool; 3) repeat-masking — the technique of matching fragments of the input to databases of repeats provides a speedy though slightly coarse alternative to alignment-based repeat maskers like `RepeatMasker`; 4) data mining — the dictionary offers a quick means of extracting clean data from the otherwise polluted biological databases, for instance allowing us to construct highly precise frame- and orientation-tests almost too easily; 5) visualization — the dictionary lends itself to insightful ways of communicating data. Fast and light, the dictionary should be extremely useful to anyone dealing with large databases of strings and especially to researchers actively involved in the Human Genome Project.

Thesis Supervisor: Bonnie A. Berger
Title: Samuel A. Goldblith Associate Professor of Applied Mathematics

[1]Supported in part by the Fannie & John Hertz Foundation (the author is a 1999 Hertz Fellow), the National Institutes of Health (NIH) Genome Training Grant, the Merck Fund, the Undergraduate Research Opportunities Program (UROP), as well as the author's parents.

To my parents.

I could write out my words of praise, love, and respect for them.
But the length of such disclosure would make this thesis pale in comparison.

In loving memory.

Joseph Perelman, 1926 – 1991.
Lydia Perelman, 1931 – 1993.
Alex Perelman, 1963 – 1998.

Acknowledgments

I would like to thank my supervisor Bonnie A. Berger for continued support and encouragement, as well as Daniel J. Kleitman who suggested the notion of a dictionary to begin with. My thanks go out to Lior Pachter and Serafim Batzoglou, the graduate students who welcomed me to the group and helped me plunge into computational biology. I owe quite a bit to MIT and the wonderful professors whose classes I was lucky enough to take over the course of the last five years; I am also grateful to a few high school teachers — most prominently Linda K. Dunnigan of Lafayette High School and Richard I. Purman of the New Horizons Governor's School for Science and Technology — for early guidance.

This work has been explicitly funded by my parents, the Undergraduate Research Opportunities Program (UROP), the Merck Fund, the National Institutes of Health (NIH) Genome Training Grant, and the Fannie & John Hertz Foundation. I would like to take this opportunity to acknowledge various scholarships and funds which helped defray the cost of my endeavor over the last five years. These are the Tau Beta Pi National Engineering Honor Society Fellowship, the Harold Levy Memorial Scholarship, State Farm Companies Foundation Exceptional Student Fellowship, Advanced Micro Devices (AMD) Scholarship, the Armed Forces Communications and Electronics Association Educational Foundation General John A. Wickham Scholarship, Microsoft Technical Award, the Armed Forces Communications and Electronics Association Lt. General Albert J. Edmonds Flagship Scholarship, Tandy Technology Scholarship, and the Robert C. Byrd Scholarship.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The Human Genome Project

Originally begun in 1990, the U.S. Human Genome Project is coordinated by the U.S. Department of Energy and the National Institutes of Health. Conceived as a 15-year endeavor, rapid technological advances have accelerated the project to an expected completion date of 2003. Project goals are to 1) *identify* all the estimated 80,000-100,000 genes in human DNA; 2) *determine* the sequences of the 3 billion chemical bases that make up human DNA; 3) *store* this information in databases; 4) *develop* tools for data analysis; and 5) *address* the ethical, legal, and social issues (ELSI) that may arise from the project [12]. Once the genes are identified and their nucleic sequences are determined, researchers will have to tackle new and more difficult problems to understand what it is that these genes do. To get a feel for just how difficult this problem is, consider the modern central biological dogma (see figure 1-1) [13]:

1. genes (DNA) are transcribed into RNA by the enzyme RNA polymerase;
2. RNA transcripts are subjected to post-transcriptional modification and control;
3. mRNA molecules are translated by ribosomes;
4. newly synthesized proteins are often modified after translation (post-translation);
5. the protein carries out its function.

Not all DNA is expressed as protein. Sections of non-expression are termed introns. Introns are excised by splicesomes during the pre-mRNA phase, and are not found in the final mRNA product. Exons are all that remain of the original DNA message upon completion of transcription. Clearly, even knowing all the nucleotides that make up a gene is not enough to predict a protein that it codes for.

As it turns out, what most people refer to as *the* Human Genome Project is actually just phase one of the mission. Among other things, it will result in a huge dataset of human DNA. Unfortunately, because most of that DNA forms intronic or intergenic regions, 99%

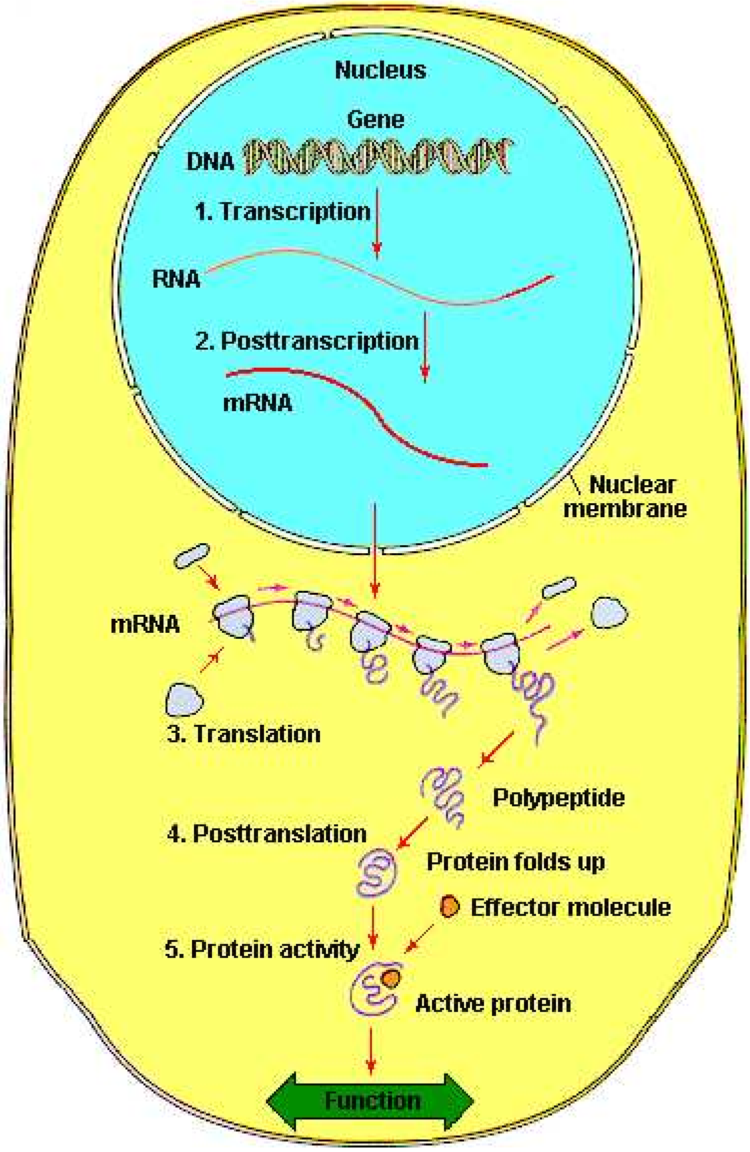


Figure 1-1: From Genes to Proteins.

of the data may not be very useful. Essentially, looking at a random position in the dataset yields a 1% chance of considering a region that is somewhat interesting. The rest probably does not constitute junk or garbage, but at the same time does not provide particularly interesting information to biologists. There is no accepted biological theory for why we have introns; some species do not. Phase two of the mission will mean going back to the lab and using more sophisticated, more expensive, more time-consuming techniques to identify the genes and annotate their structure, zooming in on the interesting regions. It is projected to take 100 years.

1.2 Gene Recognition

In light of these complications, gene recognition emerged as a subfield of computational biology to confront the challenge. Researchers in this area attempt to design algorithms and heuristics which would eventually compile to efficient computer programs for variations of the following problem: given a large sequence of genetic material (read: a string of hundreds of thousands of nucleotides)

1. locate the promoter regions (these occur just before the genes);
2. identify the genes;
3. identify the exons and the introns within each gene;
4. concatenating the coding exons together, predict the initial amino acid sequences;
5. predict the proteins made by each gene.

1.3 Gene Annotation

Of particular interest in the gene recognition problem is the subproblem of gene annotation. Suppose that we had access to a reliable promoter detector. Recall that promoters are short regions of DNA that occur close to the beginning of the gene. They are the chemical signals which alert the biological machinery that there is a gene to be expressed and specify how much of the protein should be made. Having identified the promoters in our large chunk of DNA, we could then split that DNA into shorter segments, each beginning with one promoter region and ending just before the next. We would then be confident that the resulting segments of DNA contain exactly one gene.

The next task would be to annotate that gene: to specify which segments are introns (to be cut out) and at which point the cell begins and ends its translation of the concatenation of the remaining segments (exons). Of tremendous help in this task are several biological facts:

1. translation begins after an `ATG`;
2. translation ends at the first stop codon (one of $\{\texttt{TAA}, \texttt{TAG}, \texttt{TGA}\}$, see table 1.1);

3. introns start with a `GT`;
4. introns end with an `AG`.

1	Amino Acid				3
A	K	T	R	I	A
A	N	T	S	I	C
A	K	T	R	M	G
A	N	T	S	I	T
C	Q	P	R	L	A
C	H	P	R	L	C
C	Q	P	R	L	G
C	H	P	R	L	T
G	E	A	G	V	A
G	D	A	G	V	C
G	E	A	G	V	G
G	D	A	G	V	T
T	-	S	-	L	A
T	Y	S	C	F	C
T	-	S	W	L	G
T	Y	S	C	F	T
	A	C	G	T	
	2				

Table 1.1: High level view of DNA to protein translation.

Fields 1, 2, and 3 specify the first, second, and third nucleotide of the DNA triplet (codon). The dash (-) stands for very special triplets called *stop codons* which instead of coding for an amino acid (see table 1.2) terminate the translating process.

Index	Symbol	Name
1	A	Alanine
2	C	Cysteine
3	D	Aspartic Acid
4	E	Glutamic Acid
5	F	Phenylalanine
6	G	Glycine
7	H	Histidine
8	I	Isoleucine
9	K	Lysine
10	L	Leucine
11	M	Methionine
12	N	Asparagine
13	P	Proline
14	Q	Glutamine
15	R	Arginine
16	S	Serine
17	T	Threonine
18	V	Valine
19	W	Tryptophan
20	Y	Tyrosine

Table 1.2: The Twenty Amino Acids.

This limits the number of possible annotations or "parses" of a gene substantially, but does not reduce the solution space to a single crisp answer. As a result, there is a need for good tools for evaluating or scoring a parse. Armed with an arsenal of such tools, a gene annotation algorithm could intelligently decide between parses and predict the "best" one. Even if the level of gene annotation were not good enough for the gene recognition problem, it would still be extremely useful. One of the main challenges facing biologists today is the accurate annotation of newly sequenced genomic data. If the annotating algorithm could latch on to more than just a handful of the most basic signals, it would save a lot of hours of human work. Available for this task are extremely large databases of proteins (*e.g.* OWL [19]), expressed sequence tags or snippets of coding regions of DNA (*e.g.* `dbEST` [18]), and much smaller databases of annotated genes (*e.g.* USC [9]).

Exons, the coding regions of DNA, tend to endure high selective pressure to not mutate much. As a result, large pieces of coding DNA seem to be conserved between various proteins and even among species. Therefore, when faced with a new sequence, if one could find segments in it that match reasonably long regions in databases of known exons and proteins, one could then safely label these portions of the new sequence as "exon," pruning the search space of parses and yielding hints as to the origin and function of the sequence. At the very least, the matches could be promoted to the statistical analyzers and artificial intelligence routines in gene recognition programs as hints that certain regions should be marked as exons in the likely parses.

Introns, the non-coding regions between exons which get spliced out, are under much less of a selective pressure to preserve themselves: most changes in introns go unnoticed. Therefore, finding a reasonably long match between an exon and an intron is highly unlikely. Still, introns tend to possess areas called repeats which are long sequences of DNA that occur much less frequently in exons. Presence of a region similar to a repeat strongly indicates that the region belongs to an intron. Similarly, when viruses wedge themselves in host DNA, they often go unnoticed if the insertion happens to land in an intron. Since introns are cut out, most likely, the change does not affect the host. Sometimes the altered DNA, faithfully reproduced by the cell's machinery, makes it into the host's offspring. In that case, the virus may be preserved in that specie's genome. Clearly, presence of viral DNA in a non-viral genome indicates that the region in question most likely is part of an intron.

1.4 Dictionary

We have developed a fast and fully automatic tool for looking up genomic information. Qualitatively, given a brand new sequence, the dictionary answers the question "Have we even seen anything like it before?" Quantitatively, the answer to that question consists of a list of all matches (longer than a cutoff value) between the substrings of the input sequence and the substrings of all the sequences forming the dictionary's basis.

All of the dictionary code is in the C++ programming language, written and thoroughly tested under the Linux operating system. We designed and optimized the dictionary to utilize the resources of the host computer as efficiently as possible. The alphabet over which the dictionary interprets sequences remains entirely user-defined and unrestricted. Combined with the lack of limits on lengths, numbers, and types of sequences forming the dictionary, the uses for it are virtually endless: the sky is the limit.

To address for instance the gene annotation problem, one could build dictionaries out of exonish regions, proteins, repeats, viral strands, etc. With such equipment, when confronted by a new sequence, the biologist could rapidly blast it against the databases of sequences s/he cares about. Immediately, the researcher would have access to insights for regions that seem to match certain characteristics. For instance, having found the regions matching proteins, the biologist could tabulate statistics on the numbers of matches within windows of particular lengths, actually pull out the sequences and take a closer look at the matching regions, and most importantly take into account the sequences' descriptive names. Proteins' accession codes tend to specify the species of origin and mention that protein's function. With the dictionary, researchers can efficiently reap insights which typically take graduate students in biology months, if not years, to discover. Combined with the capabilities for visualizations, the dictionary should become indispensable to the biologists in the field, dealing with newly sequenced genomic data.

Our group's gene recognition program latches on to the matches produced by the dictionary as pillars on which to base the likely parses, rapidly slashing the search space. We have also exploited the dictionary's capabilities to identify pseudogenes, mask repeats, and to mine the data hidden in the mostly disorganized biological databases. These applications suggest that the dictionary could significantly impact advancements in computational biology. Moreover, the dictionary delivers a completely general suite of string-matching algorithms. We hope to find them applied fruitfully not only in the realm of computational biology in light of the Human Genome Project, but also in all other disciplines dealing with very large databases of strings.

Chapter 2

Overview

2.1 Brief History

This work began when the author joined Bonnie A. Berger's Computation & Biology Group at the MIT Laboratory for Computer Science (LCS). Having just completed the junior year, the author was hired into the Theory of Computation Group in May of 1998 to work on the Gene Recognition Project under the Undergraduate Research Opportunities Program (UROP). Bonnie A. Berger, the research supervisor, pointed out the following problem: the database of expressed sequence tags `dbEST` contains several million snippets of DNA, each about a thousand characters in length; it would be nice to be able, given a long input sequence, to quickly extract a few hundred similar sequences from `dbEST`. The manageable set of similar sequences would register for closer examination: regions in the input sequence similar to snippets from `dbEST` promised to be the exons of a gene.

Daniel J. Kleitman of the Applied Mathematics Department also mentored the project. He suggested an initial stab at the problem: consider all of the possible t-tuples of DNA, for a positive integer t. . Given a sequence of DNA, it is easy to make a list of all the t-tuples it contains. If one could construct a inverse table — one that given a particular t-tuple could return the list of sequences that contain it — then it would be possible to simply pull out the sequences hit most by the tuples from the input. The length of the tuple shouldn't be too short: at the extreme case of $t = 1$, pretty much every sequence will share all 1-tuples with all other sequences, yielding very little information. At the same time, t couldn't be too long; otherwise, there may not be enough statistical frequency data for all tuples. Since `dbEST` contained on the order of a billion tuples, we set $t = 11$ for a total of $4^{11} = 2^{22} = 4\text{M}$ tuples.

The initial task of inverting a table seemed trivial yet daunting at the same time. Consider a zero/one matrix, with the truth value in position (s, t) indicating the absence/presence in sequence s of tuple t, which is just an integer between 0 and $4^{11} - 1$, inclusive. Inverting the table corresponds to taking the transpose of the matrix. It seems the matrix could be simply written horizontally and then read out vertically. The problem is that the matrix is huge. With millions of sequences and millions of tuples to consider, the number of elements in the matrix would be a whopping trillion.

The initial gut reaction framed the problem as an engineering challenge. Taxing the system to the max, the author hacked up code that solved the problem brute-force. Tackling the issue head-on, we made the observation that once built, the dictionary would be invaluable. Therefore, it may make sense to invest quite a bit of time into the building process. Noticing that different pieces of the matrix could be transposed in parallel, we engaged four computers in the task; each one ran highly optimized code, bit-packed the matrix, and had the maximum allowed number of files — just under 2^{16} — open for writing. It took more than a week — about a month's worth of sequential time — to build the dictionary. But the data we ended up with actually worked properly and made the utility of the dictionary clear as day.

Recovering from the adrenaline rush of the past week, the author approached the problem in a cooler algorithmic style. Clearly, the matrix must be very sparse. There are four million entries per sequence, while sequences average only about a thousand characters in length. We latch onto this observation to significantly improve the running time and the space required. We begin with the pairs (s, t) ordered by the first element, the sequence index. We would like to list these pairs ordered by the second element, the tuple index. Put this way, the challenge reduces to a textbook sorting problem. Since both s and t are non-negative integers, *RadixSort* does the trick with time and space required both linear in the number of pairs — the number of non-zero entries in the matrix. In practice, there is the complication that there may not be enough memory to accommodate all of these pairs. We chose to sort groups of pairs large enough to barely fit into memory, writing them out to disk and later merging the resulting segments. We could now build dictionaries in just a matter of hours using a single computer.

The main part of the dictionary consisted of just two files which allowed for efficient queries. Let (t_i, s_i) be the ith pair in the set sorted by t with collisions resolved by s. We maintained a position file `.pos` of $|(t_i, s_i)|$ words, with

$$\texttt{.pos}[i] = s_i.$$

In addition, we maintained an index file `.idx` of $4^{11} + 1$ words, with $\texttt{.idx}[0] \equiv 0$ and

$$\texttt{.idx}[i] = \texttt{.idx}[i-1] + |\{(i-1, s) \mid \forall s\}|\,.$$

In other words, $\texttt{.idx}[i]$ points to the offset in `.pos` where the list of sequences containing tuple i begins. Asymptotically, getting this list requires only a $\Theta(1)$ overhead: simply position the file pointer for `.idx` to the ith word, read the two values a and z, position the file pointer for `.pos` to the ath word, read the indices of sequences containing tuple i until the file pointer reaches the zth word. In practice, files may be fragmented. For the Linux inode-based file system, we expect the overhead to be logarithmic in the size of the file.

2.2 From Heuristics to Guarantees

Having the machinery in place for quickly determining which sequences contain a particular tuple brought us a bit closer to our goal, but not much. We were faced with the challenge of somehow pulling out the best few hundred. Having to create a scoring function seemed to invite a heuristic plague.

For each sequence in the database, we could tell how many distinct tuples it shared with the input. However, we had no idea whether or not these tuples were adjacent or spread out. One extreme would be to pull out the sequences that have the most tuples in common. Of course, this favors longer sequences over shorter ones. In fact, a few really long sequences — say on the order of a million characters in length — would always get pulled out, with only four million distinct tuples to go around. The other extreme would be to pull out the sequences that shared the largest percentage of own tuples with the input. Naturally, this favors shorter sequences. We didn't like the spectrum in between either. After experimenting with far-out scoring functions — like the ratio of the intersection of the tuples in the input and in the target with the union of the tuples in the input and the target — we decided that any heuristic would miss some very similar regions. Keeping in mind that the selection process had to be computationally inexpensive, we shied away from the approach in question.

Instead, we favored an algorithmic framework with actual guarantees. It seemed a nice idea to pull out all sequences sharing at least k consecutive characters with the input. The user could provide $k \geq 11$. From there, we established the notion of a match.

2.3 Definition of a Match

Consider two sequences s_0 and s_1 of length l_0 and l_1. Let $e(s_j)$ be the encoding of s_j over an alphabet Σ, with $e(s_j)[i]$ encoding the ith character in s_j for $-1 \leq i \leq l_j$; $0 \leq e(s_j)[i] \leq |\Sigma|$. Let $e(s_j)[-1] = e(s_j)[l_j] = 0$, with the zero reserved for the invalid symbol. Also, let the zero/one indicator function $f_=(x, y) = 1$ if and only if $x = y$ and $x, y \in \{1, \ldots, |\Sigma|\}$, since invalid characters don't match each other.

A match of length l exists between s_j and $s_{\bar{j}}$ at positions p_j and $p_{\bar{j}}$ if and only if

$$\neg f_=(e(s_j)[p_j - 1], e(s_{\bar{j}})[p_{\bar{j}} - 1]),$$

$$f_=(e(s_j)[p_j + k], e(s_{\bar{j}})[p_{\bar{j}} + k]), 0 \leq k < l,$$

$$\neg f_=(e(s_j)[p_j + l], e(s_{\bar{j}})[p_{\bar{j}} + l]),$$

with $p_j \in \{0, \ldots, l_j - 1\}$ and $p_{\bar{j}} \in \{0, \ldots, l_{\bar{j}} - 1\}$. In other words, we concern ourselves only with the longest matches.

Consider identical strings s'_0 and s'_1 of length $l' > 0$. Clearly, all substrings $s'_0[i \ldots j]$ and $s'_1[i \ldots j]$, $0 \leq i \leq j < l'$, are also identical. We avoid reporting a quadratic explosion of

$$\sum_{i=0}^{l'-1} \sum_{j=i}^{l'-1} = \frac{l'(l'+1)}{2}$$

trivial matches for every longest match of length l'. Existence of a match in the dictionary's sense implies the presence of all shorter sub-matches contained within.

2.4 Match Filter

We now describe the procedure used to extract the sequences containing at least one valid match. To do so, simply slide along the input sequence, considering the 11-tuple at each position. For every valid tuple, pull out the sequences from the database that also have it and add each sequence to a pool of "live" ones; keep track of the position when a sequence came to life. Remove from the pool any sequence that did not contain the present tuple. When removing the sequence, pay attention to the position when it came to live. If it has been alive for at least k iterations, mark it as interesting. In this way, we pruned out a lot of sequences that shared many tuples with the input, when it was clear that they did not have many matching consecutive characters. To improve locality, we actually sorted the tuples in the input and pre-fetched the sets of sequences containing those tuples. Having found the interesting sequences, we compared each against the input and reported all of the matches no shorter than k. Of course, even those sequences marked as interesting were not guaranteed to actually contain matches.

2.5 Inefficiencies

We have discovered an algorithmic approach for extracting interesting sequences. The notion of a match introduced guarantees: if there is a match of length k or longer between the input and some sequence in the database, it will be reported; nothing short of this requirement will clutter the output.

Still, several issues remained. First of all, we built the dictionary over a fixed alphabet. Because there are only four characters in the case of DNA, and because four is a power of two, we could get away with a very efficient implementation. However, when we decided to build a similar dictionary out of a set of proteins, we had to redo a lot of the work, but now with an alphabet of twenty amino acids. To battle this defect and to allow the dictionary to seamlessly integrate sequences defined over arbitrary alphabets, we made the alphabet an input to the program that builds a dictionary, see section 5. This was the easiest problem to solve.

Second, the fact that tuples were indexed by sequences in which they appeared create quite a bit of inefficiency. If there were a match of length k in a sequence much longer than k characters, the entire sequence would still have to be pulled out and searched for a match. The existence of tuples in a sequence did not tell us where to look for them. It would be nice to concentrate only on those areas of a sequence where the matches were likely. In the worst case, the database could consist of a single extremely long sequence. Then all of the effort going into marking that sequence as interesting would be pure waste.

Third, the fixed tuple size seemed arbitrary. Why eleven? It is clear that the tuple should be longer than one or two or three; it seems obvious that tuples should be shorter than one hundred. But why not ten, or twenty, or thirty? The trouble with a fixed tuple size is two-fold. First of all, a dictionary built to operate over t-tuples simply could not detect all matches at a resolution of lower than t. This may or may not be a serious problem. Still, a dictionary based on t-tuples ignored the huge hint provided by the user: k. If k were rather large — say one hundred — and t were eleven, as before, the dictionary would still pull out all of the places where 11-tuples matched and try to piece together longer hits. What we really wanted was a variable tuple-length dictionary — one that could be built once and for all to be able to operate at any resolution and to dynamically adjust its searching method to run as quickly as if it had been built with a fixed k.

2.6 Modern Dictionary

We have managed to gather enough algorithmic insights to build a dictionary paradigm addressing all of the problems mentioned above. The new streamlined dictionary builds `dbEST` in close to hour; smaller dictionaries like the protein `OWL` database are assembled in a matter of minutes.

The basic idea is simple. For all intents and purposes, the entire database of sequences could be treated as a single sequence formed by concatenating individual sequences, with invalid characters interspersed between them. Let position p index a subsequence that begins at the offset p in this global sequence and runs until the end. Sort these positions to produce a list of positions ordered lexicographically over some alphabet Σ by the subsequences they index. To use the dictionary with some input sequence, looking for matches no shorter than k, first identify the k-tuples in it. Then, use binary search to find those subsequences in the database that begin with those same k-tuples. Having found them, report the true matches. Care must be taken to sort and search efficiently and to reconstruct the indices of the interesting sequences along with the positions of the matches properly. The following table lists the building times for a few useful dictionaries:

We devote the first part of this thesis to an explanation of how to use the dictionary and a description of the algorithms and the code that actually makes it work. In the second part, we pay attention to applications, citing interesting ways in which we have already put

Dictionary	*Number of Sequences*	*Non-Zero Encodings*	*Space* $\vert_{\mathtt{z=0}}$[a] *(b)*	*Time* $\vert_{\mathtt{z=0}}$ *(s)*
Repeats [24]	2,170	1,747,928	8,850,139	6
`DNA`	816.022	1,770,768	7,447,143	31
USC Genes [9]	490	3,054,237	15,303,963	7
`DNA`	6,237.576	3,056,412	13,076,941	58
Translated Repeats[b]	2,170	6,851,493	34,605,319	22
`AA`	3,268.088	7,091,752	30,969,677	28
OWL[c] [19]	312,942	100,727,386	515,682,508	463
`AA`	322.105	100,800,059	451,429,220	593
Masked OWL[d]	312,942	100,177,377	513,480,320	463
`AA`	322.105	100,800,059	449,153,263	598
`dbEST`[e] [18]	3,934,985	1,528,178,687	7,799,707,036	6,956
`DNA`	390.199	1,535,427,276	6,562,129,892	31,478
Alphabet	*Average Length*	*Combined Length*	*Space* $\vert_{\mathtt{z=9}}$ *(b)*	*Time* $\vert_{\mathtt{z=9}}$ *(s)*

Table 2.1: Building Statistics

[a]The `Z` parameter corresponds to the gzip compression level of sequences. For these runs, `A` = 32 and `L` = 50000000, see 6-1 for a detailed explanation of these parameters.

[b]These are concatenated translations of each sequence from Repeats, in each of the possible three frames and four orientations, separated by an invalid character.

[c]OWL release of 05-23-1999.

[d]These are the proteins from OWL, with all matches of length eight or more to Translated Repeats replaced with invalid characters; it took roughly fifteen minutes to mask OWL by separately querying each sequence against the Translated Repeats dictionary.

[e]`dbEST` release of 04-22-2000.

the dictionary to work in our lab. Finally, we outline possible future directions for the idea and interesting software developments that could lead into other theses.

Part I

Concepts & Constructs

Chapter 3

Sorting

3.1 Definition of the Problem

Consider an alphabet Σ with $|\Sigma| = \alpha, \alpha > 0$. Let s be a valid sequence of length $l = L(s)$ over Σ if $0 < s_i \leq \alpha$ for $0 \leq i < l$ and $s_l = 0$. Define ${}_k s$ as the substring of s which begins at the kth position s_k and runs to the end of s, $0 \leq k < l$. Let $S(s)$ be the set of all such substrings $\{{}_k s \mid 0 \leq k < L(s)\}$. The task at hand is to sort all of the sequences in this set. More generally, given n valid sequences s^j with $L(s^j) > 0$ for $0 \leq j < n$, the problem is to sort all of the sequences in

$$\bigcup_{j=0}^{n-1} S(s^j) = \bigcup_{j=0}^{n-1} \left\{{}_k s^j \mid 0 \leq k < L(s^j)\right\}$$

lexicographically with respect to Σ. The challenge is to sort these sequences efficiently in terms of both time and space, paying attention not only to the theoretical aspects of the asymptotics but also to such practical systems issues as locality, page-swapping, etc.

An interesting and useful variation of this problem involves a parameter m or minimum length. In it, all sequences shorter than m are discarded from the set; remaining sequences are sorted only by the first m characters. That is, sequences whose first m characters match up are considered lexicographically equal. Our solution to the original problem lends itself to a simple and elegant modification to solve this variation of the problem.

3.2 Input and Output

We assume, for practical purposes, that $\alpha < 256$, and represent each character as a single-byte **unsigned char** in the C++ programming language.

```
static void Dictionary::SubStringSort ( unsigned char const alphasize,
                                        unsigned long int const minlen,
                                        unsigned char *const contig,
                                        unsigned long int const contiglen,
                                        unsigned long int *const disk,
                                        unsigned long int const disklen
                                      ) throw ();
```

The input consists of six parameters. The first is the size of the alphabet `alphasize`, which is equal to $\alpha = |\Sigma|$. The second is `minlen`, representing the m from the variation of the problem; if each string is to be sorted completely, then m could be set to `contiglen`.

The third is `contig`, a pointer to a memory location containing the n sequences preceded by a zero: it is a concatenation

$$0 \cdot s^0 \cdot s^1 \cdot \ldots \cdot s^{n-1}$$

of length

$$\lambda = 1 + \sum_{j=0}^{n-1} L(s^j).$$

The fourth parameter `contiglen` represents λ and indicates that `contig` points to an array of λ bytes.

The fifth is `disk`, a pointer to a memory location — which may be mapped to disk, as the name suggests — where the solution is to be written. The sixth and final parameter `disklen` represents δ, the number of non-zero characters in `contig`, and indicates that `disk` points to an array of 4δ bytes, assuming 32-bit **unsigned long int**s.

Since each string to be sorted can be represented by a single integer, its offset in `contig`, the output is simply a listing of all δ valid offsets, properly ordered. The result is written in the form of an array of `disklen` **unsigned long int**s to `disk`.

3.3 Brute Force Attempts

It seems that the most straight-forward approach to solving this problem is to write out all of the valid offsets onto `disk` and sort them using the *QuickSort* algorithm; its comparison function would have to be overridden to compare the strings at the offsets as opposed to the offsets themselves. On average, the running time of this approach would be $O(x\delta \lg \delta)$, with worst-case performance of $O(x\delta^2)$, where x is the time it takes to execute a comparison. Since comparisons in this case involve strings, x is $O(\delta)$. On average, assuming that each character is an independent outcome from a uniform distribution,

$$x < \sum_{i=1}^{\infty} i \frac{\alpha - 1}{\alpha^i} = \frac{\alpha}{\alpha - 1} = \Theta(1),$$

for $\alpha \geq 2$. However, a worst-case running time of $O(\delta^3)$ is not impossible.

The most basic example of an input exhibiting such behavior would be a single sequence of the form $\underbrace{ccc\cdots c}_{\delta}0$. Any comparison between two of its substrings requires pairwise comparisons of as many characters as there are in the shorter substring. The combined algorithm performs rather poorly when *QuickSort* partitions around an extreme pivot. If the pivot always corresponds to the shortest (and therefore smallest) string, the running time is

$$T(\delta) = \sum_{i=1}^{\delta} i^2 = \frac{\delta(\delta+1)(2\delta+1)}{6} = \Theta(\delta^3)$$

and if the pivot always corresponds to the longest (and therefore largest) string, the running time is

$$T(\delta) = \sum_{i=1}^{\delta}\sum_{j=1}^{i} j = \sum_{i=1}^{\delta} \frac{i(i+1)}{2} = \frac{1}{2}\sum_{i=1}^{\delta}\left(i^2+i\right) = \frac{\delta(\delta+1)(\delta+2)}{6} = \Theta(\delta^3).$$

Performance improves if each partitioning of the *QuickSort* algorithm is "lucky" enough to select the median pivot and the running time ends up being

$$T(\delta) = \sum_{i=1}^{\lfloor\frac{\delta}{2}\rfloor} i + \sum_{i=\lceil\frac{\delta}{2}\rceil}^{\delta} \frac{\delta}{2} + T(\frac{\delta}{2}) + \left(T(\frac{\delta}{2}) + \frac{\delta}{2}\Theta(\frac{\delta}{2}\lg\frac{\delta}{2})\right) = \Theta(\delta^2 \lg \delta)$$

Surely, this is a far-fetched scenario. Besides, one could check to make sure that the input is not composed of the same character... Still, the algorithm has a rather poor worst-case running time.

Here is a less degenerate case. Suppose the input consists of $k > 1$ identical sequences of length $\frac{\delta}{k}$ each. At some point, *QuickSort* would zoom in on the sets of identical substrings, taking a quadratic number of comparisons to sort each subgroup. The running time then comes out to

$$T(\delta) = \sum_{i=1}^{\frac{\delta}{k}} ik^2 = k^2\frac{\frac{\delta}{k}(\frac{\delta}{k}+1)}{2} = \Theta(\delta^2).$$

This is much better than before, but still isn't great. However, *QuickSort* could be modified to check whether or not the input consists of identical strings before partitioning to remove the worst-case quadratic behavior. The check would be quite simple and efficient: run through the input once, looking only at the first character of each string to be sorted. If they are distinct, proceed with partitioning, etc. If not, increment the pointer to each string and repeat, effectively chopping off the first characters. One would need to be a little careful regarding string termination though.

An even better solution would be to use a *BucketSort* / *CountingSort* combination. One could consider the first k characters of each string in the *CountingSort* pass, resulting in $k^{\alpha+1}$ buckets for *BucketSort*; the reason for $\alpha + 1$ is that the null terminator must be included as a possible character. Afterwards, each bucket could be sorted using the modified *QuickSort* from above, ignoring the first k characters of each string in the bucket.

All of these approaches remain brute-force, however, because they fail to address the inherent nature of the problem. The strings to be sorted are not unrelated. They are substrings of each other. It seems that one could latch onto this insight, reuse a lot of computations, and produce a very fast and efficient algorithm.

3.4 Saving a Factor of Two

Our first insight reduces the problem by at least a factor of two. Sorting fewer than half of the strings, we show how to reconstruct the other half in linear time using very little extra space. Specifically, assume we could obtain the ordered list of all strings s such that $L(s) \geq 2$ and $s_0 < s_1$ in addition to the list of all strings of length one. Consider a character y. Let X be the set of all characters smaller than y: $X = \{c \mid 1 \leq c < y\}$; let Z be the set of all characters greater than y: $Z = \{c \mid y < c \leq \alpha\}$. Finally, let A be the set of all characters: $A = \{c \mid 1 \leq c \leq \alpha\}$. In what follows, let $x \in X$, $z \in Z$, and $a \in A$. In order to produce the sorted list of all strings beginning with y we partition these strings into five groups:

1	$y0$
2	yxa^*0
3	$yy^+((xa^*)\|\epsilon)0$
4	yy^+za^*0
5	yza^*0

Table 3.1: Sorting Partition

It remains to sort each of these. The first group is trivial: any listing of strings beginning with y of length one is already sorted. The second group is easy if we have already sorted the strings beginning with the characters x: for each x from 1 to $y - 1$ read the sorted listing of offsets; for each offset o such that `contig`$[o-1]$==y, add $o - 1$ to this group. The third group can be generated in the same spirit as the second by reading sorted offsets from groups one, two, and what is available of three. The fourth group is slightly trickier: it can be generated in reverse by reading the offsets from group five and what is available of four in reverse. Group five is sorted by assumption. Of course, one need not be quite as explicit about the boundaries between the sorting of strings that begin with different characters. The following algorithm is slightly more efficient in accomplishing the same task.

Assume without loss of generality existence of arrays **unsigned long int const** $*$**const** `aleph`$[1 \ldots \alpha]$ and **unsigned long int const** $**$**const** `doublet`$[0 \ldots \alpha][0 \ldots \alpha]$. These may be left over from the sorting job, with aleph$[i]$ indicating the number of occurrences of the character i in contig and doublet$[i][j]$ indicating the number of occurrences of i followed immediately by j. In addition, let

$$\pi = \sum_{i=1}^{\alpha-1} \sum_{j=i+1}^{\alpha} \texttt{doublet}[i][j]$$

be the number of pre-sorted strings, whose ordered offsets are stored in the first π locations in disk. Also, let **unsigned long int** *src = disk + π. To make illustrations simpler, consider an alphabet of size three where characters one through three stand for symbols A, B, and C; let the dash (-) represent the gap.

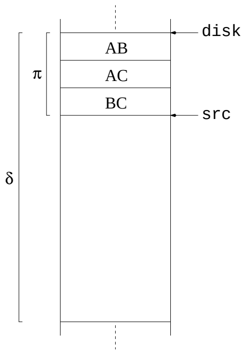


Figure 3-1: Initial Memory Configuration

Define ** **const** top$[1 \ldots \alpha]$, ** **const** bot$[1 \ldots \alpha]$. First, set up top$[i]$ and bot$[i]$ to point into those locations in disk where the offsets for strings beginning with the character i start and end:

```
{
    unsigned long int *const diskm1 = disk - 1;
    unsigned long int total = 0;

    for (unsigned char a = 1; a <= α; a++)
    {
        top[a] = disk + total;
        bot[a] = diskm1 + (total += aleph[a]);
    }
}
```

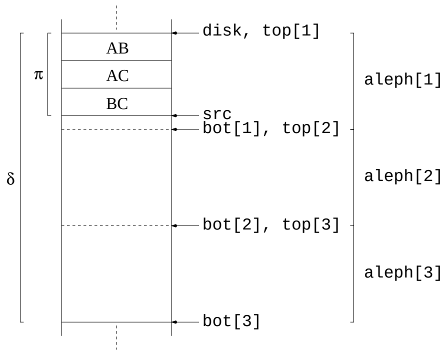


Figure 3-2: Now with `top` and `bot`

Next, move the π sorted offsets into their proper positions. This is accomplished in-place, by reading the offsets backwards, starting from `src`. For each character y, from $\alpha - 1$ to 1, two goals are accomplished: group five is moved into place and group four is generated immediately:

```
for (unsigned char a = α - 1; a; a--)
{
    unsigned long int total = 0;
    for (unsigned char b = a + 1; b <= α; total += doublet[a][b++]);

    unsigned long int *bt = bot[a], *btm = bt + 1;
    unsigned long int const *const sent = bt - total;
    while (bt > sent) *(bt--) = *(--src);

    while (bt < --btm)
    {
        unsigned long int const pos = *btm - 1;
        if (*(contig + pos) == a) *(bt--) = pos;
    }

    bot[a] = bt;
}
```

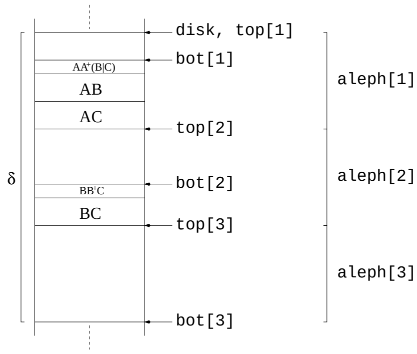


Figure 3-3: Groups Four and Five Constructed

Now, locate all strings of length one and record their offsets into the appropriate positions. This constructs all groups one:

```
{
    unsigned char const *const csent = contig + contiglen;
    for (unsigned char const *start = contig, *next = start + 1; start < csent; start = next++)
    {
        unsigned char const x = *start;
        if (x && !*next)
        {
            *(top[x]++) = start - contig;
            next++;
        }
    }
}
```

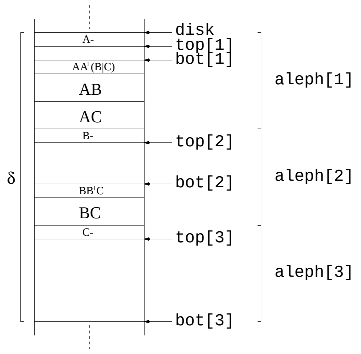


Figure 3-4: Groups One, Four and Five Constructed

All that remains is to fill in the gaps. We do this by gliding down the offsets from the very beginning of `disk`; for each offset o, if y == `contig`$[o]$ and y < `contig`$[o-1]$, $o-1$ is written into the appropriate position (pointed to by `top`[`contig`$[o-1]$], which is incremented). While `top`$[y]$ < `bot`$[y]$, the offset is written also if y == `contig`$[o-1]$.

```
src = disk;
for (unsigned char a = 1; a <= α; a++)
{
    unsigned long int const* const sentb = src + aleph[a], *const senta = bot[a] + 1;

    while (src < senta)
    {
        unsigned long int const pos = *(src++) - 1;
        unsigned char const x = *(contig + pos);
        if (x >= a) *(top[x]++) = pos;
    }

    while (src < sentb)
    {
        unsigned long int const pos = *(src++) - 1;
        unsigned char const x = *(contig + pos);
        if (x > a) *(top[x]++) = pos;
    }
}
```

As a result, all of the strings are now sorted. The time taken is clearly $\Theta(\delta + \alpha^2)$ and the amount of extra space taken is only $\Theta(\alpha^2)$. Of course, $\alpha \ll \delta$; in fact, for all practical purposes α can be treated as a constant which never exceeds 255 and is usually smaller by an order of magnitude. The only question that remains is how big is π, the number of strings that actually had to be sorted. The number of distinct doublets xy, $x < y$ is

$$\sum_{i=1}^{\alpha-1} \sum_{j=i+1}^{\alpha} 1 = \sum_{i=1}^{\alpha-1} (\alpha - (i+1) + 1) = \sum_{i=1}^{\alpha-1} (\alpha - i) = \alpha(\alpha-1) - \frac{\alpha(\alpha-1)}{2} = \frac{\alpha(\alpha-1)}{2},$$

while the total number of distinct doublets is α^2. If doublets occur with equal frequencies, then the fraction of strings that needs to be sorted is

$$\frac{\alpha(\alpha-1)}{2\alpha^2} = \frac{1}{2} - \frac{1}{2\alpha}.$$

The smallest interesting alphabet — the binary system — has $\alpha = 2$, leading us to expect to have to sort only $\frac{1}{4}$ of the number of strings. As α increases though, the expected number of strings to be sorted approaches $\frac{1}{2}$.

Expectations are nice, but we set out to guarantee a reduction by a factor of two. Here is the trick. For all inputs, the number of doublets xy plus the number of doublets yx is smaller than the total number of strings δ (there must be at least one string of length one). By symmetry then, either the number of doublets xy or the number of doublets yx must be smaller than $\frac{\delta}{2}$ If the number of xys is the smaller, we are done. If not, consider the transformation which reverses the alphabet. That is, replace every character c in `contig` with $\alpha - c$. Now the number of xys will be the smaller. Proceed, noting that α is now the terminator. The offsets in `disk` will simply be listed backwards. Reversing elements of an array is a simple exercise, easily done in-place in linear time.

3.5 Modified *CountingSort*

We rely heavily on ideas from *CountingSort* in our code. However, because we do not value stability highly in our application, we modify the traditional algorithm and sacrifice stability in favor of a significant reduction in the memory requirement. We describe the modification here because the standard algorithms reference [7] does not cover it[1], leading us to believe that the reader may not be aware of this result.

Briefly, here is how the standard *CountingSort* algorithm works. In comes an array of non-negative integers a of length n, $0 \le a_i \le m$, for $0 \le i < n$. The algorithm initializes a counter $c_j = 0$ for each $0 \le j \le m$ and slides through a, incrementing c_{a_i} for each i. Now the value c_j contains the number of js in a. Next, c_j is updated to produce for each j the offset c'_j in the sorted array where j would be written if it existed, with $c'_0 \equiv 0$ and $c'_{j+1} = c'_j + c_j$, using a procedure which transforms c into c' in-place. The standard

[1]However, the text does mention a related in-place algorithm, in which the sorted output is to be written to a data stream, such as a graphics display, in exercise 9.2-4 on page 177.

algorithm then produces a stably-sorted version of a in a new array b. Sliding through a, for each a_i, the algorithm sets $b_{c_{a_i}} = a_i$ and increments c_{a_i}, which guarantees not only that b contains the elements of a in sorted order, but also that ties are resolved by the order of the elements' appearance in a. When m is bounded by a constant, the algorithm performs rather well, with the exception of the extra $\Theta(n)$ space required for writing b (or for backing up a).

We modify *CountingSort* to assuage the memory hit. Although our algorithm still requires $\Theta(n)$ extra space, these asymptotics are misleading. In practice, the algorithm rarely requires any. We intervene in the normal proceedings of *CountingSort* just before it writes b. In our case, b is an array of bits of length n with all elements set to zero. The truth value in b_i indicates whether or not a_i is already in its proper sorted position. In the beginning, we cannot make this claim for any i. Once the algorithm completes, b should be full of ones.

The modified algorithm slides through a, looking for the a_i whose $b_i = 0$. The counter c_{a_i} tells exactly where in a, $o = c_{a_i}$, a_i really belongs. However, unless $o = i$, a_o contains an element of a that would be irrecoverably lost if we wrote over it. We tackle the problem as follows: let a temporary variable $t = a_o$, write over a_o with a_i, and set b_o to one. Now, attempt to place t the same way we placed a_i. As we work our way through this chain of elements of a, the cycle is bound to complete: at some point $o = i$. By then, a_i will be in the proper position, together with all the elements that got bumped along the way. The algorithm resumes sliding through a searching for more such chains by looking at the bits in b. The procedure takes a little toying with to command trust.

Regarding the memory requirement: in theory, n bits take up $\Theta(n)$ extra memory. In practice, the case rarely arises that $\exists i$ such that $a_i > \frac{\texttt{ULONG\_MAX}-1}{2}$, which means that the extra "done" bit could be tagged onto the elements of the array. Specifically, just before diverging from normal *CountingSort*, we replace a_i with $2a_i$. When considering elements of a_i we then always strip off the last bit via a_i `>>` 1. To set b_o to one, simply increment a_o by one; to test b_i's truth value, use a_i`&`1. Finally, just before the algorithm terminates, we must be careful to remember to strip off the extra bits (all of which are now one).

Although the author came up with the algorithm from scratch, we assume that the trick is standard practice. Therefore, we conclude that a formal proof would be inappropriate for this thesis. However, we have presented an overview for completeness because the algorithm did not make it into [7].

3.6 Sorting the Other Half

The problem has been reduced. We are still faced with a `contig` of δ characters, but we have to sort fewer than $\frac{\delta}{2}$ of them. The task is now to write out only those offsets o for which `contig`$[o]$ `!=` 0 and `contig`$[o]$ `<` `contig`$[o+1]$ onto `disk` in sorted order. A nice benefit is that more than $\frac{\delta}{2}$ of the locations in `disk` are free to be used as working memory.

We could resort to one of the brute-force algorithms at this point, but there are still rewards to be drawn because the strings are still not independent. Every string s to be sorted has the form $s_0 s_1 \ldots s_k \ldots$, where $s_0 < s_1$, $s_i \geq s_{i+1}$ for $1 \leq i < k$, and either $s_k = 0$ or $s_k > s_{k-1}$ for some minimum $k \geq 2$. Consider two sequences s^a and s^b which share the first $k+1$ characters, with $s_k \neq 0$. Had the sequences ${}_{k-1}s^a$ and ${}_{k-1}s^b$ already been sorted, we would know the positions of their offsets in `disk`. To compare s^a and s^b we would need only to compare these values instead of comparing the remaining characters. This insight allows us to reuse previous computations and lays the foundation for the following algorithm. First, define several static variables (to be used by the recursive helper-function):

```
unsigned char flipflop = 0;
unsigned char const *ptr;
unsigned long int depth, minlen,
    amemsize = (α + 2) * sizeof (unsigned long int),
    *const aleph[0 ... α + 1],
    *const asent = aleph + (α + 1),
    **const doublet[0 ... α][0 ... α],
    **const offset[0 ... α][0 ... α];
```

Let `offset`$[i][j]$ represent the location in `disk` where the offsets of strings beginning with ij for $1 \leq i < j \leq \alpha$ belong, now that we are sorting only π sequences. Compute `doublet` and `offset`:

```
{
    unsigned long int total = 0;
    unsigned char const *const csent = contig + len;
    for (unsigned char const *start = contig; start < csent; doublet[*start][*(++start)]++);

    for (unsigned char a = 1; a < α; a++)
        for (unsigned char b = a + 1; b <= α; b++)
        {
            offset[a][b] = total;
            total += doublet[a][b];
        }
}
```

Next, observe that `disk` will contain π offsets, while there is enough memory in `disk` for more than 2π offsets. Let **unsigned long int** *****const* `destination`$[1 \ldots \pi]$ point to that chunk of working memory; also set **unsigned long int** `*indx = destination`. We then place the offsets to be sorted into `disk`; for each offset o where `disk`[i] $= o$, we also set `indx`[i] to be that o's order of appearance in `contig` among strings to be sorted. Since we have already counted all doublets, we pre-sort `disk` and `indx` by the first two characters of each string:

```
{
    unsigned long int tmp = 0;

    for (unsigned char const *start = contig, *next = start + 1; start < csent; start = next++)
        if (*start && *start < *next)
```

```
        {
            unsigned long int const q = offset[*start][*next]++;
            disk[q] = start - contig;
            indx[q] = tmp++;
        }
}
```

The first major subgoal is to sort each of the π strings in `disk` just up to the point where that string merges with another string to be sorted. The result will be a list of indices, each with two extra bits appended, of the form

$$\texttt{disk}[i] == ((j << 2) \mid (b_1 << 1) \mid b_0),$$

meaning that the jth string to be sorted will be in ith position if $b_0 == 0$; or, if $b_0 == 1$, meaning that the jth string is still tied for the ith position with the kth string if

$$\texttt{disk}[i \pm 1] == ((k << 2) \mid (b_1 << 1) \mid b_0).$$

That is, b_0 indicates whether or not the string is still tied with some other string; if $b_0 == 1$, then b_1 indicates which of strings adjacent to it in `disk` are tied together. We accomplish this subgoal via a helper function `PosToOrder`; let **unsigned long int** `*doff = disk`.

```
ptr = contig + (depth = 1);
for (unsigned char a = 1; a < α; a++)
    for (unsigned char b = a + 1; b <= α; b++)
    {
        unsigned long int const len = doublet[a][b];

        if (len > 1)
        {
            PosToOrder (doff, indx, len);
            doff += len, indx += len;
        }
        else if (len)
            *(doff++) = *(indx++) << 2;
    }
```

```
void PosToOrder ( unsigned long int *dump,
                  unsigned long int *indx,
                  unsigned long int const len
                ) throw ();
```

This function takes the block of positions and the block of indices of strings that are tied in the first k characters, and the total number of strings to be sorted. The output, written out to `dump`, is the list of indices with the extra two bits, as described above. First, to each index in `indx`, two zeros are appended:

```
unsigned long int sub = 0;
{
    unsigned long int const *const isent = indx + len;
    for (unsigned long int *i = indx; i < isent; *(i++) <<= 2);
}
```

Then, the upcoming character of each string is considered and the total number of each character represented is tallied:

```
again:
depth++;
ptr++;
sub++;

if (depth < minlen)
{
    memset (aleph, 0, amemsize);
    unsigned long int const *const dsent = dump + len;
    for (unsigned long int *start = dump; start < dsent; aleph[*(ptr + *(start++))]++);
```

These counts are then used to compute the offset for each character, unless only a single character is represented and that character is non-zero:

```
    for (unsigned long int *aptr = aleph; aptr < asent; aptr++)
    {
        unsigned long int const q = *aptr;
        if (q == len && aptr > aleph) goto again;
        *aptr = *asent;
        *asent += q;
    }
```

Also, since **aleph** is static, remember the boundary between those strings which will need to be further sorted (because their upcoming character is not greater than the previous one) and those that will not.

```
    unsigned long int const *isent = dump + aleph[1];
    unsigned long int const *const rsent = dump + aleph[*(ptr + *dump - 1) + 1];
```

Finally, we get to do something a little more interesting. The following code implements a reordering of `dump` and `indx` by the up-coming character of each string; the sort is in-place (but is not stable[2]). After the sort is complete the values in `indx` end in 01 as opposed to 00 though. The process is quite straight forward: the last bit of every value in `indx` starts out as zero; `indx`[i] & 1 == 1 signifies that `dump`[i] and `indx`[i] already contain their final sorted values. The algorithm loops through all i looking for one not yet sorted. Having found such $i_0 = i$, its final position p is determined by a counter associated with the appropriate upcoming character; the counter is subsequently incremented. The values in `dump`[p] and `indx`[p] are overwritten with `dump`[i] and `indx`[i] & 1, but not before they are saved in temporary storage. The process repeats, with p taking the place of i, until finally a p is encountered which equals i_0, completing the cycle. The algorithm carries on looking for more unsorted chains:

```
    for (unsigned long int i = 0; i < len; i++)
        if (!(indx[i] & 1))
```

[2]Stability could be restored if we made the effort to sort those strings with the upcoming character of zero by their index value. Essentially, we are invoking a modified version of *CountingSort* — one which sacrifices stability to avoid the memory hit associated with having to back up the input; see section 3.5.

```
{
    unsigned long int pos = dump[i], indx = indx[i] | 1, loc;

    while ((loc = aleph[*(ptr + pos)]++) != i)
    {
        unsigned long int const u = dump[loc], v = indx[loc] | 1;
        dump[loc] = pos, indx[loc] = idx;
        pos= u, idx = v;
    }

    dump[i] = pos, indx[i] = idx;
}
```

Those strings with the upcoming character equal to zero are already sorted. It remains to move their `indx` block onto `dump`, replacing the trailing 01 with 00. Strings with the upcoming character no greater than the last one need to be resorted: their `dump` is rid of the trailing 01s and the strings are resorted.

```
while (dump < isent) *(dump++) = *(indx++) - 1;
isent = indx + (rsent - isent);
for (unsigned long int *iptr = indx; iptr < isent; *(iptr++) >>= 2);

for (unsigned long int const *start = dump, *stop; start < rsent; start = stop)
{
    unsigned char const x = *(ptr + *start);
    for (stop = start + 1; *(ptr + *stop) == x && stop < rsent; stop++);
    unsigned long int const chunk = stop - start;

    if (chunk > 1)
        PosToOrder (dump, indx, chunk);
    else if (chunk)
        *dump = *indx << 2;

    dump += chunk;
    indx += chunk;
}
```

What's left is only to set up b_1 appropriately for the strings whose upcoming character is greater than the previous one. Afterwards, `depth` and `ptr` are returned to their original values and `PosToOrder` returns:

```
for (unsigned long int const *start = rsent, *stop; start < dsent; start = stop)
{
    unsigned char const x = *(ptr + *start);
    for (stop = start + 1; *(ptr + *stop) == x && stop < dsent; stop++);
    unsigned long int const chunk = stop - start;

    if (chunk > 1)
    {
        memcpy (dump, indx, chunk * sizeof (unsigned long int));

        if (flipflop)
        {
            unsigned long int const *const sent = dump + chunk;
            for (unsigned long int *ptr = dump; ptr < sent; *(ptr++) |= 2);
```

```
                flipflop = 0;
            }
            else
                flipflop = 1;
        }
        else
            *dump = *indx - 1;

        dump += chunk;
        indx += chunk;
    }
}
else
    memcpy (dump, indx, len * sizeof (unsigned long int));

depth -= sub;
ptr -= sub;
```

We now embark on the second major subgoal: listing in `destination`[i] the final position, in the ordered set of strings to be sorted, of the ith string. To do this, we first set `destination`[i] to be the value of the position for which the ith string is tied; we also overwrite `disk` with the indices of the sequences which are still tied:

```
unsigned long int *dump = disk, *dwrite = disk;
unsigned long int const *dsent = disk + total;

for (unsigned long int const *dread = dump, *dstop; dread < dsent; )
{
    unsigned char const lastbits = *dread & 3;
    for (dstop = dread + 1; dstop < dsent && (*dstop & 3) == lastbits; dstop++);

    if (lastbits)
    {
        unsigned long int const dst = dread - dump;
        while (dread < dstop) destination[*(dwrite++) = (*(dread++) >> 2)] = dst;
    }
    else
        for (unsigned long int dst = dread - dump; dread < dstop;
                destination[*(dread++) >> 2] = dst++);
}
dsent = dwrite;
```

At this point, exactly the indices that are tied are listed in `disk`. Also note that each group of tied indices has more than one index in it. We proceed as follows: read the entries in disk, identifying each group. Consecutive indices i and j fall in the same group if `destination`[i] == `destination`[j]. For each group found, when comparing i and j, use the values representing the positions of the sequences they blend into, `destination`[$i+1$] and `destination`[$j+1$]. We then sort the group using `HelperSort`, which returns zero if all of the elements are tied, in which case no sort was performed; otherwise, it returns one. If the group is small *QuickSort* was used; otherwise, *RadixSort* was invoked. In either case, the group is split into subgroups, which reduces the number of ties. Because subgroups of size one are no longer tied, they are eliminated from `disk`. This relaxation method iterates until all groups disappear:

```
unsigned long int const *const destinationp1 = destination + 1;
while (dsent != (dwrite = dump))
{
    for (unsigned long int const *dstop = dump, *dread = dump; dread < dsent; dread = dstop)
    {
        unsigned long int const dst = destination[*dstop];
        for (dstop = dread + 1; dstop < dsent && dst == destination[*dstop]; dstop++);
        unsigned long int const chunk = dstop - dread;

        if (HelperSort ((unsigned long int *const)dread, destinationp1, chunk))
        {
            for (unsigned long int *start = (unsigned long int *)dread, *stop; start < dstop; )
            {
                unsigned long int const v = destinationp1[*start];
                for (stop = start + 1; stop < dstop && destinationp1[*stop] == v; stop++);
                start--;

                if (flipflop)
                {
                    while (++start < stop) *start = (*start << 1) | 1;
                    flipflop = 0;
                }
                else
                {
                    while (++start < stop) *start <<= 1;
                    flipflop = 1;
                }
            }

            for (unsigned long int *start = (unsigned long int *)dread, *stop; start < dstop; )
            {
                unsigned char const lastbit = *start & 1;
                for (stop = start + 1; stop < dstop && (*stop & 1) == lastbit; stop++);
                unsigned long int const newdst = dst + start - dread, subgroup = stop - start;

                while (start < stop)
                {
                    *start >>= 1;
                    destination[*(start++)] = newdst;
                }

                if (subgroup > 1)
                {
                    memcpy (dwrite, start - subgroup, subgroup * sizeof (unsigned long int));
                    dwrite += subgroup;
                }
            }
        }
        else
        {
            memcpy (dwrite, dread, chunk * sizeof (unsigned long int));
            dwrite += chunk;
        }
    }
```

```
    dsent = dwrite;
}
```

Having computed `destination`, filling up `disk`$[0 \ldots \pi - 1]$ with sorted positions of strings beginning with $xy, x < y$ is straight-forward:

```
dwrite = destination;
for (unsigned char const *start = contig, *next = start + 1; start < csent; start = next++)
    if (*start && *start < *next) dump[*(dwrite++)] = start - contig;
```

How expensive was all this? Accomplishing the first subgoal is clearly linear in time; since the main sorting procedure is in-place, the amount of extra space required is proportional to the size of the call stack of the recursive helper function. This is equal to the length of the longest non-unique substring starting with xy, $x < y$, to be sorted. On average, we expect roughly $\frac{\lg \pi}{\lg \alpha}$ characters to be enough to uniquely specify the beginning of a string; in practice, this number is negligible. In general, the size of the stack is bounded by $\frac{\delta - \pi}{2}$.

The second subgoal requires very little extra memory: just enough for *RadixSort* to back up the group being sorted. Since there are $\frac{\alpha(\alpha-1)}{2}$ groups, we expect each to have the size

$$\frac{\pi}{\frac{\alpha(\alpha-1)}{2}} = \frac{2\pi}{\alpha(\alpha-1)} < \frac{\delta}{\alpha(\alpha-1)} \approx \frac{\delta}{\alpha^2}.$$

Of course, in the worst case, the amount of memory needed may be as large as π. On average, we expect a constant number of iterations through `disk` to be enough to eliminate all groups (a comparison of two sequences is expected to require looking at a constant number of characters), with the entire process taking $O(\pi)$ time. Each iteration is guaranteed to partition at least one group (`contig` is not a circular buffer). Therefore, at most $O(\pi^2)$ time is needed. The most basic example of an input exhibiting such behavior would be a single sequence of the form $\underbrace{acacac \cdots ac}_{\delta}0$. Thus, the worst-case bound is tight: $\Theta(\pi^2)$.

3.7 Complexity Bounds

We have presented an efficient combined algorithm for the problem:

	Time	Extra Space
Our Algorithm		
Expected Behavior	$O(\pi + \delta + \alpha^2) = O(\delta + \alpha^2)$	$O(\max[\frac{\lg \pi}{\lg \alpha}, \frac{\delta}{\alpha^2}] + \alpha^2) = O(\frac{\delta}{\alpha^2} + \alpha^2)$
Worst-Case Behavior	$\Theta(\pi^2 + \delta + \alpha^2) = \Theta(\delta^2 + \alpha^2)$	$\Theta(\max[\frac{\delta-\pi}{2}, \pi] + \alpha^2) = \Theta(\delta + \alpha^2)$
Brute-Force Algorithm		
Expected Behavior	$O(\delta \lg \delta)$	$O(\lg \delta)$
Worst-Case Behavior	$\Theta(\delta^3)$	$\Theta(\delta)$

Table 3.2: Complexity Bounds.

Here α is the size of the alphabet, δ is the total number of strings to be sorted, and $\pi < \frac{\delta}{2}$; $\alpha \ll \delta$. In practice, $2 \leq \alpha \ll 256$. These asymptotics represent a factor of δ improvement over the brute-force attempts in terms of time at the expense of only a modest amount of space. We must admit that the brute force algorithm is much simpler to understand, to code, and to test — it can be implemented in just a few lines of C. Still, because we expect to be dealing with input strings millions of characters in length, we conclude that the savings are worth the effort.

Chapter 4

Searching

4.1 Definition of the Problem

We are now ready to tackle the problem of finding matches, no shorter than a threshold integral value k in length, between a set of input sequences and set of known sequences in a database. We sweep several issues under the rug: without loss of generality, assume that the database consists of a single sequence B (for big) and that the input consists of a single sequence S (for small). To handle multiple sequences, we could simply concatenate them into a longer sequence. We would need to be careful to insert a zero between the sequences to prevent spurious matches that span several sequences. In addition, we would need to post-process the matches found to compute the true indices of the sequences and the proper positions. We abstract away from these details, saving a thorough treatment of the matter until 7.3.

Let $L(B) = b$ and $L(S) = s$, with both sequences defined over the same alphabet Σ; as before, $|\Sigma| = \alpha$. Then $B_i \in \{0, \dots, \alpha\}$ for $0 \leq i < b$ and $S_i \in \{0, \dots, \alpha\}$ for $0 \leq i < s$; $B_b = S_s = 0$. We impose an additional sentinel value $B_{-1} = S_{-1} = 0$ and seek the matches, as defined in 2.3, between B and S. Recall the zero/one indicator function:

$$f_{=}(x, y) = \begin{cases} 1 & \text{if } x = y \text{ and } x, y \in \{1, \dots, \alpha\} \\ 0 & \text{otherwise} \end{cases}$$

The set of matches is

$$\left\{(p_b, p_s) \mid \forall i, 0 \leq i < k, f_{=}(B_{p_b+i}, S_{p_s+i}) \text{ and } \overline{f_{=}(B_{p_b-1}, S_{p_s-1})}\right\}.$$

4.2 Brute Force Attempts

It seems that the most straight-forward approach to solving this problem is to consider all pairs (p_b, p_s). There are no more than bs such pairs; one need not test more than $k + 1$

characters for each, for an upper bound of $O(kbs)$. While the method is not particularly clever, it works well when the strings are short. This approach could also be parallelized. For instance, bs processors could solve the problem in $O(k)$ time. Although computers seem to get faster and cheaper by the minute, we prefer to optimize a sequential algorithm.

4.3 Binary Search

In solving the problem, we assume that a dictionary has been built out of B. By that we mean simply that we have access to a sorted list P^B of positions in B, ordered lexicographically by ${}_{P_i^B}B$, according to the alphabet Σ. We further assume that $s \ll b$ and construct P^S. However, when constructing P^S, we use the variant of the sort to order the subsequences based on the first k characters only. We throw away positions that index sequences shorter than k characters in length, a task easily accomplished in-place in linear time. Because S is short, the time to compute P^S is negligible. Even if S were long, since only the first k characters of each subsequence must be considered, complexity analysis from 3.2 is no longer tight. At worst, the sort consumes $\Theta(ks + \alpha^2)$ time, even though the amount of extra space required may still approach $\Theta(s + \alpha^2)$.

The high-level idea is quite simple: split P^B down the middle $m = \lfloor \frac{b}{2} \rfloor$ and consider the position $x = P^B_m$ (this takes a constant amount of time). Then search for the smallest y, $0 \leq y < s$, such that the first k-tuple in ${}_xB$ is, lexicographically with respect to Σ, no larger than the first k-tuple in ${}_yS$. We expect to have to consider no more than a constant number of characters when comparing the k-tuples, as in 3.3. If the k-tuples at ${}_xB$ and ${}_yS$ match, find the set of positions X in P^B adjacent to x which index sequences that start with the same k-tuples; analogously, find Y for y. From the Cartesian product $X \times Y$, report only those pairs (x, y) for which $\overline{f_=(B_{x-1}, S_{y-1})}$. Recursively search the remaining top portions of P^B and P^S and the remaining bottom portions of P^B and P^S. To retain the bound of an expected constant number of character comparisons between two k-tuples, we decrease k appropriately upon recursion, see 4.5.

4.4 Complexity Bounds

If the sum of the lengths of matches in a query is l, then there are fewer than l pairs (x, y) with equal k-tuples at ${}_xB$ and ${}_yS$. It takes $\Theta(l)$ time and $\Theta(1)$ space to identify and report the matches having found pairs of equal k-tuples. We break up the time it takes to locate the positions of matching t-tuples x and y into two phases.

During phase one, positions of B are split in half in constant time at each iteration, at which point positions in S are searched for the appropriate cleavage point. The expected amount of time needed to single out chunks of equivalent tuples in S is

$$\sum_{i=0}^{\lg s} 2^i \lg \frac{s}{2^i} = \sum_{i=0}^{\lg s} 2^i \lg s - \sum_{i=0}^{\lg s} 2^i \lg(2^i) = \lg s \sum_{i=0}^{\lg s} 2^i - \sum_{i=0}^{\lg s} i2^i.$$

Recall that

$$\sum_{i=0}^{n} x^i = \frac{x^{n+1}-1}{x-1};$$

differentiating both sides with respect to x, fiddling a bit with summation indices, and then muddling through the algebra yields

$$\sum_{i=0}^{n} ix^i = \frac{nx^{n+2}-(n+1)x^{n+1}+x}{(x-1)^2}.$$

Armed with these two identities, we simplify the running time for phase one:

$$\lg s \sum_{i=0}^{\lg s} 2^i - \sum_{i=0}^{\lg s} i2^i = \lg s \left(\frac{2^{1+\lg s}}{2-1}\right) - \frac{\lg s\left(2^{2+\lg s}\right) - (1+\lg s)\,2^{1+\lg s} + 2}{(2-1)^2} =$$

$$\lg s\left(2\cdot 2^{\lg s} - 1\right) - \lg s\left(2^2\cdot 2^{\lg s}\right) + (1+\lg s)\left(2\cdot 2^{\lg s}\right) - 2 =$$

$$\lg s\,(2s-1) - \lg s\,(4s) + (1+\lg s)\,(2s) - 2 =$$

$$2s\lg s - \lg s - 4s\lg s + 2s + 2s\lg s - 2 =$$

$$2s - \lg s - 2 = \Theta(s).$$

During phase two, it is no longer possible to subdivide the $\Theta(s)$ remaining chunks of P^S. We expect the section of P^B corresponding to each chunk of P^S to contain $\Theta(\frac{b}{s})$ positions. Binary search takes $\Theta(\lg \frac{b}{s})$ time to look for a k-tuple in the section of P^B. The expected amount of time needed to complete phase two and come up with the seed (x, y) for $X \times Y$ is

$$\Theta(s) \times \Theta\left(\lg \frac{b}{s}\right) = \Theta\left(s \lg \frac{b}{s}\right).$$

Assuming that $b > s$, $s \lg \frac{b}{s}$ dominates s. The amount of time spent in phase one is negligible; phase two hogs up the bulk of the cycles. Extra memory required grows proportionally with the recursive call stack, yielding $\Theta(\lg s)$ as a bound for extra space. If the sum of the lengths of matches is l and assuming that the positions of B have been sorted in a pre-processing stage, the following bounds characterize expected performance for the algorithm. The search requires

$$\Theta\left(l + \alpha^2 + s\left(\min\left(k, \frac{\lg s}{\lg \alpha}\right) + \lg \frac{b}{s}\right)\right)$$

time and

$$\Theta\left(\frac{s}{\alpha^2} + \alpha^2\right)$$

extra space.

4.5 Implementation Details

We have outlined the general idea of how the algorithm works. In this section, we proceed to specify the implementation details that actually make it work fast in practice. Here we take a step down from the pure asymptotics and take into account system-level effects and constant factors.

First of all, observe that b may be extremely large. We expect it be on the order of 2^{28}; longer sequences run into the 2Gb problem discussed in 6.2.5. Assuming 32-bit **unsigned long int**s, as before, the list of ordered positions for B could take up as much as 1Gb. While binary search seems efficient in theory, in practice cache misses and swapping pose a severe problem for performance. Without delving into the details of database implementation[1] and the intricacies of B-trees, we opt for a very simple solution. To improve locality for binary search, we maintain not only the arrays of sorted positions P^B and P^S, but also their sampled counterparts $S(P^B)$ and $S(P^S)$, see 6.3.5. Assuming 4Kb pages, each containing 1,024 words, we let $S(P^i)$ contain the last word of the ith page of P^j. $S(P^j)$ has the nice property that it is still lexicographically ordered, allowing binary search to proceed as normal, at a fraction of the memory requirement: the increase is just 0.1%. Given that the order of magnitude of the length of P^B is capped at 1Gb, the sampled file $S(P^B)$ takes up roughly 1Mb of space. At this time, pretty much every personal computer can accommodate this requirement without swapping. For the pre-built dictionary, we store $S(P^B)$ in a separate file to avoid having to scan all of P^B each time the dictionary runs. Given left and right boundaries l and r, instead of letting the middle element looked up be $m = \lfloor \frac{l+r}{2} \rfloor$, we identify the first index in the sample $\mathtt{l} > l$ and the last index in the sample $\mathtt{r} < r$. Unless $\mathtt{l} > \mathtt{r}$, set m to the index in the sample `m` that's between `l` and `r`. The following code does the trick:

[1]Unfortunately, the author has not found a course on databases at MIT.

```
static inline void bSearchSmp  ( unsigned long int &m,
                                 unsigned long int &v,
                                 unsigned long int const l,
                                 unsigned long int const r,
                                 unsigned long int const * const P,
                                 unsigned long int const * const S
                               ) throw ();
```

Here, l and r form the inclusive boundaries of the region searched in the array P; S is the sampled version of P. References m and v store the output, with $l \leq m < r$ and $v = P(m)$ upon return. Implementation is straight-forward:

```
{
    long int const l = (l >> lg(getpagesize() / sizeof(unsigned long int))), r = (r >> lg(getpagesize() / sizeof(unsigned long int))) - 1;

    if (l <= r)
    {
        long int const m = (l + r) >> 1;
        v = S[m];
        m = ((m + 1) << lg(getpagesize() / sizeof(unsigned long int))) - 1;
    }
    else
        v = P[m = ((l + r) >> 1)];
}
```

We rely on `bSearchSmp` to select a pivot for binary search in an array, abstracting away the sampling details. We also abstract away the actual reporting of matches. Having found corresponding blocks in P^B and P^S, such that the substrings starting at these positions in B and S begin with the same k characters — none of which are zero, by the way, since P^S indexes only the substrings at least k characters long — we use a helper function to extract the matches from this Cartesian product:

```
void Dictionary::ReportMatches  ( unsigned long int const sleft,
                                  unsigned long int const sright,
                                  unsigned long int const bleft,
                                  unsigned long int const bright
                                ) throw ();
```

Parameters `left` and `right` demarcate inclusive regions within P^B and P^S. In addition, we introduce another inline function to compare two tuples a and b, which are pointers into sequences of characters, with B serving as the sentinel and pointing just after tuple b:

```
static inline long int TupCmp  ( unsigned char const * a,
                                 unsigned char const * b,
                                 unsigned char const * const B
                               ) throw ();
```

Essentially, `TupCmp` offers a streamlined version of the familiar `memcmp` routine. Our function is faster for two reasons. First of all, we provide a sentinel, as opposed to a length of memory to be compared, eliminating an addition operation for each call. Second, we compare memory as long integers, as opposed to characters, up to a point; this divides the number of operations by the number of bytes in a long integer, a factor of four (for some reason, `memcmp` does not do this). Again, implementation is straight-forward:

```
{
    while (b < B && *((long int *)a) == *((long int *)b))
    {
        a += sizeof (long int);
        b += sizeof (long int);
    }

    if (b < B && *a == *b)
        {a++; b++; if (b < B && *a == *b)
            {a++; b++; if (b < B && *a == *b)
                {a++; b++; }}}

    return (b < B ? (long int)(*a) - (long int)(*b) : 0);
}
```

We finally embark on describing how the main searching routine works. Our implementation is a little ugly, mostly because we tend to frugal with memory. That is, whenever possible, we attempt to replace a recursive call with a **goto** to the beginning of the function. First, the header:

```
void Dictionary::BinarySearch    (    long int togo,
                                      long int sleft,
                                      long int sright,
                                      unsigned char sa,
                                      unsigned char sz,
                                      long int bleft,
                                      long int bright,
                                      unsigned char ba,
                                      unsigned char bz,
                                 )    throw ();
```

This function has access to private members of the dictionary class, including pointers to

```
unsigned long int const *const spos, *const ssmp, *const bpos, *const bsmp;
```

which represent P^S, $S(P^S)$, P^B, $S(P^B)$, along with

```
unsigned char const *sseq, *bseq;
```

which represent S and B. Input parameters include `togo`, which represents k; `sleft` and `sright` are the inclusive boundaries of the region to be searched in `spos`; at the same time, `sa` is the upper bound on the first character of the first sequence in that region, while `sz` is the lower bound on the first character in the last sequence in the region. Analogous explanations render the meaning of the parameters beginning with `b`. Now for the local variables:

```
{
    unsigned long char const *seq, *tup, *tupsent;
    unsigned long int mid = 0, midpos = 0;
    unsigned long int const *pos, *smp;
    long int e0, e1, c0, c1;
    int cmp, sub = 0;
```

Each pass of `BinarySearch` will pick a tuple `tup`, with `tupsent` = `tup` + `togo` and attempt to find the region in `pos` listing positions of those substrings in `seq` beginning with an equivalent tuple; `smp` is the sampled version of `pos`. It isn't clear from the outset which would be searched, P^B or P^S. In phase one, we search P^S. By the time we reach phase two, the region of P^S handed down to us contains only a single tuple, at which point we search what's left of P^B. We use `mid` and `midpos` to represent m and v, respectively. As the search zooms in, we increment the pointers to `seq`, `sseq`, and `bseq` so as not to compare the first few characters which are by then known to be the same. We store in `sub` a counter for the number of such increments, so as to undo the effect before returning. Finally, `cmp` is sometimes used to store the value of a comparison returned by `TupCmp`.

We first check to make sure that a match is possible at all, given the bounds for the first characters:

```
recurse:

    if (sa > bz || ba > sz)
        goto done;
```

We continue in the same vein, considering the case when the first characters of all substrings in the two regions are the same:

```
    if (sa == sz && ba == bz)
        if (sa == ba)
            if (togo == 1)
            {
                ReportMatches (sleft, sright, bleft, bright);
                goto done;
            }
            else
            {
                sseq++; bseq++; sub++; togo--;
                sa = *(sseq + spos[sleft];
                sz = *(sseq + spos[sright];
                bz = α;
                ba = 1;
            }
        else
            goto done;
```

If the both regions contain substrings that start with the same character and if those characters are different for the two regions, then no match is possible. Otherwise, we virtually strip off that first character, unless it is the last character, in which case we report the hits.

Next, we determine which region will be searched. If the first and the last substrings in the region of P^S already begin with the same `togo` characters, we are in phase two and proceed to search the region of P^B. Otherwise, we are in phase one, in which case we split the region of P^B down the middle and proceed to search the region of P^S. First, the setup:

```
if (sa != sz || TupCmp (sseq + spos[sleft] + 1, sseq + spos[sright] + 1, sseq + spos[sright] + togo))
{
    seq = sseq;
    pos = spos;
    smp = ssmp;
    bSearchSmp (mid, midpos, sleft, sright, bpos, bsmp);
    tupsent = (tup = bseq + midpos) + togo;
    b0 = mid - 1;
    b1 = mid + 1;
    s0 = sleft;
    s1 = sright;
}
else
{
    seq = bseq;
    pos = bpos;
    smp = bsmp;
    tupsent = (tup = sseq + spos[sleft]) + togo;
    b0 = sleft - 1;
    b1 = sright + 1;
    s0 = bleft;
    s1 = bright;
}
```

With the preliminaries out of the way, we are ready to search. The post-condition is that `s0` be the index of a position in `pos` marking the beginning of a string starting with the same `togo` characters as `tup`; if no such string exists, we expect `s0` to be the largest index possible of a position in `pos` marking the beginning of a string whose first `togo` characters lexicographically precede those in `tup`. Similarly, `s1` should be the smallest index possible of a position in `pos` marking the beginning of a string starting whose first `togo` characters come lexicographically after those in `tup`.

```
{
    unsigned long int l = s0, r = s1, sub = 0;

    if (TupCmp (seq + pos[l], tup, tupsent) >= 0)
        ;
    else if (TupCmp (seq + pos[r], tup, tupsent) <= 0)
        l=r;
    else
    {
        unsigned char const A = (pos == spos ? 1 : 0), Z = α;
        unsigned char a = *(seq + pos[l]), z = *(seq + pos[r]);
```

```
        while (l < r)
        {
            if (a == z)
            {
                a = A;
                z = Z;

                sub++;
                seq++;
                if (++tup == tupsent) break;

                continue;
            }

            bSearchSmp (mid, midpos, l, r, pos, smp);
            unsigned char const *const split = seq + midpos;

            if (TupCmp (split, tup, tupsent) < 0)
            {
                l = mid + 1;
                a = *split;
            }
            else
            {
                r = mid;
                z = *split;
            }
        }

        s0 = (s1 = l) - 1;
        if ((cmp = TupCmp (seq + pos[l], tup, tupsent)) < 0)
        {
            s0++;
            s1++;
        }

        tup -= sub;
    }

}
```

If we are in phase two, we must swap (s0,s1) and (b0,b1):

```
if (pos == bpos)
{
    unsigned long int tmp;
    tmp = s0; s0 = b0; b0 = tmp;
    tmp = s1; s1 = b1; b1 = tmp;
}
```

With these values swapped and cmp containing the result of the comparison of between tup and what the search actually produced, we are ready to identify the region of matches:

```
if (!cmp)
{
    while (s0 >= sleft && !TupCmp (sseq + spos[s0], tup, tupsent) s0--;
    while (s1 <= sright && !TupCmp (sseq + spos[s1], tup, tupsent) s1++;
```

```
        while (b0 >= sleft && !TupCmp (bseq + bpos[b0], tup, tupsent) b0--;
        while (b1 <= bright && !TupCmp (bseq + bpos[b1], tup, tupsent) b1++;
        ReportMatches (s0 + 1, s1 - 1, b0 + 1, b1 - 1);
    }
```

With the middle pieces extracted from both regions of P^B and P^S, we recurse:

```
    {
        unsigned char const
            recurse_top = (s0 >= sleft && b0 >= bleft),
            recurse_bot = (s1 <= sright && b1 <= bright);

        if (recurse_top)
            if (recurse_bot)
                BinarySearch (togo, sleft, s0, sa, *(sseq + spos[s0]), bleft, b0, ba, *tup);
            else
            {
                sz = *(sseq + spos[sright = s0]);
                bright = b0;
                bz = *tup;
                goto recurse;
            }

            if (recurse_bot)
            {
                sa = *(sseq + spos[sleft = s1]);
                bleft = b1;
                ba = *tup;
                goto recurse;
            }
    }
```

And terminate:

```
done:

    sseq -= sub;
    bseq -= sub;
}
```

Chapter 5

Alphabet

5.1 Motivation

The concept of a dictionary makes little sense without some notion of an alphabet. An alphabet allows for interpretation of sequences, sets up a lexicographic ordering, and provides a basis for normalizing the sequences to be compared. While most are familiar with the idea of an alphabet, we still dedicate an entire chapter to this simple yet extremely useful construct. We abstract away the alphabet into an independent class. A dictionary then encapsulates an instantiation of the alphabet object and invokes it to operate over the sequences it contains. This setup allows for easy adoption of new user-defined alphabets at run-time, attributing high flexibility to our program.

5.2 Definition

An alphabet Σ is defined over a set of characters

$$C = \{c_i \mid 0 \leq i < n\}$$

and symbols

$$S = \{s_j \mid 0 \leq j \leq |\Sigma| < n\}.$$

In general n is unbounded, but in practice it is not; following the standard C++ convention, we set n to 256 by storing each character in a single-byte **unsigned char**. This definition is not complete without a mapping between C and S. Each character c_i maps to exactly one symbol s_j, its encoding $s_j = e(c_i)$; multiple characters may map to the same symbol. Each symbol s_j maps to exactly one character c_i, its decoding $c_i = d(s_j)$; this requires that $s_j = e(c_i)$. Thus, we allow multiple characters to stand for the same symbol, with only a single character acting as that symbol's normal representation. A dictionary of size $|\Sigma|$ consists of $|\Sigma| + 1$ symbols. All characters whose mappings are explicitly defined cover the symbols s_1 through $s_{|\Sigma|}$; all other characters implicitly map to the invalid symbol s_0, with $d(s_0)$ set to a special user-defined gap character.

5.3 Usage

The above-mentioned framework allows us to define alphabets rapidly. Consider the biological alphabet of DNA with $S = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$. We could set up a case-insensitive mapping as follows:

$$e(\mathrm{A}) = e(\mathrm{a}) = s_1 = \mathtt{A}; \quad d(\mathtt{A}) = \mathrm{A},$$

$$e(\mathrm{C}) = e(\mathrm{c}) = s_2 = \mathtt{C}; \quad d(\mathtt{C}) = \mathrm{C},$$

$$e(\mathrm{G}) = e(\mathrm{g}) = s_3 = \mathtt{G}; \quad d(\mathtt{G}) = \mathrm{G},$$

$$e(\mathrm{T}) = e(\mathrm{t}) = s_4 = \mathtt{T}; \quad d(\mathtt{T}) = \mathrm{T},$$

with all other characters mapping to $s_0 = -; d(-) =$ -. Specifying an alphabet is quite simple. One has to create a text file consisting of several strings. The first character of the first string is used to represent the gap. All subsequent strings must contain at least two characters. Their first two characters c and s specify a mapping $e(c) = s$ and set $d(s) = e$ if $d(s)$ has not already been set. For example, one could specify the DNA alphabet from above in a file `dna.alpha` as follows:

```
-
AA  aA
CC  cC
GG  gG
TT  tT
```

Table 5.1: DNA Alphabet Specified in a Text File `dna.alpha`

A slightly less trivial encoding could be used to equate complementary strands of DNA by setting

$$e(\mathrm{A}) = e(\mathrm{a}) = e(\mathrm{T}) = e(\mathrm{t}) = s_1 = \mathtt{X}; \quad d(\mathtt{X}) = \mathrm{x},$$

$$e(\mathrm{C}) = e(\mathrm{c}) = e(\mathrm{G}) = e(\mathrm{g}) = s_2 = \mathtt{Y}; \quad d(\mathtt{Y}) = \mathrm{y},$$

with the gap defined as before. Of course, more interesting variations are possible. Characters could be classified and grouped together by mass, size, polarity, or other properties when dealing with strings of amino acids and richer sequences.

If the dictionary were created over the English alphabet, it may be useful to mask letters into consonants and vowels, for instance. It should be clear that some interesting and powerful effects arise from the ability to invoke simple substitutions without a hassle. To achieve deeper effects, one may opt to pre-process the input and parse out token characters using more sophisticated techniques before building a dictionary. Still, the fact that the dictionary operates over arbitrary alphabets is quite convenient. For example, a linguist may be interested in defining a phonetic alphabet, compiling pronunciations of words from several languages, and using our dictionary to find like-sounding fragments.

5.4 Internal Representation

We represent the dictionary in $3 \times 256 + 4 = 772$ bytes, assuming 32-bit **unsigned long int**s; a single such integer contains the size of the dictionary $|\Sigma|$. The rest of the memory is distributed among three arrays of 256 **unsigned char**s each. Arrays `encode`[256] and `decode`[256] capture the mappings $e(\cdot)$ and $d(\cdot)$:

$$s_j = e(c_i) \rightarrow \texttt{encode}[i] == j,$$

$$c_i = d(s_j) \rightarrow \texttt{decode}[j] == i,$$

with `decode`[j] == 0, $\forall j > |\Sigma|$, and `decode`[0] set to the gap. This information would be sufficient, but for efficiency's sake we also include a filter:

$$\texttt{filter}[i] == \texttt{decode}[\texttt{encode}[i]],\ 0 \leq i < 256.$$

Alphabet objects are immutable. An alphabet may be instantiated given a file pointer. The default constructor assumes that the file contains strings, as described earlier:

```
Alphabet (FILE * f) throw (char const * const);
```

A more efficient constructor simply reads 516 bytes from the file, interpreting them as the size of the alphabet, followed by `encode` and `decode` (`filter` is reconstructed in the obvious way):

```
Alphabet (gzFile gzf) throw (char const * const);
```

However, this constructor is intended to be used only with files created by an alphabet object using the internal `Save` function:

```
void Save (gzFile gzf) const throw (char const * const);
```

There is also a trivial destructor:

```
~Alphabet () throw ();
```

5.5 Functionality

We endow the alphabet class with several useful routines. The function

```
inline unsigned long int Size (void) const throw();
```

returns the size of the alphabet. Functions

```
inline unsigned long int Encode   (   char *seq,
                                      unsigned long int const len
                                  )   const throw ();
```

and

```
inline unsigned long int Encode   (   char const *seq,
                                      unsigned char *dst,
                                      unsigned long int const len
                                  )   const throw ();
```

write the encoding of the first `len` characters in `seq` onto either `seq` itself or onto `dst` respectively and return the number of valid (non-zero) symbols encountered; the encoding of a character c is `encode`$[c]$. Functions

```
inline void Decode (char *seq, unsigned long int const len) const throw ();
```

and

```
inline void Filter (char *seq, unsigned long int const len) const throw ();
```

overwrite the first `len` characters in `seq`, replacing each character c with either `decode`$[c]$ or `filter`$[c]$, respectively. More interesting routines follow.

The function

```
inline void getTuple   (   char *const str,
                           unsigned long int i,
                           unsigned long int const t
                       )   const throw ();
```

writes the ith lexicographically ordered t-tuple in `str`, assuming $i < |\Sigma|^t$ of course. That is, the function represents i in base $|\Sigma|$, with $s_1, \ldots, s_{|\Sigma|}$ serving as digits $0, \ldots, |\Sigma| - 1$, represented by their corresponding characters `decode`$[1], \ldots,$`decode`$[|\Sigma|]$. When the function returns,

$$\sum_{d=0}^{t-1} (\texttt{decode}[\texttt{str}[(t-1)-d]]-1)|\Sigma|^d = i,$$

while the function

```
inline void nextTuple  ( char * const str,
                         unsigned long int const t
                       ) const throw ();
```

replaces the t-tuple i in `str` with the next lexicographically ordered t-tuple $j = i + 1$; if $i + 1 = |\Sigma|^t$, j wraps around to zero. Also,

```
inline unsigned long int indexTuple  ( char const * str,
                                       unsigned long int const t
                                     ) const throw ();
```

returns the value i of the t-tuple represented in `str`. Finally, the function

```
inline unsigned long int MakeTupleList  ( char const * seq,
                                          unsigned long int const t,
                                          unsigned long int const len,
                                          unsigned long int * list
                                        ) const throw (char const * const)
```

replaces `list`$[i]$ with the value of the t-tuple ending in the ith character in `seq` for $0 \leq i \leq$ `len`. Invalid tuples are marked with the special value $|\Sigma|^t$, which is also the return value of the function. For $i \in \{0, \ldots, t-1, \texttt{len}\}$, `list`$[i]$ is always set to this special value.

These functions abstract away the details inherent to particular alphabets. For example, they allow the user to tabulate statistics like frequency counts of tuples of various lengths in a set of sequences quickly and painlessly. All functions are limited by finite-precision arithmetic and rely on $|\Sigma|^t \leq$ `ULONG_MAX`; an exception is thrown if this assumption is violated, of course.

Chapter 6

Builder

6.1 Overview

The builder is an executable which actually creates a dictionary, performing the pre-computations necessary to make future queries efficient. We explain how one should use the builder first and provide several insights into its implementation later. Both descriptions are straight-forward.

6.2 Usage

```
val> ./Builder

==================================
Dictionary Builder v9.11, May 2000
Valentin I. Spitkovsky (val@theory.lcs.mit.edu)

MIT Laboratory for Computer Science
Theory of Computation: Computational Biology
Gene Recognition Group
======================

usage: ./Builder -lL -aA -zZ <alphabet> <dictionary> < <database>

       where

                  -l sets the subdictionary length to L
                  -a sets the length of the accession code (tag) to A
                  -z sets the gzip compression level to Z

          <alphabet> is the source file for the alphabet
        <dictionary> is the name of the target dictionary
          <database> is the source file of tag-sequence pairs

val>
```

Figure 6-1: Builder Usage

6.2.1 <database>

The main source of raw data for the dictionary is captured in this parameter, which is assumed to name a text file consisting of $2n$ strings, $n > 0$. Each pair of strings represents a sequence: the first is that sequence's name or accession code; the second is the sequence itself. We have been careful to make sure that the builder would need to make only a single pass through the database, which allows the database to be read from standard input. As a result, the user is free to connect the output of another process with the input to the builder through a pipe. This is particularly useful when databases get large. For instance, the database of expressed sequence tags `dbEST` contains many gigabytes of raw data. Naturally, most prefer to keep it compressed. Having the builder accept the database from standard input allows the user to build a dictionary without the hassles of freeing up large chunks of disk space and uncompressing the database there. Instead, the user may take advantage of a pipe. Assuming that the database was compressed with `gzip` and exists in compressed form as `<database>.gz`, the builder could be invoked with

```
gunzip -c <database>.gz | Builder -lL -aA -zZ <alphabet> <dictionary>,
```

as opposed to

```
Builder -lL -aA -zZ <alphabet> <dictionary> < <database>.
```

If the database were spread over multiple files, they could be concatenated and piped into the builder with `cat`. More creative uses of the feature could include various text preprocessors and parsers. The main benefit of pipes, in this case, is avoiding resorting to disk for temporary storage; the savings in time and space tend to become substantial when databases get large.

6.2.2 <alphabet>

This parameter specifies the text file that defines the alphabet over which the dictionary — and all of its future queries — will operate. For more on how to create alphabet descriptions, see table 5.1 along with the surrounding text.

6.2.3 <dictionary>

All of the files that make up the newly created dictionary will start with this parameter. It is also the handle for this dictionary expected by the constructor for future instantiations of dictionary objects encapsulating the information in `<database>` interpreted over the `<alphabet>`. Naturally, we recommend highly descriptive names. For example, `~/Dictionaries/dbEST.11131999` may be a good name and location for the dictionary of expressed sequence tags released on November 13, 1999.

6.2.4 `-aA`

Since each sequence has a name, the dictionary incorporates this information. However, names may tend to be long. In an attempt to not waste space while reserving the ability to retrieve accession codes quickly, the builder stores at most `A` of the first few characters of a sequence's name, for a total of n`A` bytes (the builder pads shorter accession codes). We assume that for most practical purposes `A` will tend to be rather small.

6.2.5 `-lL`

As we hinted earlier, databases may get extremely large. In our experience with `dbEST`, we have had to push the operating system to the limit. The problem is that 2^x-bit operating systems have trouble with files longer than $2^x - 1$ bytes. Today's Linux is based on a 32-bit architecture and uses a signed integer for addressing purposes. At $2^{31} = 2147483648$, the numbers wrap around to $-2^{31} = -2147483648$, which sets the stage for some very interesting behaviors[1]. We avoid dealing with this 2Gb limit by splitting dictionaries into subdictionaries, none of which require files exceeding `L` words.

In practice, this limit is also nice because `L` could be set to take advantage of the configuration of the machine executing the code. Once a process runs out of memory, swapping may severely degrade performance. We find that setting `L` just a tad below one fifth of the available RAM leads to very efficient building (on our 300Mb 32-bit machine, we set `L` to 50,000,000). If building speed is not an issue, `L` could be set much larger to improve amortized query time, however `L` certainly should not exceed the minimum of 2^{31} and the amount of available RAM. Optimum `L` would more than likely be below this minimum. Necessarily, no sequence in the database may be longer than `L`; we also require that $\mathtt{A} \leq \mathtt{L}$, which seems like a reasonable restriction.

6.2.6 `-zZ`

We expect the accession codes and sequences to compress rather well. Therefore, we provide an option to store these files in zipped form. The parameter `Z` corresponds directly to the `gzip` compression level. It is an integer between zero and nine, with zero implying no compression and nine representing the maximum memory to speed tradeoff. We outsource all compression routines to the `zlib` library [32].

6.3 Internal Representation

The builder scans the `<database>`, collecting sequences for the dth subdictionary, starting with $d = 0$. Let c^d be the number of sequences in this subdictionary and let l_i^d be the length of each of those sequences, for $0 \leq i < c^d$. At some point, the builder will either exhaust

[1]Writing in the year 2000, this author finds the predicament amusing.

the database or reach a point where

$$1 + c^d + \sum_{i=0}^{c^d-1} l_i^d > \mathtt{L} + 2,$$

in which case the next subdictionary will be started, with the last sequence read serving as its first sequence. Assuming that no sequence is longer than L characters, every sequence fits into some subdictionary; of course, if the builder encounters a longer sequence, it immediately alerts the user of the problem.

Let s_i^d for $0 \leq i < c^d$ represent the sequences in subdictionary d. Then the layout for that dictionary — a contig in memory — is

$$0 \cdot s_0^d \cdot 0 \cdot s_1^d \cdot 0 \cdot \cdots \cdot s_{c^d-1}^d \cdot 0,$$

containing

$$L^d = 1 + c^d + \sum_{i=0}^{c^d-1} l_i^d$$

characters. The builder then encodes the contig using the alphabet's `Encode` function and sets $\bar{z}^d$ to be number of non-zero characters in contig afterwards. The builder spreads the information in the subdictionary over five files.

6.3.1 `<dictionary>.`d`.acc`

The builder stores the accession codes for the c^d sequences, each padded up to A bytes with zeros, in this file, for a total of $\mathtt{A}c^d$ bytes. The builder compresses this data before writing it to disk.

6.3.2 `<dictionary>.`d`.seq`

The builder then stores the encoded contig in this file, for a total of L^d bytes. The builder compresses this data before writing it to disk.

6.3.3 `<dictionary>.`d`.idx`

In order to be able to tell later where one sequence begins and another one ends, the builder stores the positions of the boundaries — the zeros wedged into the unencoded contig — in this file[2]. Its length is $4(c^d + 1)$ bytes, again assuming a 32-bit architecture.

[2]Although the first number is always one, its presence eliminates the need for an explicit sentinel during binary search when trying to pinpoint the index of a sequence to which a position in contig belongs.

6.3.4 `<dictionary>.`d`.pos`

The bulk of the work done by the builder's pre-processor rests in this file. It contains $\bar{z}^d$ integers, each representing a position in contig containing a non-zero character. Each position represents a subsequence that begins at the corresponding offset in the contig and runs until the next zero. In this file, positions are ordered lexicographically by the sequences that they represent, according to the alphabet. This is the biggest file in the subdictionary, taking up $4\bar{z}^d < 4(L^d - 2)$ bytes, again assuming 32-bit **unsigned long int**s.

6.3.5 `<dictionary>.`d`.smp`

To avoid swapping in fragments of the `.pos` file whenever possible during binary search, we store in this file its sampled version. In `.smp` we save only the last word of each page from `.pos`. Assuming 4Kb pages, each containing 1,024 words, this file is quite a bit smaller than its more detailed companion. However, it still orders all of its positions lexicographically by the sequences that they represent, according to the alphabet. Modest in size, this file takes up only $\lceil \frac{\bar{z}^d}{256} \rceil$ bytes.

6.3.6 `<dictionary>.inf`

This file contains the information pertaining to the overall dictionary. It consists of the alphabet, the value of the maximum accession code length `A`, and a triplet of integers for each subdictionary. Assuming D subdictionaries make up a dictionary, the `.inf` file contains a total of

$$516 + (\texttt{sizeof}(\textbf{unsigned long int}))(2 + 3D) = 524 + 12D \text{ bytes},$$

given 32-bit **unsigned long int**s. The three values represent c^d, L^d, and $\bar{z}^d$. The builder compresses this file before writing it to disk.

6.4 Complexity Bounds

The streamlined builder scans through the database only once, writing the `.inf`, `.acc`, and `.idx` files to disk on the fly; the `.seq` file is written out as soon as it is encoded, requiring only a single pass through that data, once it's been read. No asymptotic waste here. However, the entire encoded contig is stored in memory, taking up L^d bytes, $\bar{z}^d$ of which are non-zero. The builder then maps $\bar{z}^d$ words of disk space onto memory for `.pos`, along with a much smaller chunk for `.smp`. Having ordered the positions, the builder writes out the samples in a single pass. The amount of base memory involved is just what is needed to store the contig and the positions, assuming the user would like to avoid swapping. This requires $L^d + 4\bar{z}^d$ bytes on a 32-bit operating system and is under $5L^d$ bytes. So far, the builder has not used more than the essential $\Theta(L^d)$ memory and $\Theta(L^d)$ time needed to read in the input and write out the answer.

The crux of the computationally- and memory-intensive work lies in sorting the positions file. We invoke `Dictionary::SubStringSort`, taking advantage of the sorting routines developed in chapter 3, to achieve expected bounds of

$$O(\frac{\bar{z}'^d}{\alpha^2} + \alpha^2) \text{ and } O(\bar{z}'^d + \alpha^2)$$

in extra space and time respectively, with $\bar{z}'^d < \frac{\bar{z}^d}{2}$. In the worst case,

$$\Theta(\bar{z}'^d + \alpha^2)$$

extra space will be required. However, the amount of extra time that could be taken is no longer tightly bounded by

$$O((\bar{z}'^d)^2 + \alpha^2),$$

since multiple sequences make up the input. At worst, c^d valid segments of length l_i^d each face the builder. Let l'^d_i represent the number of characters in segment i that actually need to be sorted. Recall that

$$\bar{z}' = \sum_{i=0}^{c^d-1} l'^d_i < \frac{1}{2}\sum_{i=0}^{c^d-1} l_i^d = \frac{\bar{z}}{2},$$

with $l'^d_i < l_i^d$. Also, let the zero/one indicator function $f_{\geq}(x,y) = 1$ if and only if $x \geq y$. If the longest segment contains

$$m = \max_{k=0}^{c^d-1} l'^d_k$$

characters to be sorted, then the worst-case running time is bounded by

$$\sum_{j=0}^{m}\sum_{i=0}^{c^d-1} f_{\geq}(l'^d_i, j) = O(m\bar{z}'^d).$$

Chapter 7

Dictionary

7.1 Overview

A dictionary object encapsulates an alphabet, a set of sequences interpreted over that alphabet, along with the sequences' accession codes. The dictionary stores this information, together with pre-computed data derived from it, in a way that facilitates efficient execution of certain queries. As with the builder, we explain how one should use the dictionary first and provide several insights into its implementation later.

7.2 Usage

To instantiate a dictionary object in a C++ program, having actually built the dictionary, one need only know its path or handle. That is, the `<dictionary>` parameter passed to the builder, or its equivalent, should also be passed to the constructor:

```
Dictionary (char const * const stem) throw ();
```

The dictionary also possesses a trivial destructor

```
~Dictionary () throw ();
```

which simply closes the batch, if one has been started but hasn't been closed (see below).

7.2.1 Basic Functions

Some very basic but useful functions follow:

```
inline unsigned long int accCodeLength (void) const throw ();
```

returns the value of the maximum accession code length `A`,

inline unsigned long int `numSequences` (**void**) **const throw** ();

returns the total number of sequences contained in the dictionary,

inline long double `numValidChars` (**void**) **const throw** ();

returns the total number of nonzero encodings in the sequences of the dictionary,

inline long double `mem` (**void**) **const throw** ();

returns the number of (uncompressed) bytes taken up by the files of the dictionary,

void `getStats` (**long double** & `sx`, **long double** & `ssx`) **throw** ();

stores in `sx` and `ssx` the basic statistics for the lengths of the sequences in the dictionary:

$$\texttt{sx} = \sum_{i=0}^{n-1} l_i \text{ and } \texttt{ssx} = \sum_{i=0}^{n-1} l_i^2,$$

with n equal to `numSequences()` and l_i representing `lenSequence(`i`)`. Unimaginatively,

unsigned long int `lenSequence` (**unsigned long int const** `index`) **throw** ();

returns the length of the ith sequence in the dictionary, $i =$ `index`; zero is returned if $i \geq n$. In addition,

void `getSequenceName` (**unsigned long int const** `index`, **char** * **const** `name`) **throw** ();

replaces the first `A` characters in `name` with the accession code of the ith sequence, but only if $0 \leq i < n$; also,

void `getSequence` (**unsigned long int const** `index`, **char** * **const** `seq`) **throw** ();

replaces the first l_i characters in `seq` with the decoded ith sequence and sets `seq`$[l_i + 1]$ to zero to terminate the string; for $i \geq n$, the function acts as if $l_i = 0$.

7.2.2 Query Processing

The only query permitted by the dictionary reports all matches, no shorter than a threshold length, between an input sequence and the sequences in the database. To amortize the costs of running a query, we introduce the concept of a batch, which allows the user to query multiple sequences against the database at once. For each match, the dictionary reports five values:

Value	*Interpretation*
`is`	the index of the input sequence
`isp`	the position of the match in the input sequence
`ds`	the index of the database sequence
`dsp`	the position of the match in the database sequence
`hl`	the length of the match (hit length)

Table 7.1: A Match

Since the number of matches fluctuates from query to query, there is no best way to report them. Returning arrays of matches may work well up to a point, but could exhaust the memory. The user, on the other hand, may wish to store the matches to disk, send them over the network, or deal with them on the fly in other ways. Also, the user may not be interested in all of the data returned. For instance, the length of the match could be easily reconstructed given the other values; if the batch or the database contained only a single sequence, then maintaining lists of sequence indices would be pure waste. We keep the dictionary flexible by defining an abstract `DicUser` (dictionary user) class, specifying only a single function:

```
virtual inline void yield   (   unsigned long int const is,
                                unsigned long int const isp,
                                unsigned long int const ds,
                                unsigned long int const dsp,
                                unsigned long int const hl
                            );
```

to be called upon by the dictionary to handle communication.

We are now ready to discuss batches. The user may start a new batch by calling

```
void startBatch   (   unsigned long int const cutoff,
                      unsigned long int const maxlen,
                      DicUser * const user
                  )   throw ();
```

which first of all closes the batch, if one has been started but hasn't been closed (see below), and starts a new one. No matches shorter than $\max(1, \texttt{cutoff})$ will be reported. The dictionary will process a query when the user requests that the batch be closed or when the cumulative length of the sequences submitted exceeds a threshold value[1]. No sequence in the batch may exceed `maxlen` characters in length. The function

```
user->yield(is,isp,ds,dsp,hl)
```

will be used to report matches. Sequences may be added to the batch via

[1]The program builds a mini-dictionary out of the sequences in the batch. Here, `maxlen` serves as the parameter `L`. The same considerations pertaining to the choice of `L` apply to the choice of `maxlen`; see section 6.2.5. Having run the query, the dictionary starts an identically parameterized new batch with the sequence that broke the threshold as its only member. A separate counter keeps track of the number of sequences processed to get the sequence indices right.

```
void addtoBatch ( char const * const seq,
                  unsigned long int const len
                ) throw (char const * const);
```

The function throws an exception if `len` exceeds `maxlen`. Finally, the user may close a batch, requesting receipt of all satisfactory matches by calling

```
void closeBatch (void) throw ();
```

A word of caution: the query may begin reporting matches before this function is invoked; `user` must be prepared to accept these matches just before the second call to `addtoBatch`. While we guarantee to return all satisfactory matches, we do not impose any ordering restrictions; `user` must be prepared to deal with matches returning in any of the number of possible permutations of orders.

7.2.3 Internal Functions

Three internal functions end up public because the builder uses them. While we intend for them to remain internal, we take a moment to describe them here for completeness and to make life simpler for those adventurous enough to incorporate them in other code. These are:

```
void BinarySearch ( long int const togo,
                    long int const eleft,
                    long int const eright,
                    unsigned char const ea,
                    unsigned char const ez,
                    long int const cleft,
                    long int const cright,
                    unsigned char const ca,
                    unsigned char const cz
                  ) throw ();
```

which finds matches between two dictionaries and is described in detail in chapter 4;

```
static void Dictionary::SubStringSort ( unsigned char const alphasize,
                                        unsigned long int const minlen,
                                        unsigned char * const contig,
                                        unsigned long int const contiglen,
                                        unsigned long int * const disk,
                                        unsigned long int const disklen
                                      ) throw ();
```

which sorts all of the substrings within a string and is explained carefully in chapter 3; and

```
static void Dictionary::PageSample ( unsigned long int const * const array,
                                     unsigned long int const len,
                                     FILE * const f,
                                     int const fd,
                                     char const * const msg,
                                     char const * const src
                                   ) throw (char const * const);
```

which writes in the file f (with file-descriptor `fd`) the sampled version of an array of length `len`. This routine captures in f the last word of each page from `array` and is used to create the `.smp` files described in section 6.3.5.

7.3 Implementation Details

From the length of the (uncompressed) `.inf` file, the dictionary quickly determines the number of subdictionaries D. Having instantiated the appropriate alphabet and determined the maximum allowed accession code length `A`, the dictionary loads c^d, L^d, and $\bar{z}^d$ for each subdictionary $0 \leq d < D$; these are the number of sequences, their cumulative length (plus one), and the number of nonzero encodings in subdictionary d. The dictionary then constructs the running sum

$$R(d) = R(d-1) + c^{d-1},$$

for $1 \leq d \leq D$, with $R(0) \equiv 0$, and also maps all of the uncompressed subdictionary files (`.idx`, `.pos`, and `.smp`) onto memory. This lets the dictionary access all of the necessary files as if they were arrays. The operating system takes care of intelligent caching, swapping, and other chores using its virtual memory machinery. Compressed files (`.acc` and `.seq`) are unzipped directly into memory from disk when they are first needed. Although mmap would have allowed the dictionary to swap in only the pieces of these files that are actually used, because these files compress well we trade this convenience for the opportunity to significantly reduce the amount of disk space involved. With these in place, the most basic functions become trivial: `accCodeLength` simply returns `A`, `numSequences` returns $R(D)$, `numValidChars` returns $\sum_{d=0}^{D-1} \bar{z}^d$, `mem` returns the total uncompressed lengths of all dictionary files, and `getStats` simply cycles over all sequences, adding up `lenSequence` and its square.

Dealing with routines that specify a sequence index i is a bit tricker. The index must be broken down into the appropriate subdictionary d and the index within the subdictionary i^d: d is the maximum integer from $\{0, \ldots, D-1\}$ such that $R(d) \leq i$; $i^d = i - R(d)$. We find d in $\Theta(\lg D)$ time via binary search over R. Having determined these values, the three sequence-specific functions become quite manageable as well (we assume here that $0 \leq i < R(D)$): `lenSequence` returns

$$.d.\mathtt{idx}[i^d + 1] - .d.\mathtt{idx}[i^d] - 1,$$

`getSequenceName` copies `A` characters from

$$.d.\texttt{acc} + \texttt{A}i^d$$

onto `name`, and `getSequence` uses the alphabet to decode `lenSequence` characters from

$$.d.\texttt{seq} + .d.\texttt{idx}[i^d]$$

onto `seq` and caps it with a null-terminator.

Of course, the real work accumulates in the three functions dealing with batches. Together, they act very much like the builder. That is, `startBatch` initiates the building process; `closeBatch` behaves as the end-of-file signal. As we mentioned before, `maxlen` takes the place of `L`; the `A` parameter is no longer necessary as the batch does not keep track of sequence names. Only the `.pos` and `.smp` files are actually written. In addition, a vector maintains `.idx` and an array of characters retains `.seq`. We do not construct `.acc`. There are other crucial differences. As `addtoBatch` receives incoming sequences, we do some pruning. If a sequence contains segments of valid characters shorter than `cutoff`, those segments are masked with zeros — there is no reason to sort them since none of them could be a part of a match. If after masking no nonzeros remain in the sequence, we pretend that the sequence had been empty. We cannot act as if the sequence never arrived at all — that would mess up the indices of subsequent entries — but we don't have to take up more than a single character in `.seq` for it. Finally, we resort to the variation of the substring sort problem from chapter 3. When time comes to actually perform a query, we only sort the positions based on the first `cutoff` characters, leaving out of `.pos` positions of substrings shorter than `cutoff` in length. We proceed to forage for matches by running `BinarySearch` from chapter 4 on all d subdictionaries. Each match

$$(\texttt{is} = 0, \texttt{isp}, \texttt{ds} = 0, \texttt{dsp}, \texttt{hl})$$

must be modified to take into account the fact that there are many sequences involved. Suppose N sequences from this batch have already been processed, with the current batch working on the next n. Let x be the largest integer below n such that

$$.\texttt{idx}[x] \leq \texttt{isp};$$

analogously, let y be the largest integer below c^d such that

$$.d.\texttt{idx}[y] \leq \texttt{dsp}.$$

Then the true match should be

$$(N + x, \texttt{isp} - .\texttt{pos}[x], R(d) + y, \texttt{dsp} - .d.\texttt{pos}[y], \texttt{hl}),$$

which is what `user->yield` returns. We take $\Theta(\lg n)$ time to find x and $\Theta(\lg c^d)$ time to find y via binary search over `.idx` and `.`d`.idx`.

7.4 Complexity Bounds

As the dictionary executes a query of an additional n sequences of length l_{N+i}, for $0 \leq i < n$, against a subdictionary d, it finds m matches. Suppose that the sum of the lengths of these m matches is l and let C equal the `cutoff` value and

$$L = \sum_{i=0}^{n-1} l_{N+i}.$$

Then it takes an expected

$$\Theta\left(l + |\Sigma|^2 + L\left(\min\left(C, \frac{\lg L}{\lg |\Sigma|}\right) + \lg\left(\frac{c^d}{L}\right)\right) + m\left(\lg n + \lg c^d\right)\right)$$

time and

$$\Theta(\frac{L}{|\Sigma|^2} + |\Sigma|^2)$$

extra space to execute the query, see section 4.4. The extra addend of $m(\lg n + \lg c^d)$ comes from the fact that each match must be post-processed to adjust the sequence indices and position offsets properly. In practice, this time is negligible.

Chapter 8

Best Cartesian Pairs

8.1 Definition of the Problem

Although the problem considered in this chapter does not relate to the rest of the dictionary, it poses an algorithmic challenge that arises in the dynamic programming algorithm in the gene recognition program developed in our group. At some point, the engine subdivides a region into two subregions, runs dynamic programming on each, and attempts to reassemble optimal parses of the entire region using the sub-parses from each subregion. As a result, given two arrays A and B of length a and b respectively, we consider the pairs from $A \times B$. The challenge is to find the c best pairs (A_i, B_j). By best, we mean those pairs that have the highest sums $A_i + B_j$. We present in this section an efficient algorithm for finding c such highest-scoring pairs from $A \times B$ given the scoring function $f_+(x, y) = x + y$; correctness holds for arbitrary scoring functions $f(x, y)$ monotonically increasing in both x and y.

8.2 Brute Force Attempts

Clearly, one could easily solve this problem head-on by generating all pairs in $A \times B$ and sorting them. Given ab pairs, no such algorithm could perform asymptotically better than $\Theta(ab)$ time and space. Daniel J. Kleitman suggested a slightly more efficient approach: having sorted the arrays A and B and given a threshold sum s, one could easily determine whether or not the number of pairs summing up to s or more exceeds c in $\Theta(c)$ time. One could then run binary search on the range of values in $f(A_i, B_j)$. Assuming that one could sort the arrays A and B in $\Theta(a + b)$ time, the entire algorithm would take

$$\Theta\left(a + b + c \lg\left(\left(\max_{i=0}^{a-1} A_i + \max_{j=0}^{b-1} B_j\right) - \left(\min_{i=0}^{a-1} A_i + \min_{j=0}^{b-1} B_j\right)\right)\right)$$

time and $\Theta(a + b + c)$ space. Nicholas T. Ingolia came up with a very curious solution to achieve $\Theta\left(a + b + c \lg^2 c\right)$ time and $\Theta(a + b + c)$ space.

8.3 An Optimal Algorithm

Assuming that the elements of A and B consist of integers, easily sortable in linear time via *RadixSort*, we present an optimal algorithm that achieves

$$\Theta\left(a + b + c\right)$$

time and space bounds. If the elements could not be sorted in $\Theta(n)$ time, we could rely on traditional $\Theta(n \lg n)$ sorting algorithms like *MergeSort* and *Randomized QuickSort*, resulting in $\Theta\left(a \lg a + b \lg b + c\right)$ time requirements, which may no longer be optimal.

First, we sort A and B, modifying the algorithm to expect arrays sorted in descending order. We then reduce the problem of finding the c highest scoring pairs in arrays of lengths a and b to a smaller problem of finding $\frac{c}{2}$ such pairs in shorter arrays of length $\frac{a}{2}$ and $\frac{b}{2}$. Let A' and B' consist of only the even elements of A and B, with $A'_i = A_{2i}$ and $B'_j = B_{2j}$. Suppose we knew the $\frac{c}{2}$ highest scoring pairs in $A' \times B'$,

$$\{(A'_i, B'_j)\}.$$

We claim that the c highest scoring pairs in $A \times B$ all belong to

$$\{(A_{2i}, B_{2j}), (A_{2i+1}, B_{2j}), (A_{2i}, B_{2j+1}), (A_{2i+1}, B_{2j+1})\}.$$

To extract these c pairs, we could sort the $4\frac{c}{2} = 2c$ potential candidates in linear time; if that were not possible, we could find the cth largest element of that set in linear time and then proceed to partition the set using that element as the pivot. Let $T(a, b, c)$ be the running time for the original problem, and let $T'(a, b, c)$ be the running time for the modified version, in which the input arrays have been sorted. Then the recurrences wind up being

$$T(a, b, c) = \Theta(a) + \Theta(b) + T'(a, b, c),$$

with

$$T'(a, b, c) = T'(\frac{a}{2}, \frac{b}{2}, \frac{c}{2}) + 2c = \Theta(c).$$

Here, $T'(a, b, 1) = \Theta(1)$ because the answer to that problem is just (A_0, B_0); also, since ab shrinks faster than c, if the case arises where $c \geq ab$, then the algorithm simply returns all pairs ab, taking $T'(a, b, c) = \Theta(ab) = O(c)$ time. Of course, there is no need to recurse from c all the way down. Instead, the procedure could be implemented in reverse.

8.4 Proof

	A_0	A_1	A_2	A_3	A_4	A_5	A_6	A_7
B_0	**o**	x	**o**	x	**o**	z_1'	z_1	·
B_1	x	·	x	·	x	·	·	·
B_2	**o**	·	**o**	·	**o**	·	·	·
B_3	x	·	z_2'	·	·	·	·	·
B_4	**o**	·	·	·	·	·	·	·
B_5	·	·	·	·	·	·	·	·
B_6	·	·	z_2	·	·	·	·	·
B_7	·	·	·	·	·	·	·	·

We present our argument pictorially. Let **o** represent the $\frac{c}{2}$ pairs that solve the subproblem. If there is an **o** in a row other than the first, then there is an **o** directly above it, since that pair would have to score higher as the arrays A and B are now sorted. No pair (A_{2i}, B_{2j}) not part of that solution could score higher than the pairs that were returned. Consequently, no pair to the right, just below, and diagonally from a rejected pair could score higher either. We prove that the $\frac{c}{2}$ pairs forming the solution and their three immediate neighbors must contain the c highest scoring pairs by showing that among the neighbors of the pairs in the solution there must be at least $\frac{c}{2}$ pairs scoring higher than the rejected pairs; consequently, these extra $\frac{c}{2}$ pairs must score higher than the rejected pairs' neighbors as well. As a result, the $2c$ elements consisting of the pairs in the solution of the subproblem and their neighbors contain the c pairs that form the solution for the overall problem.

Clearly, any pair between two **o**s could not score lower than either of them and therefore could not score lower than any of the rejected pairs. We label such pairs to the right of all but the right-most **o** in the top row and above all **o**s in all other rows with xs. Observe that there are $\frac{c}{2} - 1$ such xs, all guaranteed not to score lower than any of the rejected pairs. We now show that for all rejected pairs z there exists another pair z', distinct from the xs, which could not score lower than z. That z' is the closest pair adjacent to an **o** located either directly above z or to the left of z. We point out such pairs z_1' and z_2' for z_1 and z_2. Necessarily, a $\bar{z}'$ exists even for the highest-scoring rejected pair $\bar{z}$. Therefore, the c highest scoring pairs must reside in the $2c$ locations consisting of the $\frac{c}{2}$ pairs forming the solution to the subproblem and their immediate neighbors to the right, just below, and diagonally.

The Human Genome Project Logo provided courtesy of the Human Genome Management Information System.

Part II

Applications

Chapter 9

Homology

9.1 De-Facto Standard: BLAST

One of the major applications for the dictionary we have found lies in identifying related genetic material. Since the dictionary approach presents neither the first nor the major way for accomplishing the task, we first describe the main-stream tool. The Basic Local Alignment Search Tool (BLAST [1]) is the accepted standard of finding similar regions between sequences. It stems from the alignment algorithm by Smith and Waterman [27], which some consider to be the first major contribution of computer science to biology.

We briefly outline the main idea behind alignments. Consider two sequences s_0 and s_1, with lengths $L(s_i)$. The concept of an alignment captures the notion that one could insert gaps into sequences s_i to produce s_i', subject to the constraint that $L(s_0') = L(s_1')$. One could then associate a score S with each alignment:

$$S = \sum_{k=0}^{L(s_0')-1} f(s_0'[k], s_1'[k]),$$

using a scoring function f. Typically, one scores a 1 for a character lined up with a matching character, a -1 for a character lined up with a mismatching character, and a small number like .1 for a character lined up with a gap. A gap lined up with a gap scores a zero, since both gaps need not be in the alignment. Given two sequences and a scoring function f, Smith and Waterman find in [27] the alignment that maximizes S in $\Theta(n \lg n)$ time and $\Theta(n)$ space. Here, $n = L(s_0) + L(s_1)$.

9.2 Alignment Concerns

We have several major concerns with the de-facto standard. First of all, while the values of the scoring-function f make intuitive sense, they are not founded in any theory and seem to have been pulled out of thin air. It should be clear that slight perturbations to f could often change the resulting "best" alignment significantly. In practice, alignments tend to hide information. Consider two long sequences, mostly unrelated, with a relatively short exact match in the middle. The S-maximizing alignment may thin out the real match to score more points elsewhere, rather than align that region precisely and miss many potential points outside.

Having hinted at some fuzzy cases above, we now observe the harsh reality that alignments miss entire classes of homologous regions. Let $s_0 = A^n B^n$ and $s_1 = B^n A^n$. No alignment could recognize that there are two regions that match here: the string of As and the string of Bs. At best, an alignment would find only one. While many mutations like insertions and deletions play right into the concept of alignments, other common biological operations like translocation and cross-over simply do not lend themselves to the approach.

Finally, alignments tend to be time-consuming. The $\Theta(n \lg n)$ bound captures the time necessary to align only a pair of sequences. While `BLAST` distributes some of the costs and allows for more efficient queries of an input sequences against a database, when the databases get large, alignment algorithms in their present form turn into a luxury few can afford.

9.3 Complementarity

Unlike alignments which suffer from a heuristic scoring function f, the dictionary provides a guarantee. All matches no shorter than k will be reported: nothing more, nothing less. We have also produced an efficient implementation of the dictionary which allows multiple sequences to be blasted against multiple sequences in the database in an amortized way.

Naturally, the dictionary approach has its faults. The fact that it's so precise could be a drawback. Biological sequences contain many insertions, deletions, etc. Even with these absent, there would still be data-entry errors: mechanical sequencing machines make mistakes. However, we let the user control for these with the threshold parameter k. As long as the mean space between errors falls above k, the dictionary will find plenty of matches; a higher entity could then piece those matches together, taking noise into account. However, the value of k could be lowered only to a point where a match of length k still remains a statistically significant event.

We see alignments and the dictionary approach as the two extremes in the spectrum of locating homologous regions. These tools could be used together to complement each other. Lior Pachter and Serafim Batzoglou created a combined engine for finding homologous regions [4]; they used the dictionary to rapidly locate reasonably long chunks of similar regions of DNA and then used alignment techniques to compare the short fringes, probing how far the similarity extended. Their algorithms are based on a detailed alignment of two orthologous genomic regions. In order to obtain good global alignments, they designed a new alignment system, which repeatedly aligns long matching segments within the genomic loci, and then refines the alignments in the intervening regions using shorter segments, giving rise to the first systematic design of a gene recognition program based on cross species comparison [4].

9.4 Exon Prediction

Many computational methods have been developed for the purposes of gene annotation [3]. Indeed, of the many applications of computer science in biology, perhaps the most successful has been the implementation of algorithms for finding similarities between sequences [29]. The most widely used program developed for this purpose is `BLAST` [1, 2], which is an alignment tool. `BLAST` is often manually applied for the purposes of gene annotation, including exon prediction and repeat finding. Other similarity search approaches include the `FLASH` [25] program which is an example of a clever use of a hash table to keep track of matches and positions of pairs of nucleotides in a database. The resulting information can be used to extract close matches to a given sequence. Nevertheless, neither of these search approaches have been designed for gene annotation.

Programs designed specifically to address the problem of exon prediction can be broadly categorized as either statistical or homology based. Statistically based programs include `GENSCAN` [6], `GENIE` [14], `GENEMARK` [16], `VEIL` [10] (all based on hidden Markov models), `FGENEH` [26] (an integration of various statistical approaches for finding splice sites, exons, etc.), and `GRAIL` [30] (based on neural networks). Homology based approaches exploit the fact that protein sequences similar to the expressed sequence of a gene are often in databases. Using such a target, one can successfully identify the coding regions of a gene. The alignment based `PROCRUSTES` [8, 28, 17] program represents a very successful implementation of this idea. When a related mammalian protein is available, this program gives 99% accurate predictions and guarantees 100% accurate predictions 37% of the time; however, the user supplies the target protein sequence. This difficulty is addressed in another recent approach [11]. Specifically, the `AAT` tool addresses this by automatically using `BLAST`-like information from protein or `dbEST` databases for exon prediction. The `INFO` program [15] is based on the idea of finding similarity to long stretches of a sequence in a protein database, and then finding splice sites around these regions. Such programs are becoming more important as the size of protein and `dbEST` databases increase.

The power of combining dictionary hits with a gene recognition program is emphasized by the following statistic (computed with the removal of exact matches of test sequences from the database): Out of the 10% of intron positions covered by matches of 8-tuples to the OWL database, only 0.5% were predicted to be in exons. Only .05% of the total intron base pairs were incorrectly classified as coding [22].

The results of tests using the entire OWL protein database for the Burset–Guigó and Haussler-Kulp-Reese data sets compare favorably with other statistical methods. Estimates for sensitivity and specificity per nucleotide position range within 60–90%. Predictions of exact exons also vary between the programs, with estimates between 30–70% specificity and sensitivity. The homology based AAT approach predicts nucleotides with a sensitivity of 94% and specificity of 97%, and exons exactly with a sensitivity of 74% and specificity of 78%.

The quality of our results is, of course, directly related to the presence or absence of related matches to our test genes in the database. The larger the minimum tuple length threshold k, the more the results become dependent on the general redundancy of the database. For genes with few matches, one can resort to a smaller tuple size and take the *number* of hits into account. Thus, our approach can be tuned to work either as a statistical method or as a homology based method, as well as the hybrids in between. Furthermore, as the size of the databases grow, the results can be expected to improve.

The dictionary approach has a number of advantages over standard similarity search techniques. Despite the unprecedented success of alignment algorithms in biology, the algorithms are all handicapped by the problem that short matching segments can be difficult to find in certain cases [5, 31]. An alignment between them may be overlooked if mismatches are penalized less than gaps. Even though this problem can be addressed by suitably modifying the alignment parameters, the number of such extreme examples, combined with the myriad of parameters necessary to address biological phenomena involved in sequence evolution (*e.g.,* BLAST [1]), creates a fundamental difficulty. Every given problem has a set of parameters associated with it that provides a “good” alignment, but there is no universal set of parameters that works for every problem. An advantage of our dictionary method is that when performing a database search, *all* the exact matches of segments in a sequence are rapidly detected. The FLASH program is also designed to find similarities of segments to sequences in databases. Unlike our dictionary method, the FLASH program relies upon storing the *positions* of shorter tuples in a hash table. The use of larger tuple sizes in a dictionary renders this unnecessary because longer tuples appear in fewer sequences. The advantages of our dictionary method becomes apparent when the databases involved become very large, which is the case with the current dbEST database. Furthermore, this method enables rapid calculation of frequency counts of arbitrary length tuples, which can be applied to statistics based programs that rely on such information.

This approach also has advantages over other exon prediction methods. The entire process of database search and exon prediction is automated, and the prediction is based on many good target sequences (and their fragments) simultaneously. Furthermore, in contrast to the INFO program, exons that have partial or no matches in a database can sometimes be accurately predicted by using the fact that exons are constrained independently in many ways: they require splice sites, must be frame consistent and cannot contain high complexity repeats [21]. Also in contrast to INFO, the method takes full advantage of the presence of long *and* short hits. The INFO program uses 26 tuples of amino acids, whereas the dictionary based approach can use all the tuple information. The AAT [11] tool is designed specifically to automate the process of finding a good target sequence. However, the reliance on one good target limits the ability of the program to predict exons from fragments.

The effectiveness of the dictionary method we propose is not at all obvious *a priori.* Indeed, the method can be inferior to alignment in the case where one wants to compare two similar DNA sequences, perhaps from different species. For instance, a few gaps and mismatches every 5–6 nucleotides will tend to exclude 11-tuple DNA hits. On the other hand, the method has proved to be very effective for exon prediction, especially when used in conjunction with a protein database (where mutations in the third position of codons do not necessarily alter the resultant protein), or a `dbEST` database (where exact matching fragments of a gene provide excellent candidates for the dictionary to find).

9.5 Related Applications

9.5.1 Repeat Masking

We have built dictionaries from repeat databases and used these for rapidly finding repeat segments in genes. Repeats are long sequences of DNA that occur very frequently in introns and very rarely in exons. This technique provides an alternative to alignment based repeat maskers such as `RepeatMasker` (A.F.A. Smit and P. Green [24]). The method is especially useful for exon prediction, where it is advantageous not only to mask complete repeats, but to mask segments (perhaps from repeats) that do not occur in exons. The dictionary approach possessed a tremendous speed advantage without noticeable defects in sensitivity [23].

9.5.2 Pseudogenes

Reverse transcribed genes which lack introns are often pitfalls for gene recognition programs. The identification of neighboring exons in inconsistent frames with no room for an intron immediately suggests the presence of a pseudogene. This can be easily checked and automated, in the same vein as the alternative splicing detection. Indeed, we discovered two such examples in a newly sequenced genomic segment (`GENBANK` Accession: AC001226) using the dictionary [23].

9.5.3 Frame Prediction

While tightly intertwined with gene recognition and exon prediction, we base frame prediction on statistics instead of homology. See the section on frame test, 10.3.

Chapter 10

Data Mining

10.1 Overview

The curse of computational biology today is the huge amount of poorly labeled (and often error-prone) data. As we hinted earlier, biological databases tend to be very polluted. First of all, more often than not, they were established by biologists and not computer scientists. Lacking a good foundation in databases, these researchers set up the knowledge storage as gargantuan text files with archaic rules. Since many people contribute information, we often find the format violated. To us, this means that simple programs invariably derail during attempts to extract sequences and related details from these files; we have had to resort to lex to achieve moderate levels of stability.

Poorly organized databases cause the least of our troubles. Sequences tend to be mislabeled as well. For instance, the National Center for Biotechnology Information maintains a wonderful database of expressed sequence tags `dbEST`. Expressed sequence tags are segments of cDNAs (a cDNA is a concatenated sequence of coding exons surrounded by other nucleotides: the surrounding DNA is non-coding). Unfortunately the ESTs are taken out of the cDNA without regard for frame, see 10.3. In addition, they are often entered into the database reversed (backwards), complemented (all `A`s replaced with `T`s and all `C`s replaced with `G`s and vice versa), and even reverse-complemented. This is an artifact of the sequencing techniques used today. Essentially, the cDNA are cut up into pieces, each piece is spun in one of four orientations, and the resulting snippets are dumped into huge, often mislabeled, databases. Nostradamus would have been proud. Interestingly enough, this lack of consistency (and annotation) is not a big problem for biologists: in a test-tube, the biological machinery itself reconstructs the complementary strand of the DNA and determines the direction in which the DNA should be read. However, this phenomenon poses a threat to scientists trying to do statistical analysis of the sequences. One could extract quite a bit of useful information if one could reliably determine the frame and the orientation of an EST. With four possible orientations and three potential frames, each sequence could be interpreted in twelve ways! If each orientation were equally likely, $\frac{11}{12}$ of the sequences used for statistical analyses would be erroneous; in practice, forward and reversed-complement orientations dominate, pushing this ratio closer to $\frac{5}{6}$.

To top it off, even the data in the databases are not clean. The sequencing machinery itself slips up every now and then; there are also data-entry errors. As a result, regions of the same EST may be in different frames. Moreover, researchers tend to discover and enter new sequences into the databases independently. Invariably, some patterns in the database become over-represented, biasing the frequency counts along with the derived statistics away from what's natural.

We have found the dictionary invaluable in mining the polluted biological databases for clean data. We devote the remainder of this chapter to a description of an attempt to orient the `dbEST` which produced enough clean data to create a high-precision frame-test. These applications are just the initial stabs to parade the power of the dictionary; other data-mining applications should be forthcoming.

10.2 Orientation

The dictionary could be used to orient a large fraction of the EST database, unveiling a wealth of statistical data. We have subscribed to the following approach: translate each of the ESTs in all three frames, in all four orientations, and use the dictionary to find matches to known proteins in OWL. We assume that a long match between an EST in a particular orientation and a known protein in OWL indicates that the orientation in question is that ESTs true orientation. This step would orient a large chunk of the ESTs. More could be oriented by finding matches between the remaining ESTs and those for which the orientation has already been nailed down. Presumably, the procedure would be repeated until no new information could be extracted.

In practice, we had to be a little more careful. For some reason, OWL contains not only proteins but also translated repeats. We first used the dictionary to mask the OWL dictionary. To do this, we constructed a dictionary out of the repeats that came with `RepeatMasker`. We then translated each of the repeats in all three frames and all four orientations to produce a database of possible repeat translations. Next, we masked all segments in OWL which matched 8 or more amino acids in a possible translation of a repeat. Sporting the masked version of OWL, we proceeded to look for matches of 10 or more amino acids between possible translations of the ESTs and the masked OWL dictionary. This produced a seed set of oriented ESTs. Unfortunately, we did not get a chance to extend the set via iterated runs.

10.3 Frame and Orientation Prediction

As we mentioned earlier, knowing the sequence of nucleotides from a coding region of a gene is not enough to predict the amino acids produced by those nucleotides, even if the introns and exons are known. The reason lies in the rudimentary fact that it takes three nucleotides to encode for a single amino acid (see figure 1.1). Thus, translations begun at nucleotide 0, 1, and 2 would yield three fundamentally different sequences of amino acids. Translations begun at nucleotide n would yield a tail of the sequence of amino acids whose translation began at nucleotide $n \bmod 3$.

An interesting theoretical and practical question is whether or not, by looking at a sequence of nucleotides, one could make a confident prediction of whether or not that sequence of nucleotides is "in-frame." By "in-frame," we mean that it would be translated into amino acids by the cell starting at nucleotide 0 and not nucleotide 1 or 2. If so, it would be useful to know the accuracy of some such tests as a function of the length of the sequence of nucleotides.

These are loaded questions: several assumptions should be made apparent. First of all, we assume that the DNA sequences presented as the input came from a coding region (a concatenation of exons) and not from introns or other places in the gene. Second, we should note that there is a phenomenon called "alternate splicing:" sometimes the cell itself parses the gene differently at different times, so every now and then one finds a gene which encodes for valid proteins in more than one frame. This phenomenon is extremely rare and we will assume that a sequence of coding DNA has precisely one true frame.

10.3.1 Motivation

Much more often than not, coding genetic material has a definite frame. Translated in one frame, it has a deep meaning: an encoding for a protein which will go on to perform a function in the organism. Translated in the other two frames, the meaning is not as pronounced. As a result, one expects to find much less structure in the out-of-frame translations. This effect is enhanced by the fact that the third nucleotide in the DNA triplet often does not affect the amino acid encoded (see figure 1.1). DNA to amino acid conversion has apparent built-in redundancies: the number of possible encodings ($4^3 = 64$) is larger than the number of possible amino acids (20). It follows then that out-of-frame translations should be more random than the frame-consistent translation of a sequence of DNA nucleotides. Therefore, there is hope that the information about the frame is contained within the DNA sequence itself.

The notion of a frame-test first appeared in [20]. A good frame-prediction tool could prove to be highly useful in an algorithm for gene annotation. It could be used as an overall sanity check to make sure that the predicted concatenation of exons looks like it is in-frame. More importantly, the tool could be used to help locate the boundaries of exons. One could shrink / expand predicted exons to make sure that neighboring exons combine to produce something that looks like it is frame-consistent.

Choosing the right orientation (forward, forward-complement, reverse, or reversed-complement) constitutes a problem very similar to that of frame-detection (except now three of the choices do not have a pronounced meaning, and still only one does). Techniques developed for frame-detection could be used for orientation-detection. Armed with both a reliable orientation-test and a reliable frame-test, one may be able to recover a lot more information from `dbEST`, possibly bringing order to the entire database.

10.3.2 Linear Techniques

Together with Mark Knobel, the author tackled the challenge of creating a good frame-test as a final project for Gert Cauwenberghs's Computational Laboratory in Cognitive Science (9.39) class in Spring 1999; in addition, the author worked on the same problem as a final project for Whitney Newey's Econometrics (14.31) class also in Spring 1999. The remainder of this subsection consists of excerpts from the final project reports.

Data

As often is the case in computational biology, gathering clean data is not a trivial task. We need examples of coding genetic material in-frame. Out-of-frame examples can be easily obtained by chopping off the first one and the first two nucleotides in the in-frame examples.

In a prior work, we attempted to orient the `dbEST` using the following procedure. Each EST was translated in each of the four possible directions and each of the three possible frames, yielding ($4 \times 3 =$) 12) amino acid sequences per EST. This was a rather large project, seeing as how `dbEST` consists of a little over 2,000,000 ESTs, and growing. We then compared all of the resulting amino acid sequences with the OWL protein database. The OWL protein sequence database is a composite, non-redundant database assembled from a number of primary sources and contains a little over 250,000 protein sequences all of which are "sufficiently" different from each other [19]. When we found "long" matches between a protein in OWL and a translation of an EST from `dbEST`, we labeled that EST with the direction and the frame of the corresponding translation.

In this work, to obtain an unbiased set of in-frame coding DNA sequences, we found for each protein in OWL the longest match in `dbEST`. Naturally, some proteins in OWL did not match anything in `dbEST` and some had very short longest hits. Still, assuming that OWL is non-redundant, as it claims to be, we had on our hands a well-represented group of in-frame coding DNA. We then eliminated all the short sequences of DNA because short matches between `dbEST` and OWL could be accidental. It seems that 40 amino acids (120 nucleotides) was close to the cutoff point where the number of OWL to `dbEST` hits switches from exponential to linear (see figure 10-1). Of the roughly 150,000 matches, about 50,000 survive the cutoff. We split this dataset into a training set and a testing set randomly: each of the 50,000 sequences was assigned either to the training set or to the testing set via a fair coin toss. We characterize below the resulting data sets (see figures 10.1 and 10-2). We took every effort to make sure that the training set and the testing set would be fairly selected. It should be clear that they have the same basic characteristics.

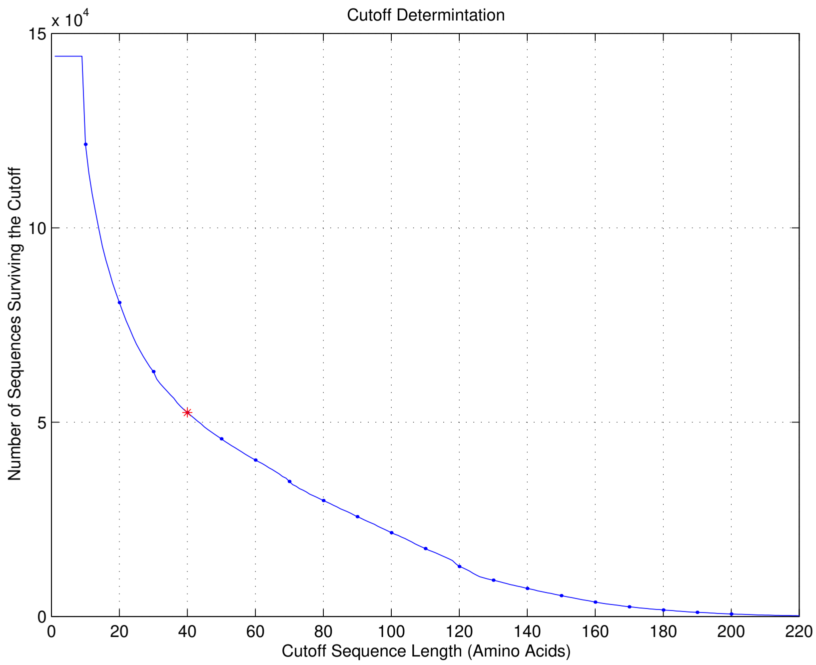


Figure 10-1: Extracting Clean In-Frame Coding DNA Data.

Units of Length: Nucleotides	Number of Sequences	Minimum Length	Maximum Length	Total Length	Average Length	Standard Deviation of the Length
Training Set	26,461	120	2463	7,509,474	283.8	122.0
Testing Set	26,777	120	1185	7,572,066	282.8	120.1

Table 10.1: Data Sets' Characteristics.

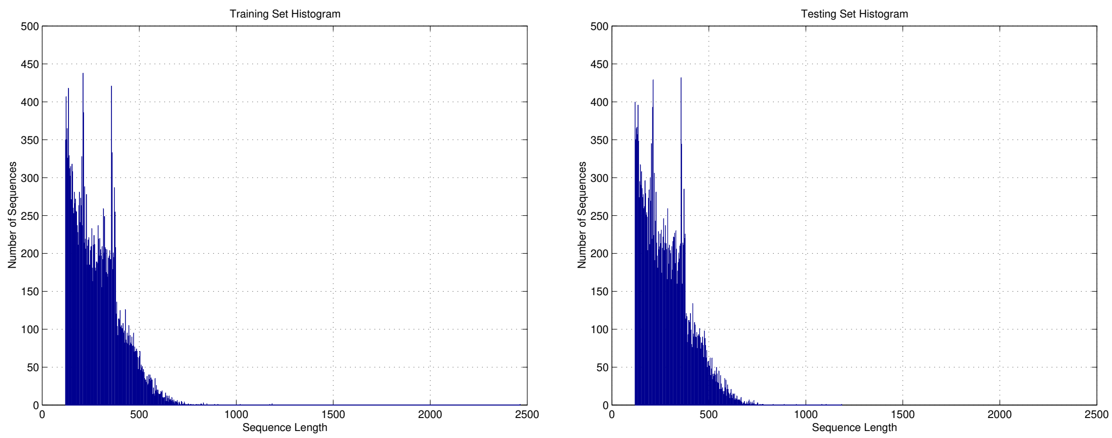


Figure 10-2: Data Sets' Histograms.

Results

Just to clarify what it is we are talking about here: a frame-predictor takes as an input a string of nucleotides $s \in \{\mathtt{A},\mathtt{C},\mathtt{G},\mathtt{T}\}^+$ and outputs a result $r \in \{\pm 1\}$. If $r = 1$, then s is predicted to be in-frame. If $r = -1$, then s is predicted to be out-of-frame.

We assume that s came from a complete coding sequence of DNA c, where

$$c = \texttt{ATG}\{\texttt{A,C,G,T}\}^{3k}\{\texttt{TAA,TAG,TGA}\}, k \in \mathcal{N},$$

subject to the constraint that all the internal codons are coding. Then c can be rewritten as a concatenation of three strings $c = usv$. The string s is in-frame if $|u| = 3k$ for some $k \in \mathcal{N}$.

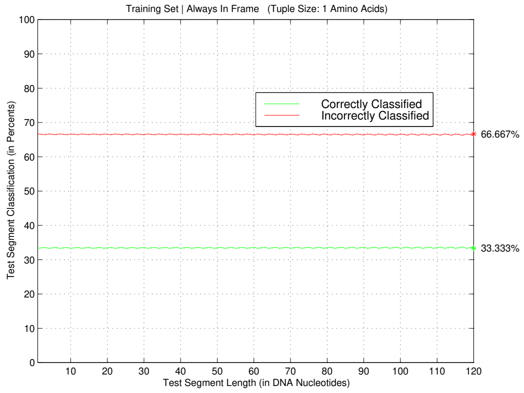
Evaluation Scheme

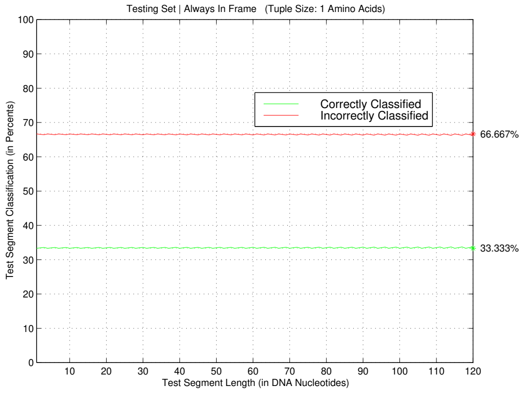
Given a predictor $\mathcal{P}(\cdot)$, we test it on each sequence in our data sets. Initially, we extract three tests from every sequence s. Let s' be s without the first nucleotide and s'' be s without the first two nucleotides. The three tests are $\mathcal{P}(s) \stackrel{?}{=} {}^{+}1$, $\mathcal{P}(s') \stackrel{?}{=} {}^{-}1$, and $\mathcal{P}(s'') \stackrel{?}{=} {}^{-}1$.

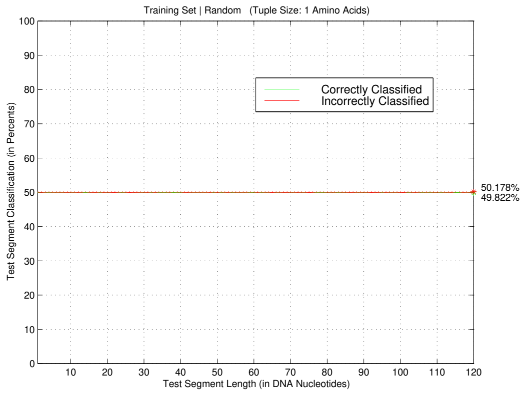
All of the sequences in our data sets are longer than 120 nucleotides. Therefore, $\mathcal{P}(\cdot)$'s performance on entire sequences is the limit of its performance as $|s|$ becomes very large. In order to understand how good of a predictor we have, it is important to understand the dynamics of its performance as $|s|$ varies in its low ranges. Therefore, in addition to the three full-sequence tests, we also test $\mathcal{P}(\cdot)$ on every subsequence z of each sequence, where $1 \leq |z| \leq 120$.

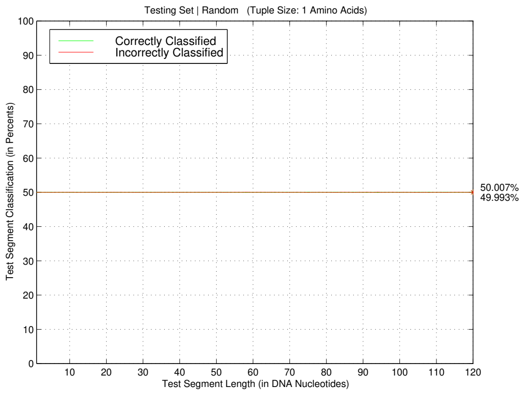
Some Basic Attempts

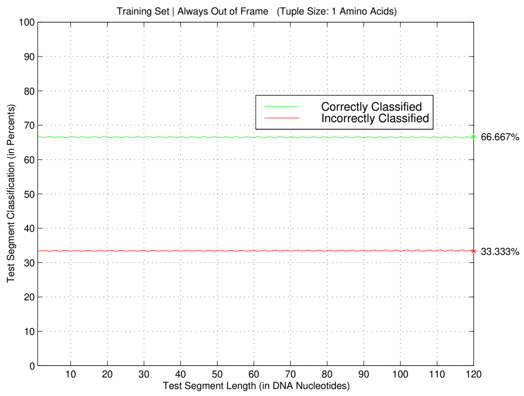
First of all, since the problem is discrete, we begin by examining the obvious predictors $\mathcal{P}^Y(\cdot)$ (Y for "yes") which always returns ${}^{+}1$, $\mathcal{P}^N(\cdot)$ (N for "no") which always returns ${}^{-}1$, and $\mathcal{P}^R(\cdot)$ (R for "random") which returns ${}^{+}1$ or ${}^{-}1$, each with probability $\frac{1}{2}$. None of these predictors actually look at their input sequence. Since there are three frames, $\frac{1}{3}$ of the test subsequences are in-frame and the other $\frac{2}{3}$ are out-of-frame. Therefore, we expect $\mathcal{P}^Y(s)$ to be correct $\frac{1}{3}$ of the time, $\mathcal{P}^N(s)$ to be correct $\frac{2}{3}$ of the time, and $\mathcal{P}^R(s)$ to be correct $\frac{1}{2}$ of the time, regardless of $|s|$. To familiarize the reader with the output format of our evaluator, we present the obvious graphs for these trivial predictors. Percentages of frames correctly predicted as a function of input length are in green; percentages of frames incorrectly predicted are in red. At the very right, percentage of frames predicted correctly among the full-length sequences is marked with a green asterisk; percentage of frames predicted incorrectly is marked with a red asterisk.

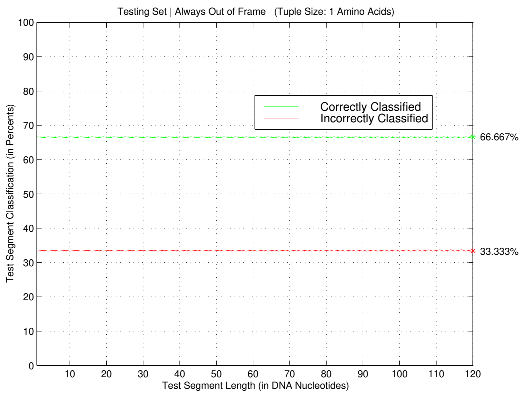




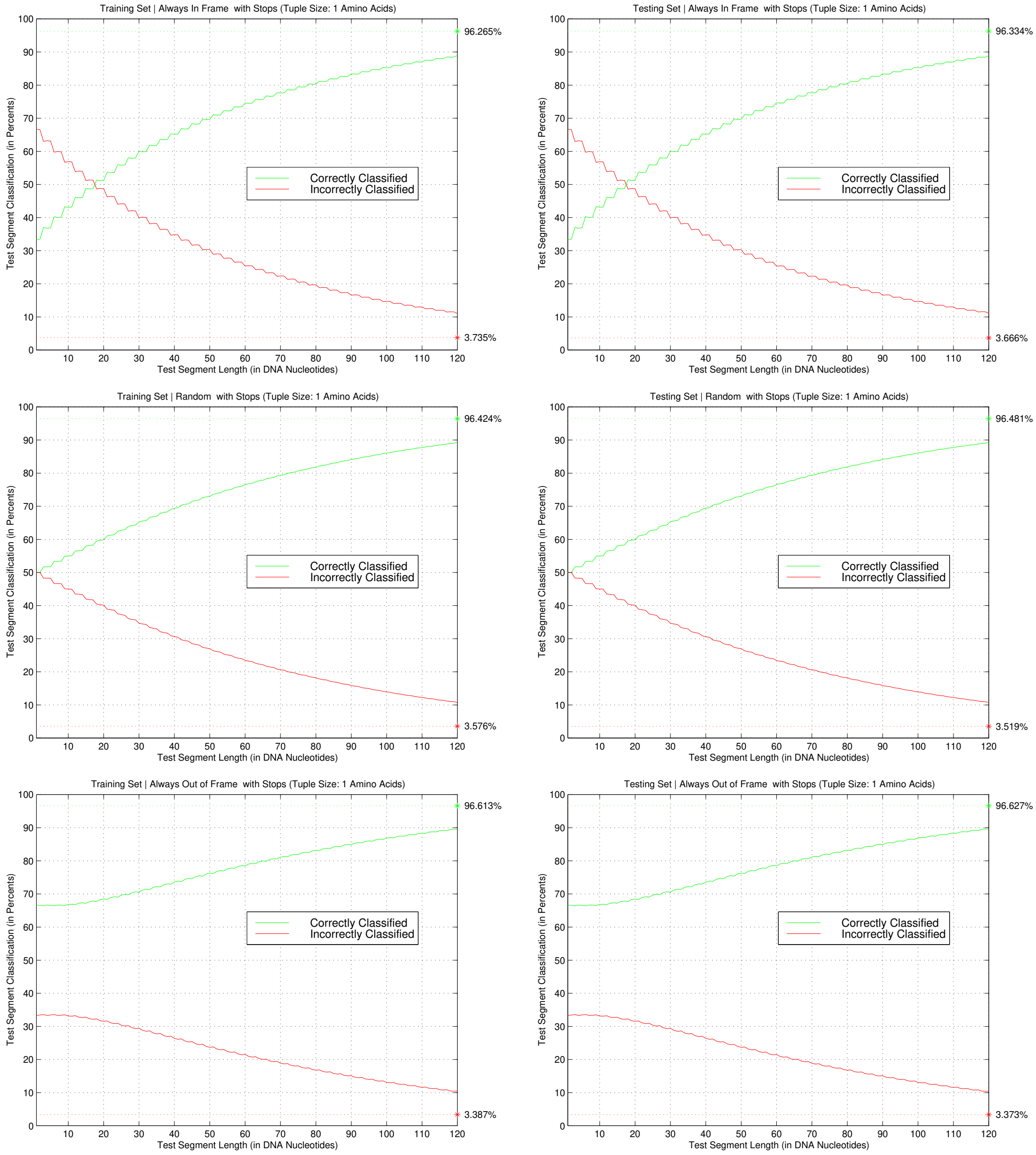








We can take advantage of a very simple fact to improve the performance of our trivial predictors. Coding sequences of DNA never contain a stop codon. Therefore, a sequence cannot be in-frame if it contains a stop-codon in frame zero; a sequence cannot be out-of-frame if it contains a stop-codon in frame one and in frame two. This basic enhancement to our trivial predictors results in better performance.



These performance graphs would not be complete without a specification of the number of test-cases at each segment length. Clearly, there are many more single-nucleotide subsequences than full-length sequences. The table on the next page tabulates these statistics for both data sets. The number of test-cases for segments of length 120 and under is always well in the millions, so we deem the performance charts to be rather reliable.

Segment Length	Total in Training Set	Total in Testing Set	Segment Length	Total in Training Set	Total in Testing Set
1	7509618	7572333	61	5921898	5965653
2	7483156	7545555	62	5895436	5938875
3	7456694	7518777	63	5868974	5912097
4	7430232	7491999	64	5842512	5885319
5	7403770	7465221	65	5816050	5858541
6	7377308	7438443	66	5789588	5831763
7	7350846	7411665	67	5763126	5804985
8	7324384	7384887	68	5736664	5778207
9	7297922	7358109	69	5710202	5751429
10	7271460	7331331	70	5683740	5724651
11	7244998	7304553	71	5657278	5697873
12	7218536	7277775	72	5630816	5671095
13	7192074	7250997	73	5604354	5644317
14	7165612	7224219	74	5577892	5617539
15	7139150	7197441	75	5551430	5590761
16	7112688	7170663	76	5524968	5563983
17	7086226	7143885	77	5498506	5537205
18	7059764	7117107	78	5472044	5510427
19	7033302	7090329	79	5445582	5483649
20	7006840	7063551	80	5419120	5456871
21	6980378	7036773	81	5392658	5430093
22	6953916	7009995	82	5366196	5403315
23	6927454	6983217	83	5339734	5376537
24	6900992	6956439	84	5313272	5349759
25	6874530	6929661	85	5286810	5322981
26	6848068	6902883	86	5260348	5296203
27	6821606	6876105	87	5233886	5269425
28	6795144	6849327	88	5207424	5242647
29	6768682	6822549	89	5180962	5215869
30	6742220	6795771	90	5154500	5189091
31	6715758	6768993	91	5128038	5162313
32	6689296	6742215	92	5101576	5135535
33	6662834	6715437	93	5075114	5108757
34	6636372	6688659	94	5048652	5081979
35	6609910	6661881	95	5022190	5055201
36	6583448	6635103	96	4995728	5028423
37	6556986	6608325	97	4969266	5001645
38	6530524	6581547	98	4942804	4974867
39	6504062	6554769	99	4916342	4948089
40	6477600	6527991	100	4889880	4921311
41	6451138	6501213	101	4863418	4894533
42	6424676	6474435	102	4836956	4867755
43	6398214	6447657	103	4810494	4840977
44	6371752	6420879	104	4784032	4814199
45	6345290	6394101	105	4757570	4787421
46	6318828	6367323	106	4731108	4760643
47	6292366	6340545	107	4704646	4733865
48	6265904	6313767	108	4678184	4707087
49	6239442	6286989	109	4651722	4680309
50	6212980	6260211	110	4625260	4653531
51	6186518	6233433	111	4598798	4626753
52	6160056	6206655	112	4572336	4599975
53	6133594	6179877	113	4545874	4573197
54	6107132	6153099	114	4519412	4546419
55	6080670	6126321	115	4492950	4519641
56	6054208	6099543	116	4466488	4492863
57	6027746	6072765	117	4440026	4466085
58	6001284	6045987	118	4413564	4439307
59	5974822	6019209	119	4387102	4412529
60	5948360	5992431	120	4360640	4385751
$\|s\|$	79386	80334			

Table 10.2: Number of Test-Cases per Segment Length

The best of the trivial predictors, $\mathcal{P}^N$ with stops, achieves the following levels of correctness:

Correctness Level	80%	90%	95%	96.6%
Sequence Length	65	120	?	$\|s\|$

Table 10.3: Performance for the Trivial Predictors

A good frame predictor would have to do much better than this one, since it extracts only a minimal amount of information from the input sequence s.

Least Squares Classification

Translated in- and out-of-frame, coding DNA yields drastically different-looking sequences of amino acids. By gathering all of the sequences s, we can obtain examples of amino acids resulting from in-frame (frame zero), frame one, and frame two: translations of s, s', and s''. We can reduce each such translated sequence to only a handful of parameters: f_i in the following way.

Consider a sequence z. Let $f_0^z = 1$, a constant and let n be the number of amino acids arising from translating z in-frame (disregard the stop codons). Now let f_i^z be the number of amino acids i arising from that translation divided by n. So f_i^z, $1 \leq i \leq 20$, is the frequency of occurrence of the ith amino acid. Any sequence of nucleotides z can be parameterized in this way. Given this parameterization, we can create three tests t^0, t^1, and t^2, with test t^i recognizing a sequence of amino acids arising from translation of nucleotides in frame i. We hope to get a positive estimate

$$\hat{t}^i = \sum_{j=0}^{20} \hat{\beta}_j f_j^z$$

if sequence z is indeed in frame i and a negative estimate otherwise. For t^0, all sequences s are positive examples $^+1$ and all sequences s' and s'' are negative examples $^-1$. For t^1, all sequences s' are positive examples $^+1$ and all sequences s and s'' are negative examples $^-1$. For t^2, all sequences s'' are positive examples $^+1$ and all sequences s and s' are negative examples $^-1$. We regressed t^0, t^1, and t^2 on f_i, $0 \leq i \leq 20$ using the ordinary least squares (OLS) approach:

$$\overline{\hat{\beta}^i} = (F^T F)^{-1} \overline{T^i}.$$

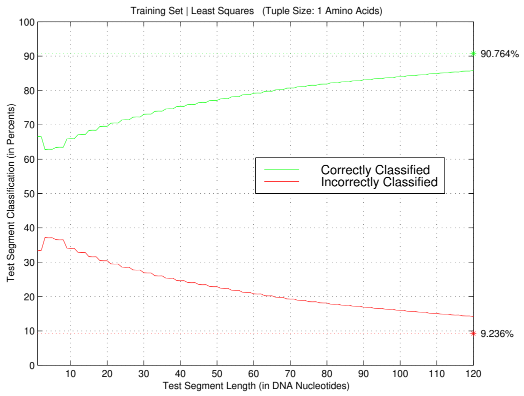
Below is a table of the coefficient estimates β_j^i, $0 \leq i < 3$, $0 \leq j \leq 20$:

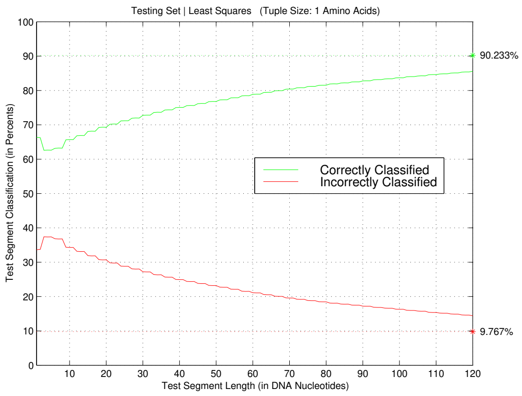
	$\hat{\beta}^0$	$\hat{\beta}^1$	$\hat{\beta}^2$
$\hat{\beta}_0$	1.128087	−3.095185	0.824240
$\hat{\beta}_1$	1.576952	1.450975	−2.885071
$\hat{\beta}_2$	−5.545160	1.461566	4.226452
$\hat{\beta}_3$	7.205222	−3.534780	−3.527584
$\hat{\beta}_4$	1.887252	−1.392461	−0.351932
$\hat{\beta}_5$	0.213390	0.697439	−0.767971
$\hat{\beta}_6$	−1.108929	−0.732509	1.984296
$\hat{\beta}_7$	−6.737540	0.357873	6.522525
$\hat{\beta}_8$	−0.358790	1.864820	−1.363174
$\hat{\beta}_9$	0.078612	3.853212	−3.788964
$\hat{\beta}_{10}$	−2.792999	6.056546	−3.120689
$\hat{\beta}_{11}$	−3.571861	13.458493	−9.743774
$\hat{\beta}_{12}$	0.458385	1.127424	−1.442951
$\hat{\beta}_{13}$	−1.854856	3.929178	−1.931467
$\hat{\beta}_{14}$	−3.787151	1.822753	2.107255
$\hat{\beta}_{15}$	−5.037488	3.679261	1.501082
$\hat{\beta}_{16}$	−2.897283	4.460794	−1.420655
$\hat{\beta}_{17}$	−0.866147	6.558592	−5.549590
$\hat{\beta}_{18}$	1.299381	2.799014	−3.955538
$\hat{\beta}_{19}$	−6.376928	5.960461	0.559323
$\hat{\beta}_{20}$	2.756001	−4.530375	1.917231

Table 10.4: Ordinary Least Squares Coefficient Estimates

We do not list the standard errors for our estimates for several reasons. First of all, as will be discussed later, OLS in the case of a discrete dependent variable inherently suffers from heteroskedasticity. Parameter estimates are still consistent, but the standard errors are wrong. We do not list the corrected standard errors either however. The purpose of standard errors is to indicate the validity of a regression. We will test that validity directly by examining our predictors' performance on both the training set and the testing set.

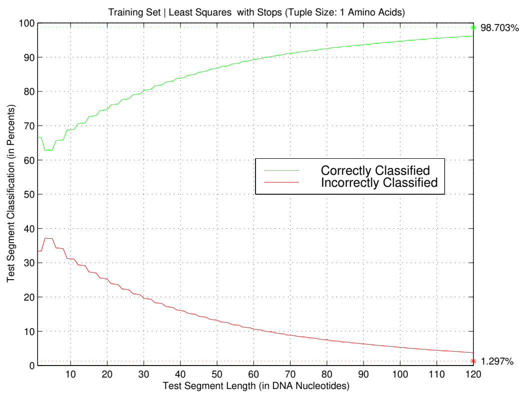
There is a straight-forward way to use the coefficients $\overline{\beta^i}$ in a frame-predictor. Given a sequence z, compute its parameters f_i^z. Use the parameter vector $\overline{f^z}$ to estimate the frame-tests t^i: $\hat{t}^i = \overline{\beta^i} \cdot \overline{f^z}$. Predict z to be in-frame if and only if $\hat{t}^0 > \max\{\hat{t}^1, \hat{t}^2\}$. Below are the performance charts for this predictor.

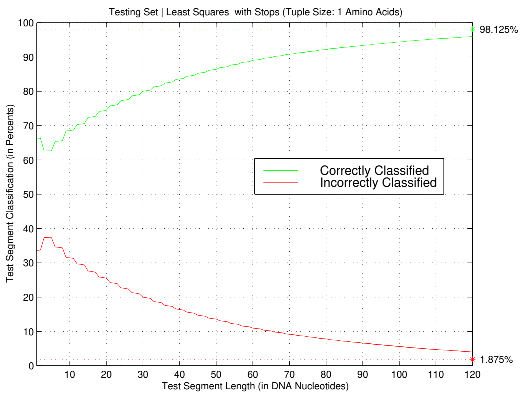




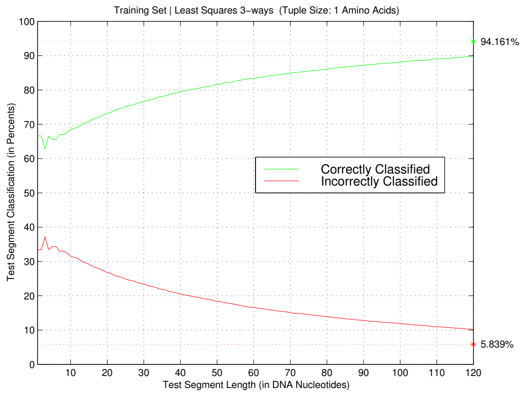
Enhancements

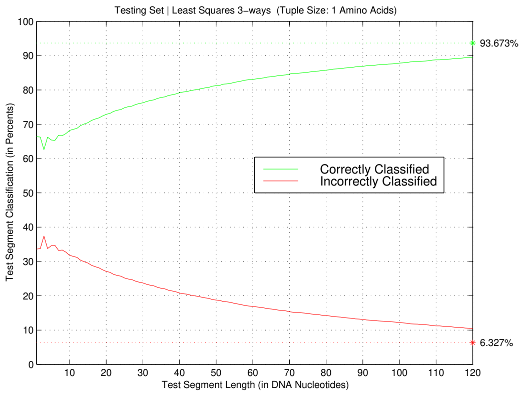
Naturally, we can enhance this test with the stop-codon information too. One simple way to do this is to override $\hat{t}^{(2-i \bmod 3)} \leftarrow -\infty$ if z contains a stop-codon in frame i. Not surprisingly, the enhanced predictor performs even better:

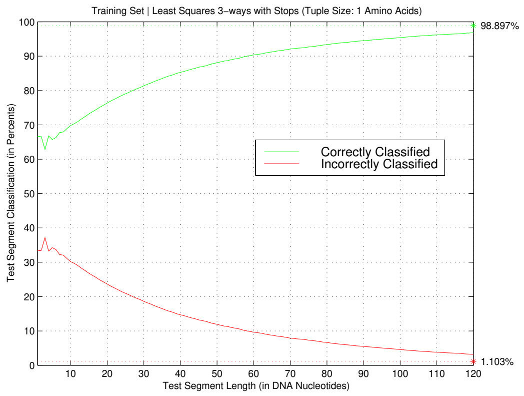


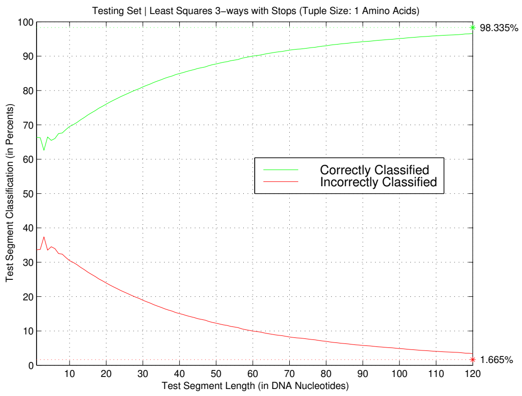


The fun does not stop here. There is another way to enhance our predictor. We can obtain three estimates of $\hat{t}^i$ for each i by scoring z translated in all three frames: by scoring z, z', and z''. The three estimates of $\hat{t}^0$ are $\hat{t}^0_z$, $\hat{t}^1_{z'}$, and $\hat{t}^2_{z''}$. The three estimates of $\hat{t}^1$ are $\hat{t}^1_z$, $\hat{t}^2_{z'}$, and $\hat{t}^0_{z''}$. The three estimates of $\hat{t}^2$ are $\hat{t}^2_z$, $\hat{t}^0_{z'}$, and $\hat{t}^1_{z''}$. We make three predictions about the frame of z based on the $\hat{t}^i$'s for z, z', and z''. We declare z to be in-frame if and only if the majority of the predictions are in-frame. Of course, this new enhancement can be used together with the stop-codon information. Below are the performance charts for the rest of the combinations.









The best of these predictors $\mathcal{P}^{OLS}(s)$ considers the stop-codon information in s and the amino-acid translations of s, s', and s'':

Correctness Level	80%	90%	95%	98.3%
Input Length	30	60	95	$\lvert s \rvert$

Table 10.5: Performance for the OLS Predictors

Longer Tuples

To further improve our predictor, we could consider not just the frequencies of single amino acids (1-tuples) but longer segments of amino acids (t-tuples). When $t = 2$, we would have to estimate β_0^i, the constant term, and β_{jk}^i for all the pairs of amino acids jk, a total of $1 + 20^2 = 401$ parameters for each of the three tests. Extracting $3 \times 401 = 1203$ parameters from the testing set of our size is reasonable. There are too many such parameters to present here conveniently. Instead, we show the correlation matrix $F^T F$ pictorially. High correlations are depicted in red (dark-grey); low correlations are depicted in blue (light-grey).

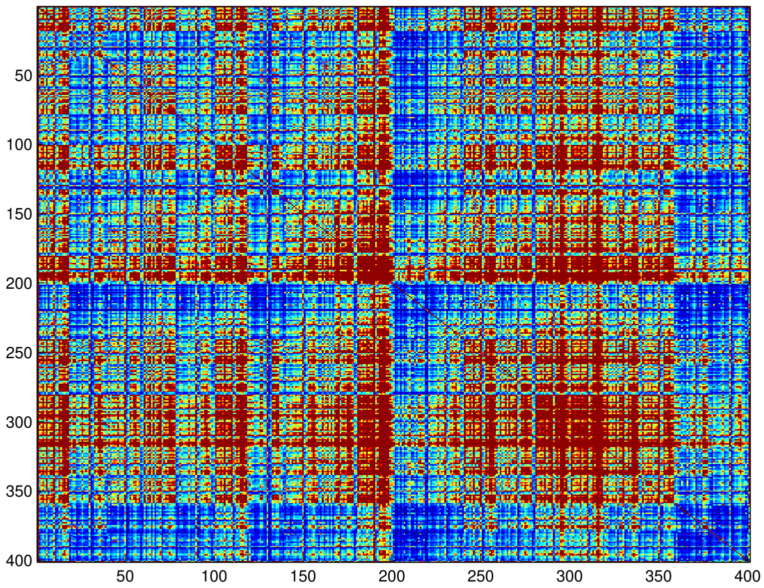
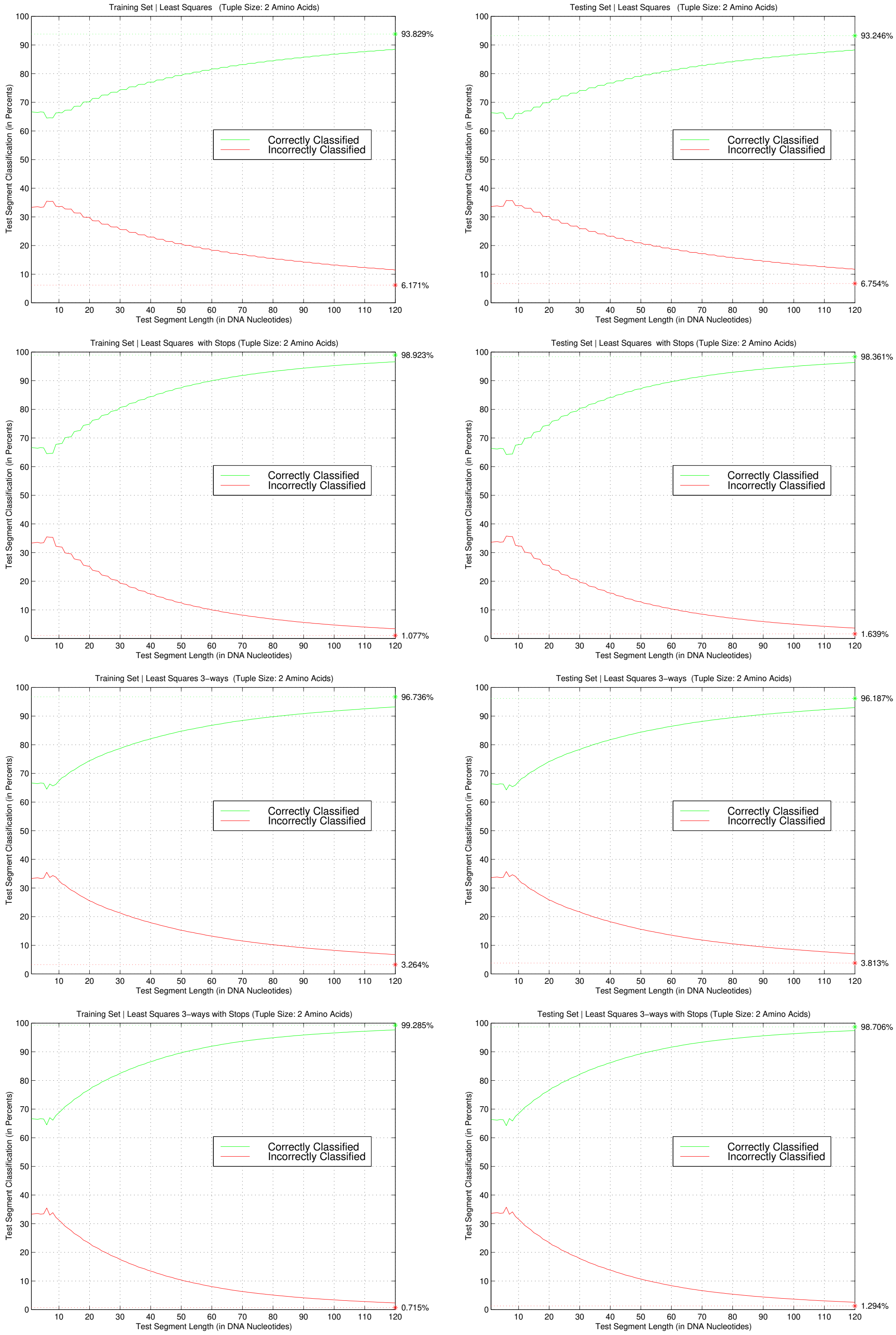


Figure 10-3: Correlation Matrix

Having run the appropriate regressions, we measure the performance of our predictors using the new parameterization.



Again, the best of these predictors $\mathcal{P}^{OLS}(s)$ considers the stop-codon information in s and the amino-acid translations of s, s', and s'':

Correctness Level	80%	90%	95%	98.7%
Input Length	25	50	85	$\|s\|$

Table 10.6: Performance for the Extended OLS Predictors

Practical Problems with OLS

Theoretically, we should be able to parameterize a sequence z efficiently using even longer tuples. Since there are twenty amino acids, there are 20^t possible t-tuples. For our three tests, we would need to extract

$$3 \times \left(1 + 20^t\right)$$

parameters from the training set. The training set contains on the order of seven million nucleotides and therefore about that many t-tuples (for $t \ll 120 < |s|$). Our parameters will certainly not be efficient when their number becomes on the order of millions. We tabulate the number of parameters we would be required to estimate for the first few values of t below.

t	$\|\overline{f}\|$
1	63
2	1203
3	24,003
4	480,003
5	9,600,003

Theoretically, we should be able to parameterize a sequence z by its 4-tuples without a problem. Practically, this is very difficult. The size of the correlation matrix $(F^T F)$ becomes rather large. For 3-tuples, it is $8,001 \times 8,001 = 64,016,001$. For 4-tuples, it is $160,001 \times 160,001 = 25,600,320,001$. Inverting (or even storing) matrices of this size is still computationally expensive. Although the 3-tuple matrix actually fits in the 300Mb memory of our computer, MATLAB runs out of memory when trying to invert it. When correlation matrices get this large, even the fact they are symmetric (which halves the memory requirement) no longer helps.

Theoretical Problems with OLS

The other major problem with OLS is that when the dependent variable is discrete, heteroskedasticity is inherent in the problem. That is, the error terms from the regression are not exogenous.

To illustrate, consider the simple case $y_i \in \{0, 1\}$. By the linear model,

$$E(y_i|x_i) = \beta_1 + \beta_2 x_i.$$

For the zero-one random variable,

$$E(y_i|x_i) = 0 \times P(y_i = 0|x_i) + 1 \times P(y_i = 1|x_i) = P(y_i = 1|x_i).$$

Thus,

$$P(y_i = 1|x_i) = \beta_1 + \beta_2 x_i.$$

First of all, clearly we have built-in heteroskedasticity. And second, there is no guarantee that $\beta_1 + \beta_2 x_i \in [0, 1]$, as a probability should. In a future work, it may be interesting to use a model like $P(y_i|x_i) = \Phi(\beta_1 + \beta_2 x_i)$, where

$$\Phi(u) = \int_{-\infty}^{u} \frac{1}{\sqrt{2\pi}} e^{-\frac{t^2}{2}} dt,$$

the standard normal distribution; it would also be interesting to try the logistics curve $P(y_i|x_i) = L(\beta_1 + \beta_2 x_i)$, where

$$L(u) = \frac{e^u}{1 + e^u}.$$

If we use a model $P(y_i|x_i) = G(\beta_1 + \beta_2 x_i)$, according to maximum likelihood estimation, we would need to find the β_j's that minimize

$$\sum_i y_i \ln\left(G(\beta_1 + \beta_2 x_i) + (1 - y_i) \ln\left(1 - G(\beta_1 + \beta_2 x_i)\right)\right).$$

Unfortunately this minimization problem does not have a closed form solution. It is, however, easy to solve using numerical methods. This may be an interesting approach for a future work.

Direct Frequency Analysis

To avoid the theoretical complications (and the hardware requirements) of generalizing OLS, we attempt to find good coefficients $\overline{\beta}$ for our parameters using brute force. We consider two extremely simple heuristics for this task.

First of all, we ignore the correlations between the frequencies of tuples of amino acids. This allows us to avoid having to deal with large correlation matrices. Second, we ignore sequence boundaries and treat all of our sequences as one large concatenated sequence. This masks the problem of biased representation of tuples. Even at the 2-tuple level, we have 400 tuples. Most sequences in our training set do not contain even half that many tuples.

We translate all of our sequences s in all three frames. We then compute the frequencies of each tuple among the in-frame translations, among translations in frame two, and among translations in frame three. In addition, we compute the overall frequency of each tuple among all of the frames. Thus, for each tuple t, we know its frequency in-frame r_t^0, in frame one r_t^1, in frame two r_t^2 and its overall frequency r_t. We then consider two schemes for estimating β_t^i.

Extremum

The "Extremum" scheme works as follows: observe that r_t^0, r_t^1, and r_t^2 cannot all be above or below r_t at the same time. One r_t^i has to be on one side of r_t and the other two must be on the other side, with the exception of the trivial case (which does not really matter) of $r_t^0 = r_t^1 = r_t^2 = r_t$. For each tuple t, we find the lone r_t^i (one which is on the opposite side of r_t from $r_t^j, j \neq i$). We set $\hat{\beta}_t$'s as follows: $\hat{\beta}_t^j \equiv 0$. If $r_t^i > r_t$, $\hat{\beta}_t^i \equiv {}^+1$. If $r_t^i < r_t$, $\hat{\beta}_t^i \equiv {}^-1$. All three constant term estimators $\hat{\beta}_0 \equiv 0$.

Side-to-Side

The "Side-to-Side" scheme works in a way that is very similar to the "Extremum" scheme. All three constant term estimators $\hat{\beta}_0 \equiv 0$, as before. However, instead of finding the lone r_t^i, we assign a unit weight to all parameter coefficient estimators as above. If $r_t^i > r_t$, $\hat{\beta}_t^i \equiv {}^+1, \forall i$. If $r_t^i < r_t$, $\hat{\beta}_t^i \equiv {}^-1, \forall i$.

Performance

Again, the best of these predictors $\mathcal{P}^{EXT}(s)$ and $\mathcal{P}^{STS}(s)$ considers the stop-codon information in s and the amino-acid translations of s, s', and s'':

Prediction Scheme And Tuple Resolution	Input Length Achieving 80% Correctness	Input Length Achieving 90% Correctness	Input Length Achieving 95% Correctness	Correctness Achieved for Complete Sequences
$\mathcal{P}_1^{EXT}$	25	60	105	99.0%
$\mathcal{P}_1^{STS}$	25	60	105	99.0%
$\mathcal{P}_2^{EXT}$	23	53	90	99.1%
$\mathcal{P}_2^{STS}$	23	53	90	99.1%
$\mathcal{P}_3^{EXT}$	22	50	80	99.1%
$\mathcal{P}_3^{STS}$	22	50	80	99.1%
$\mathcal{P}_4^{EXT}$	18	43	70	99.3%
$\mathcal{P}_4^{EXT}$	18	43	70	99.3%

Table 10.7: Performance for the Direct Frequency Predictors

These schemes for estimating the parameter coefficient seem to be very robust. They perform about equally well both compared to each other and compared to the best OLS predictors, at the appropriate tuple resolutions. Unlike the OLS coefficients, which require matrix inversions and $O(|\overline{f}|^2)$ memory space, our heuristic schemes are fast and easy to compute, and require only $O(|\overline{f}|)$ space in the process. We do not present the performance charts to conserve space. They are available in the original reports to the appropriate classes.

Conclusions

We started out with the traditional linear regression model for determining whether or not a sequence of coding nucleotides is "in-frame," using tuple frequencies as parameters. We achieved very good performance using the ordinary least squares approach to linear regressions. We were able to push the tuple resolution to its limit of four amino acids (for our training set) by estimating regression coefficients using two very simple heuristics. These schemes achieve excellent performance levels. We have characterized that performance as a function of the length of the input string. Our methods are flexible. They work given a sequence in terms of its nucleotides or in terms of a subset of its amino-acid translation. In some applications, the DNA itself may not be available. Perhaps one has access only to its frame two translation. Our predictors still do quite well. Their performance has been characterized for all combinations of the enhancements' uses. Some future research possibilities include using modified OLS for the regressions, testing alternative heuristics, and attempting the heuristics found to solve other problems (such as orientation prediction).

10.3.3 Dot Product

With the classes out of the way, the author implemented the numerical algorithms to make Probit and Logit regression models work. In practice, solving the minimization problem took rather large amounts of time. In addition, the coefficients obtained did not perform noticeably better than the plain vanilla OLS.

Disappointed by regression metrics and inspired by the performance, simplicity, and ease of implementation of the direct frequency methods, the author went on to explore the space. We have found a simple and elegant approach which is actually founded in theory, as it is related to the notion of a matched filter from signal processing.

If the number of possible tuples is n, let vectors $\bar{v}^i$ contain the frequencies of each of these tuples in frame i. The tuples may be defined over amino acids or nucleotides. Let the vectors $\bar{f}^i$ contain the frequencies of each of the same tuples in frame i in the input sequence. Then define a score vector $\bar{s}$ of length three containing the scores for each of the potential frames. We find each of the scores s_i by taking the dot product of the frequencies of the input sequence with the expected frequencies consistent with frame i. We expect the most "ringing" from the true frame since it came from a similar distribution of tuples:

$$s_0 = \bar{f}^0\bar{v}^0 + \bar{f}^1\bar{v}^1 + \bar{f}^2\bar{v}^2,\ s_1 = \bar{f}^0\bar{v}^1 + \bar{f}^1\bar{v}^2 + \bar{f}^2\bar{v}^0,\ s_2 = \bar{f}^0\bar{v}^2 + \bar{f}^1\bar{v}^0 + \bar{f}^2\bar{v}^1.$$

In the mean time, we also discovered (using the dictionary) that the OWL database was not all that non-redundant. Indeed, the training and testing sets had significant overlaps. While this did not have a major impact on the results from the previous section, we took some action, managing to avoid having to throw out sequences.

Using the representative benchmark gene data set [9] we obtained a collection of exons. These are supposed to be representative, non-redundant, and good for cross-study benchmark comparisons, etc. Merging the old training and testing set back into one, we removed from it all sequences producing an exact match of 10 or more amino acids against the dictionary of proteins produced by the benchmark exons. The resulting training set contained sequences of DNA — obtained from prior orientation attempts and assumed to be regions of concatenated exons, none shorter than 120 nucleotides — none of which code for a protein which shares even ten consecutive amino acids with the proteins from the benchmark set.

First of all, this is a bit of an overkill since in our experience, stray matches of even twenty amino acids are not so rare. However, we wanted to make sure that the testing set is indeed disjoint from the training set, as much as is practically possible. Second, the resulting training set is not non-redundant, or no less non-redundant than OWL. It is however clean, in that we are rather confident that the nucleotides listed produced a protein segment.

We have applied this method to compute "OLS" coefficients for a frame-test and an orientation-test. We drew the statistical information from our training set; results of testing them on the disjoint benchmark set are below. We considered every subsequence of every sequence in the testing set to characterize the dynamics of the performance characteristics. In the case of orientation tests, we also considered every subsequence in each of the four orientations. The stops option in the graphs that follow simply weeds out some more or less obvious options: a frame is ruled out if it contains a stop codon; an orientation is ruled out if it contains stop codons in all three frames.

10.4 Conclusion

The dictionary's raw power and speed already proved it as an excellent tools for extracting clean data out of the otherwise polluted and poorly annotated biological databases. We mined the nucleotide sequences corresponding to actual protein fragments. These made it possible to easily create very simple yet effective orientation- and frame-tests. It would be interesting to apply the dictionary to other data-mining tasks in the future.

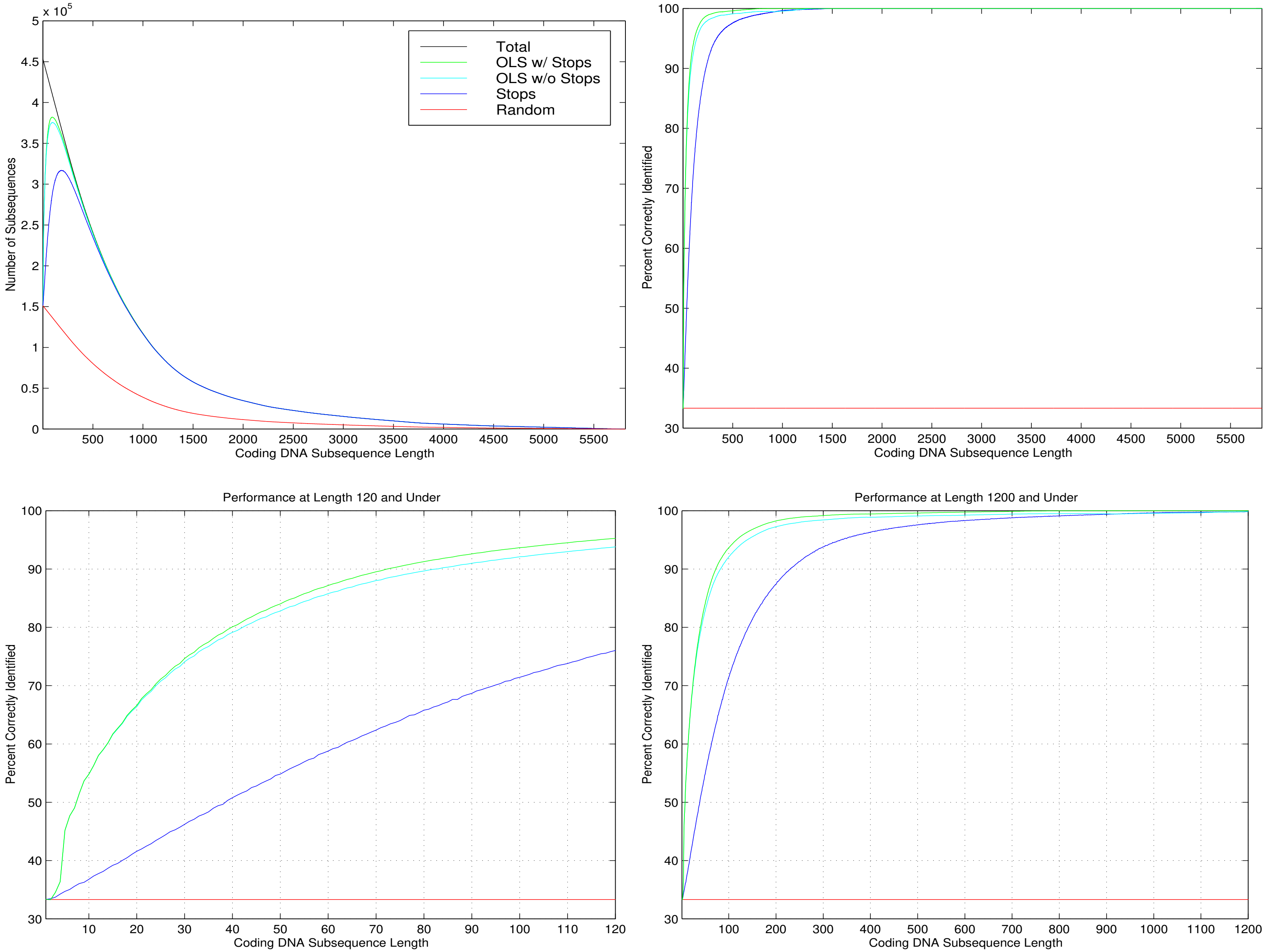


Figure 10-4: Frame-Test. Resolution: 1 Amino Acid (63 parameters).

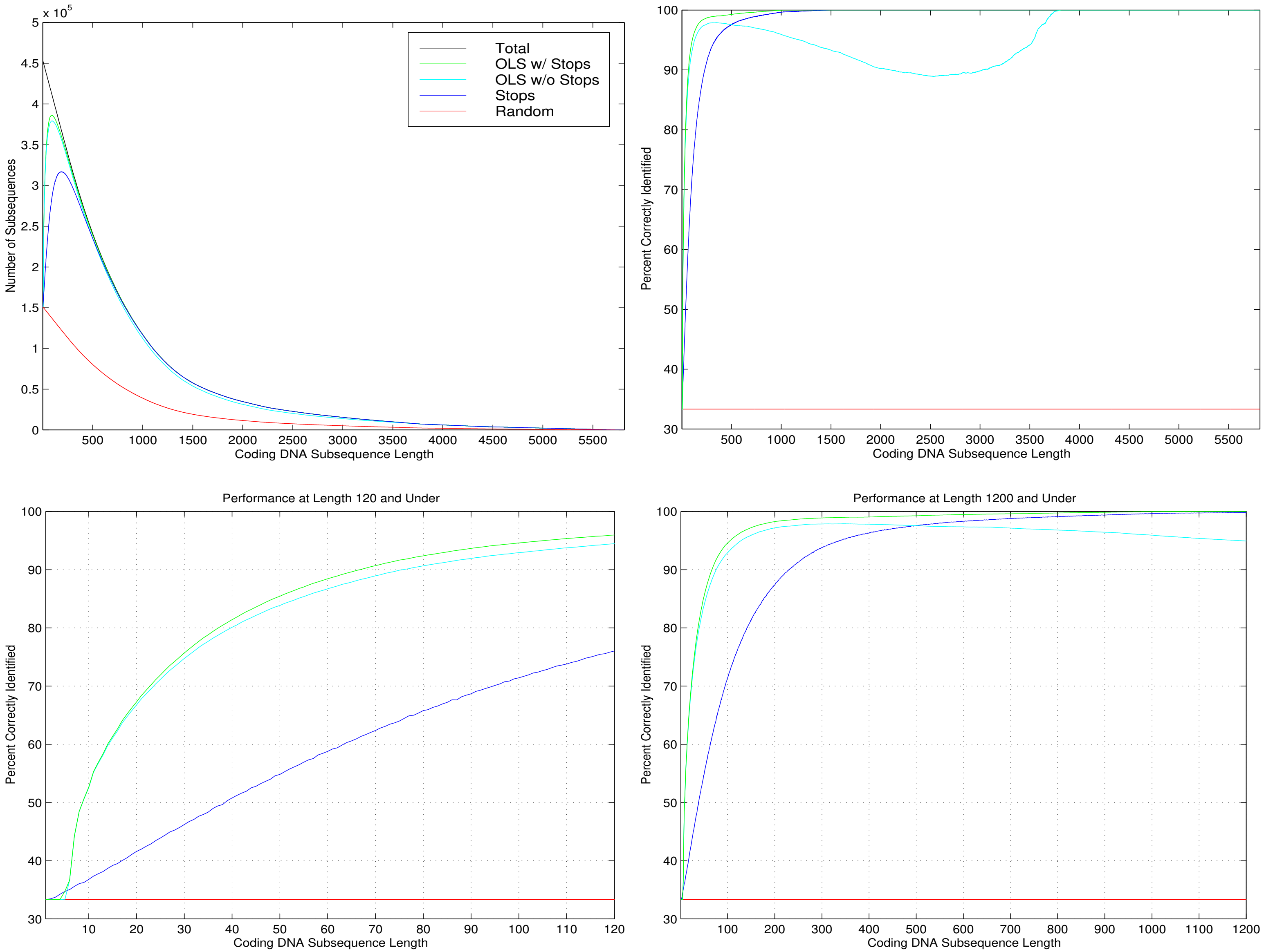


Figure 10-5: Frame-Test. Resolution: 2 Amino Acids (1203 parameters).

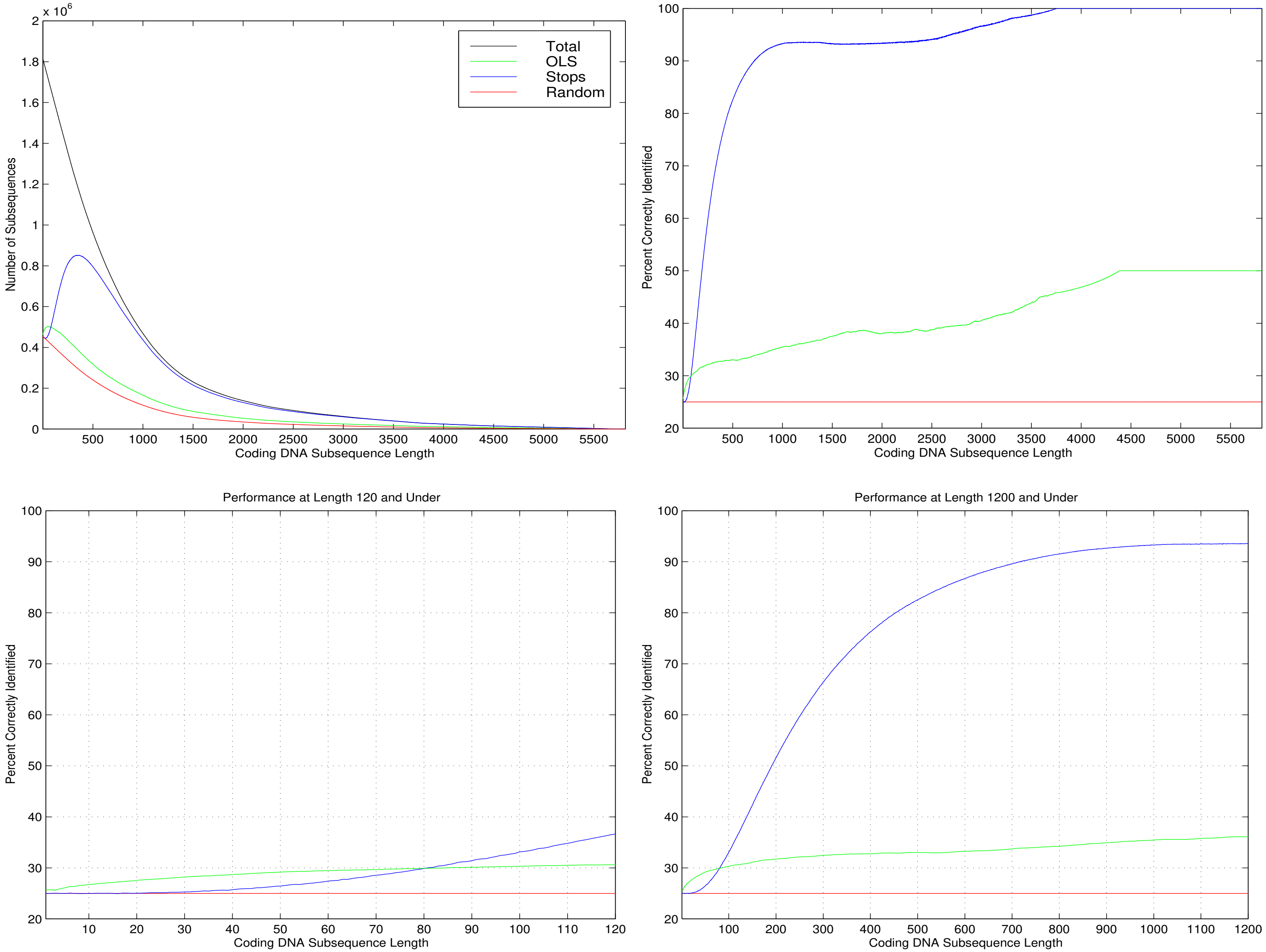


Figure 10-6: Orientation-Test. Resolution: 1 Nucleotide (5 parameters).

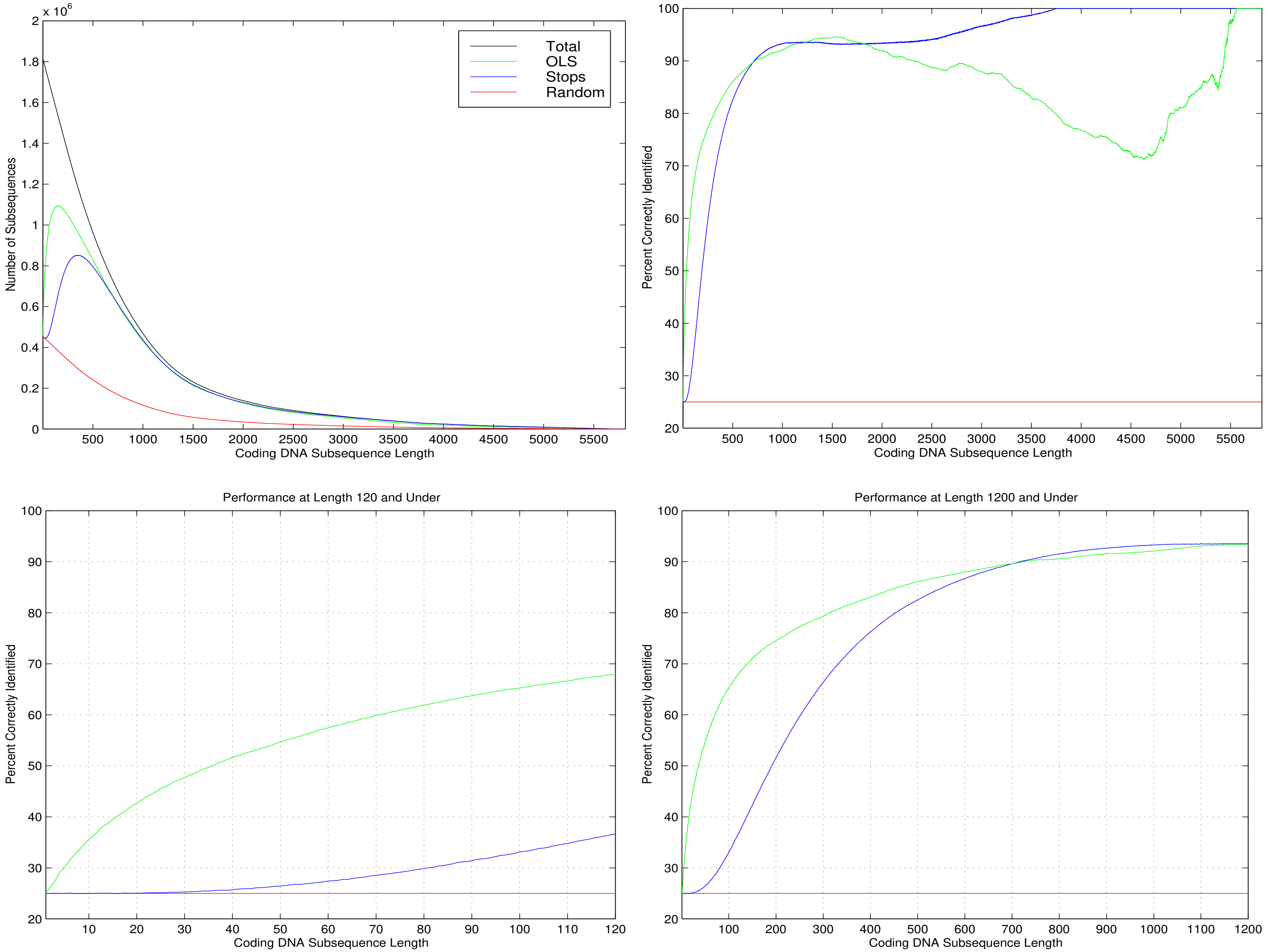


Figure 10-7: Orientation-Test. Resolution: 2 Nucleotides (17 parameters).

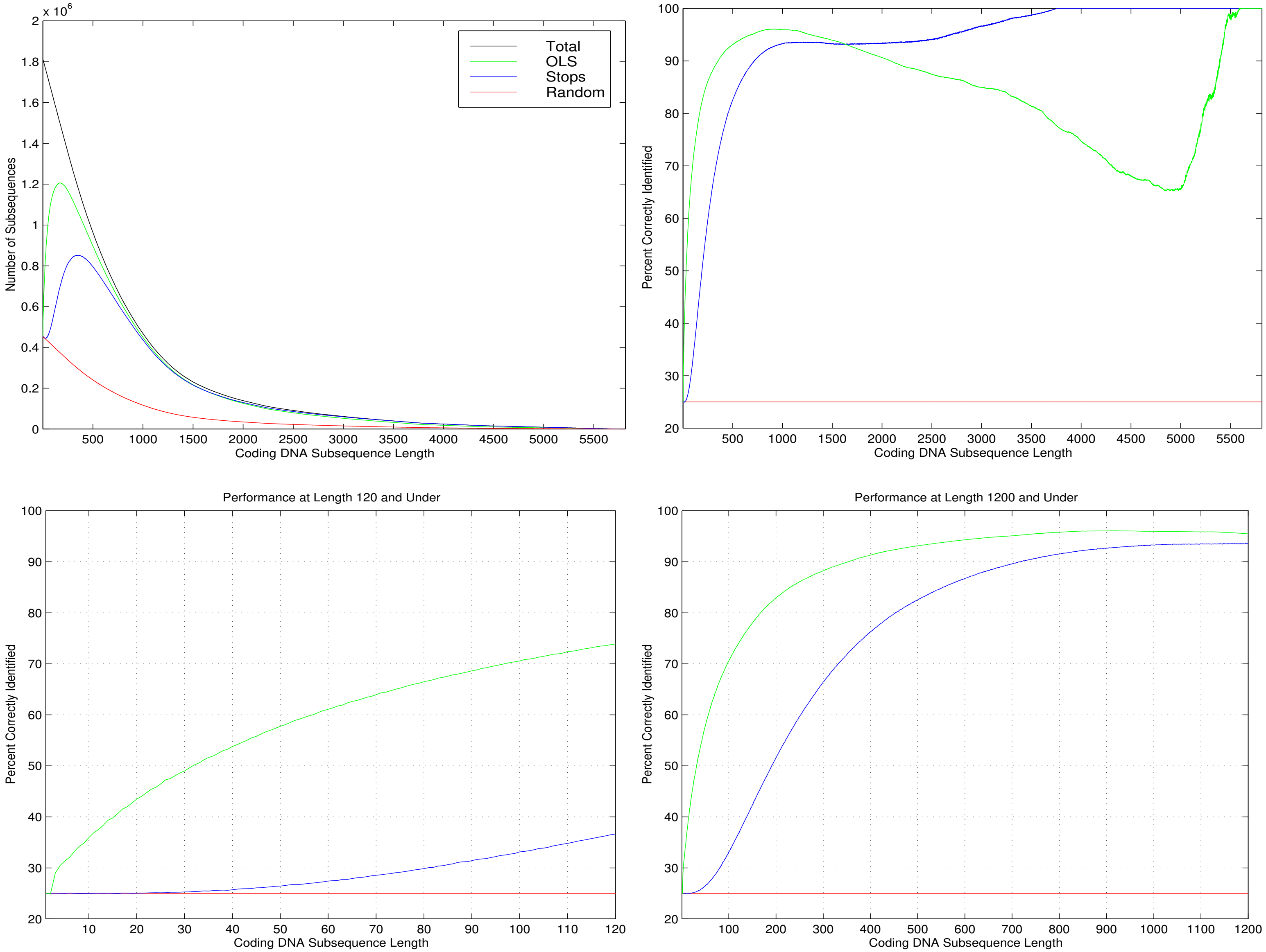


Figure 10-8: Orientation-Test. Resolution: 3 Nucleotides (65 parameters).

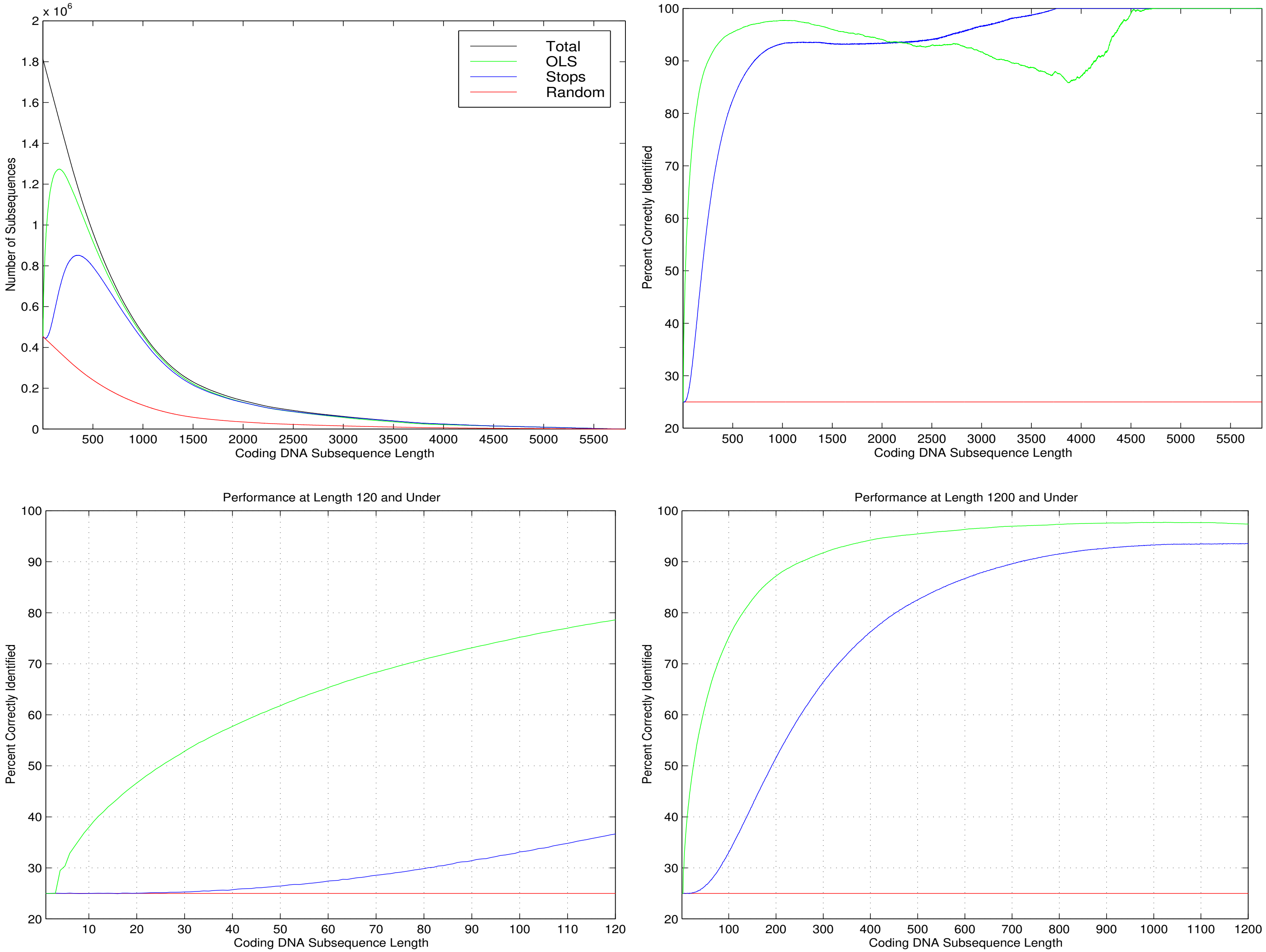


Figure 10-9: Orientation-Test. Resolution: 4 Nucleotides (257 parameters).

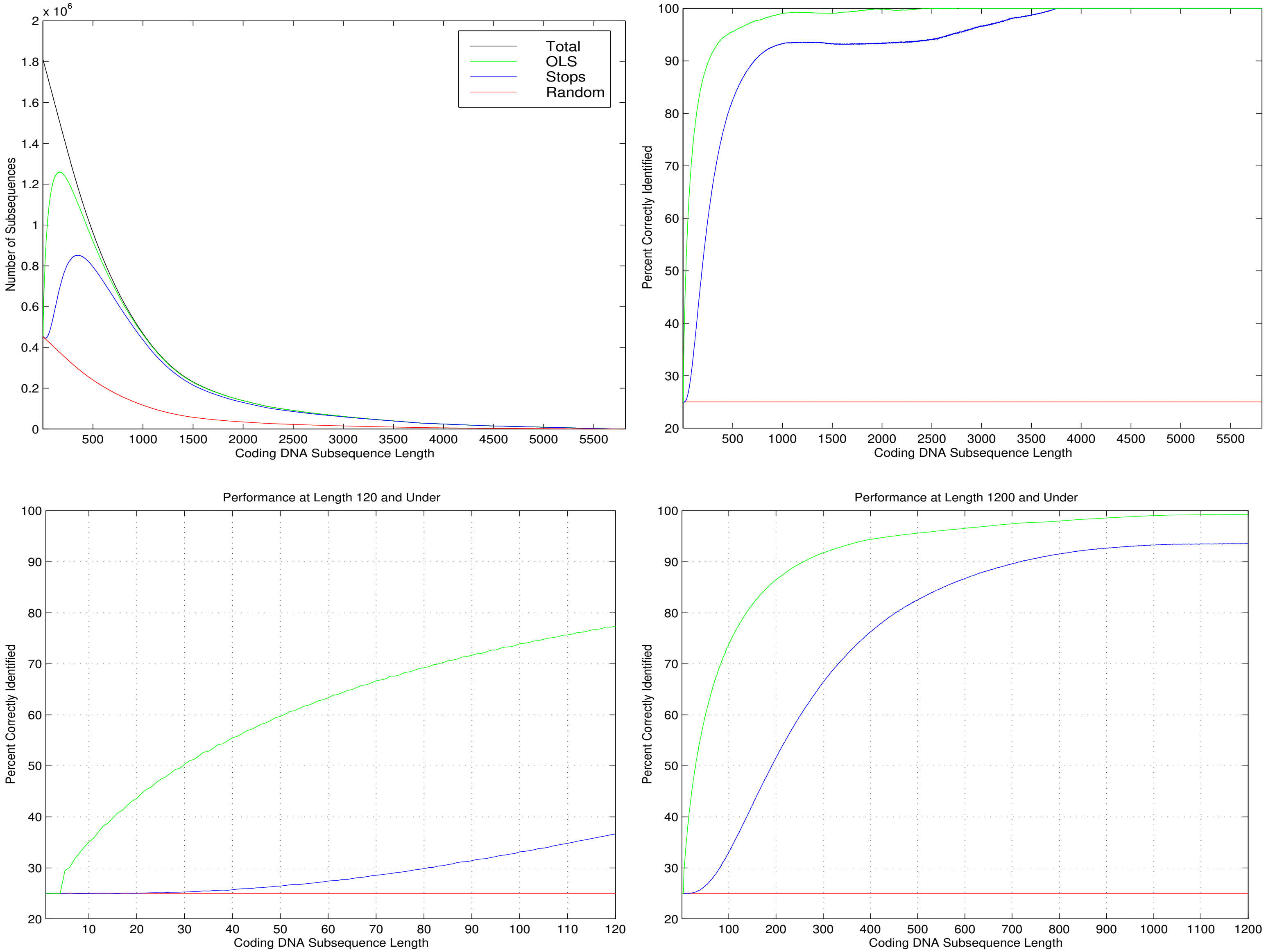


Figure 10-10: Orientation-Test. Resolution: 5 Nucleotides (1025 parameters).

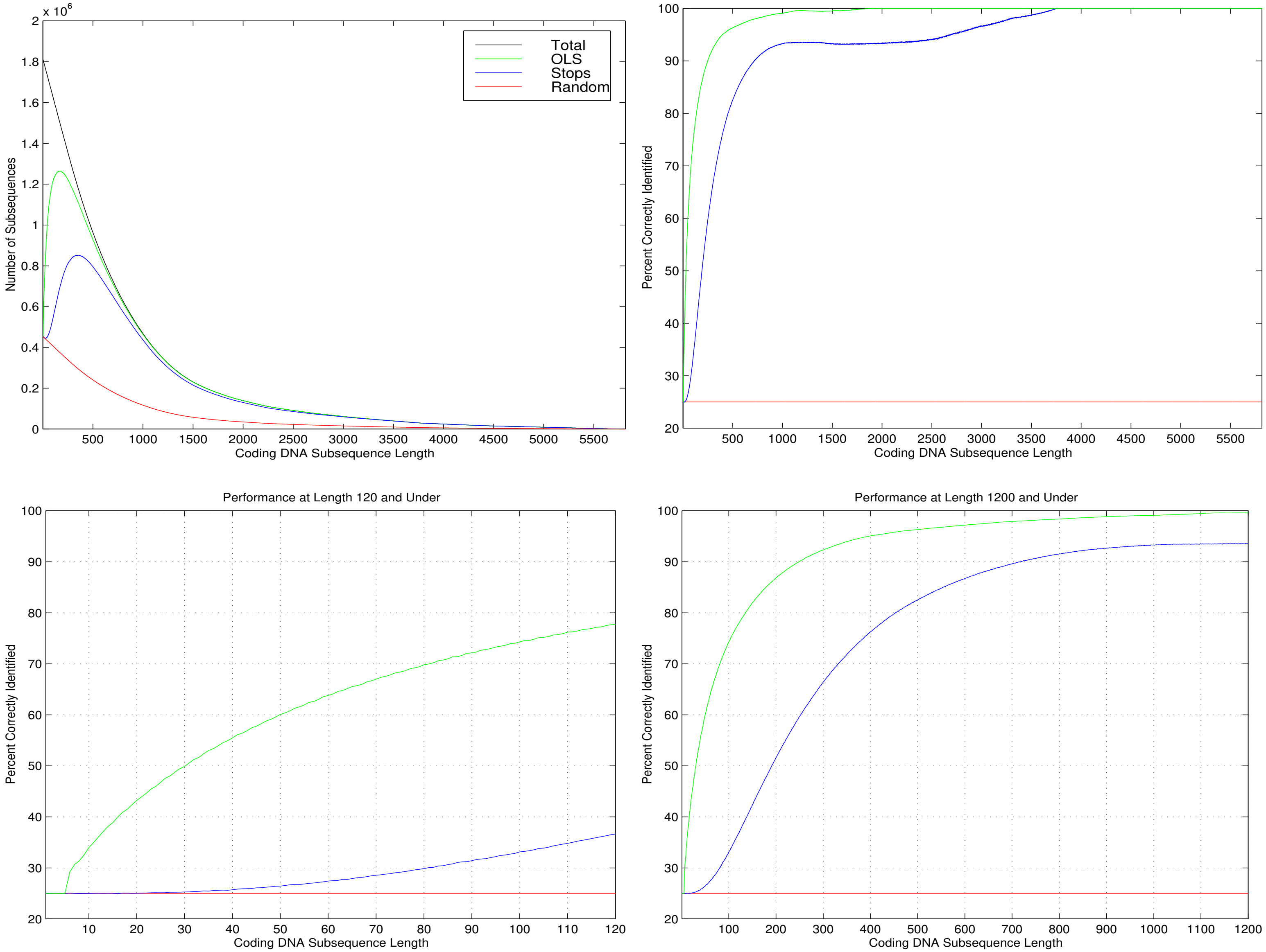


Figure 10-11: Orientation-Test. Resolution: 6 Nucleotides (4097 parameters).

Chapter 11

Visualization

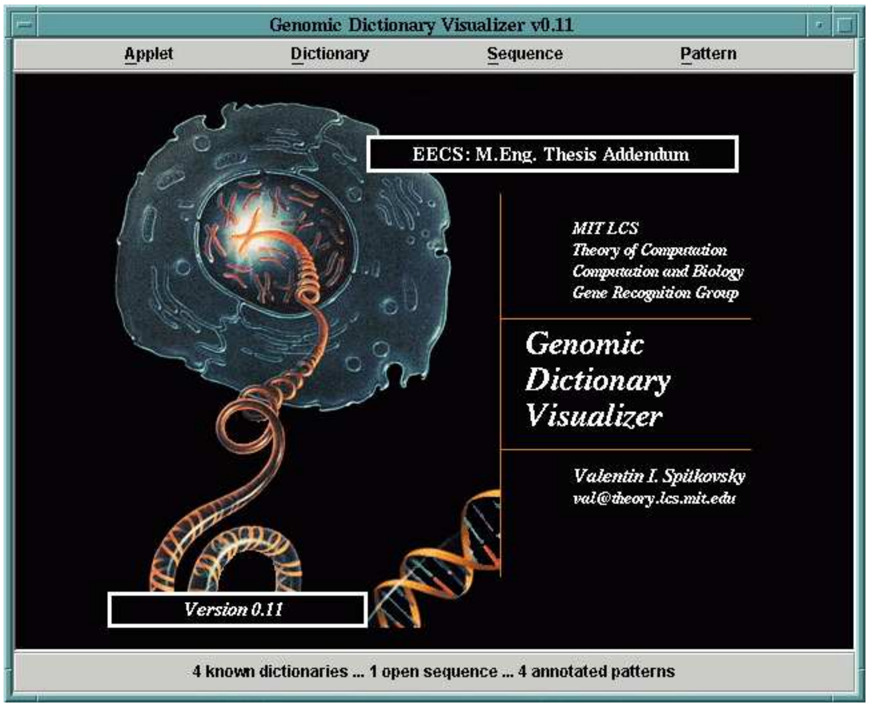


Figure 11-1: Genomic Dictionary Visualizer (GDV)

11.1 Overview

Together with Omprakash D. Gnawali, the author tackled the challenge of creating a visualization tool for the dictionary as a final project for Seth Teller's Computer Graphics (6.837) class in Fall 1999. The remainder of this chapter consists of excerpts from our final project report along with features the author later introduced on his own.

Five intensive weeks of brainstorming and coding culminated in a stand-alone Java applet: the Genomic Dictionary Visualizer (GDV). The GDV tool should prove to be useful to researchers involved in computational biology. It facilitates quick and simple visualization of interesting regions of arbitrary genetic material. While a lot of features are missing, GDV's fundamental contribution is its completeness: all of the basic functionality is there and the framework for adding features exists. To our knowledge, GDV is the first and only tool for visualizing genetic sequences.

11.2 Motivation

The dictionary generates a lot of information, which can be passed on to heuristics that implement the artificial intelligence and statistical analysis which, using the raw dictionary hit data, solve high-level gene recognition subproblems. The data shuttled between these abstraction layers is very rich in information. Some of this information may be lost or masked by the high-level programs — it would be very useful if the researchers could have access to this knowledge base. While it is possible to leave a trace of the hits found by the dictionary in the form of a text file, such communication of information is inefficient, from the human point of view: the amount of data is huge and it is virtually impossible to extract any insights for more than a few isolated regions.

A visualization tool which could depict the similarities between sequences, point out known biological signals, allow context switching between sequences on-the-fly, and just simply display rudimentary statistics is greatly needed. This demand is fueled by researchers specializing in gene recognition: if they could visualize the information their algorithms are getting, they could improve their own understanding of how those algorithms behave. That would be useful for drawing insights and for debugging. More importantly, this visualization tool would come in handy to many biologists. Having gotten a sequence hot off the press, s/he may want to know what it is s/he is looking at. Combining the GDV with some known dictionaries, the biologist could very quickly pull up the sequence on the screen and highlight the areas of activity. This would yield some clues as to what the researcher is dealing with. Parts of the genetic sequence may match some known proteins, and looking at the names of the matching sequences may even hint at the function of genes within the data.

As far as we know, such visualization tools either do not exist or aren't widely available. Yet from a computational point of view, the functionality sought is far from overwhelming. While the project may not have much to do with graphics per se, it is a perfect example of scientific visualization. The tool in question has everything to do with the problem of effective communication of a wealth of knowledge.

11.3 Goals

Our intent was to rectify the problem of a missing means of communication of biological sequence data to the scientific community. We set out to solve the problem through visualization by utilizing a very high-bandwidth communication channel we humans have:

our eye-sight. The challenge we faced was finding a fast and elegant way to deliver the important information in an understandable way, while keeping the visualization intuitive and easy to use. We have been able to split our task into several logical components or goals:

- Abstract representation of a single sequence:

  Global view of the sequence, local views of the sequence — ability to zoom in on actual characters, annotation of interesting user-defined patterns — for instance the `ATG` and the stop codons (in the case of DNA), annotation of interesting user-defined regions — perhaps the user already knows that some portions of the sequence are exons or introns (again, in the case of DNA), etc.

- Abstract representation of a query result:

  Initially, this data is a huge stream of text. We planned to set up filters, to process the information only at the desired resolution. The user may be interested in viewing only a very small subset of the result, and the abstract visualization could run very fast, as there won't be much real geometry to render. The abstract representation had to provide insights into the hidden properties of the hits.

  We intended to explore varieties of abstract representations to find a subset suitable for our task. Some basic thoughts: graphing the number of hits through locations of the input sequence, somehow incorporating the lengths of the hits into the chart, highlighting the sources of the hits if multiple dictionaries are queried using color.

- Abstract representation of just two sequences:

  The plan was to enable the user to identify a particular match, take a look at the sequence from which the match came, and compare that sequence to the input sequence at a very high level of detail. An interesting representation could include a two-dimensional image with the sequences' positions on each of the axes and the color (or surface value) at each location evaluating to some informative function. A simple function could be a binary zero-one value based on whether or not the characters at the two positions match; then the easily identifiable diagonal streaks would represent subsequence matches. A more complicated function could be something like the number of matches of some length that go through both of these locations.

- Interaction:

  While the visualization tool would be incredibly useful even in its static form, the users could benefit tremendously if empowered to interact with it dynamically. Letting the user play with the parameters — perhaps turning a knob hooked up to the hit cutoff level — and immediately see the effects of the actions could bring about further insights. Being able to see clusters of hits spring up at various areas as the hit cutoff altered could alert the user and direct her/his attention towards the interesting regions. Something like this could even be animated.

Our most important goal, however, was to end up with a finished product. In our view, it was more important to produce an intuitive, usable, stable, stand-alone tool, however simple, rather than a highly complex contraption with broken features.

Because there was a potential for visualizing a lot of data, we stressed that the tool must be fast; at the same time, the usefulness of the tool depends on its accessibility. With these concerns in mind, we chose to implement the visualization tool as a Java applet communicating with the dictionaries over TCP connections; all of the computationally intensive crunching was to be done on the host computer running the optimized C++ binaries. The applet itself was to be kept as light as possible, to maintain adequate speed. The graphics need not be glamorous: they need to be informative, efficient, and fast.

11.4 Achievements

Despite the challenge to produce a project in a matter of some four or five weeks, we managed to meet our most crucial goal: to produce a product capable of stand-alone usage. The Java applet we ended up with is as bare as could be, yet it is complete. It contains most of the basic functionality we set out to give it.

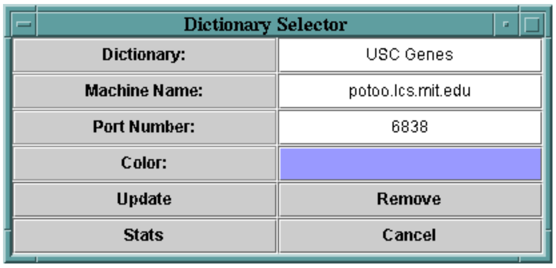


Figure 11-2: Dictionary Selector

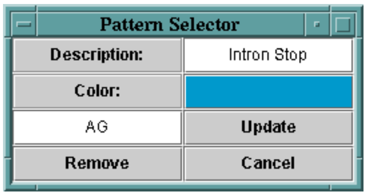


Figure 11-3: Pattern Selector

There is an easy way to quickly gather statistics about a known dictionary:

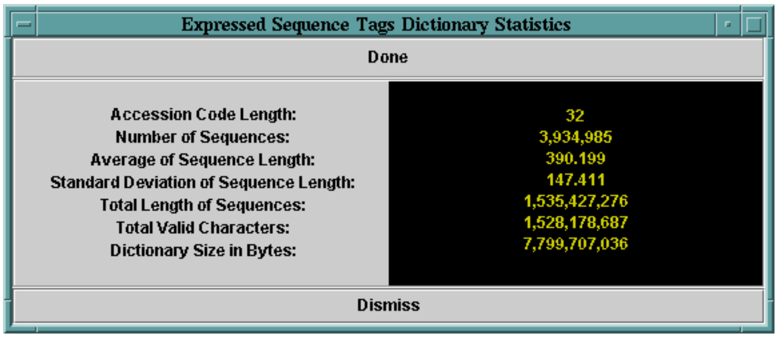


Figure 11-4: Dictionary Statistics

The applet maintains a dynamic base of known dictionaries and patterns to highlight:

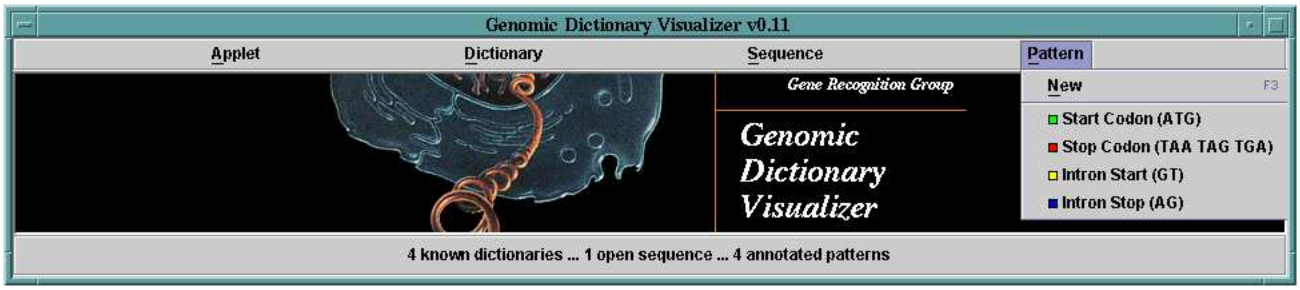


Figure 11-5: Pattern Menu

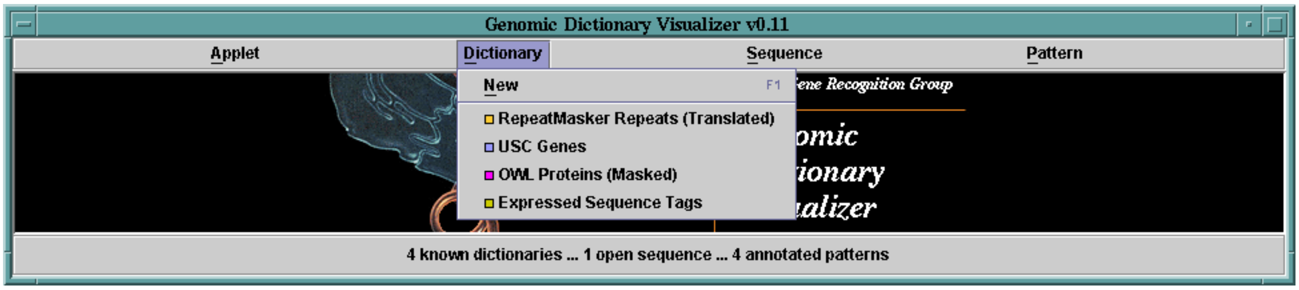


Figure 11-6: Dictionary Menu

Each dictionary and pattern is associated with a color which is later used to highlight the appropriate regions in loaded sequences. The user can open multiple sequences at once, pulling them out of either known dictionaries or files / URLs:

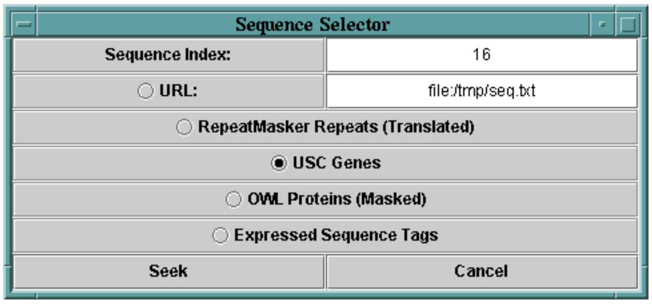


Figure 11-7: Sequence Selector

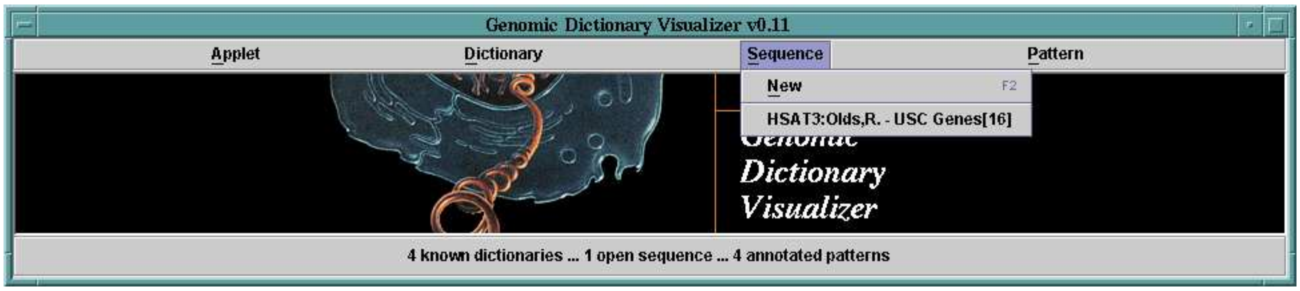


Figure 11-8: Sequence Menu

In addition to known patterns and dictionary queries, a sequence can be annotated via known segments if some of the structure has been solved:

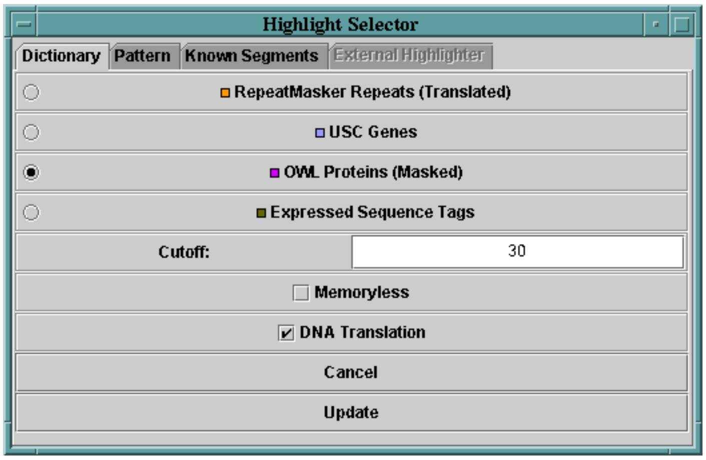


Figure 11-9: Dictionary Highlight Selector

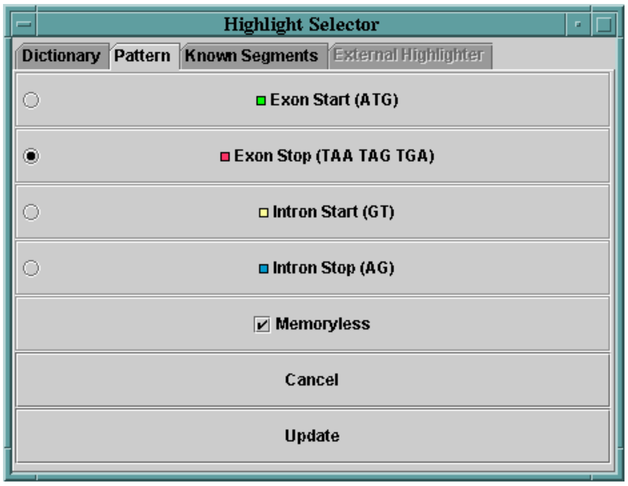


Figure 11-10: Pattern Highlight Selector

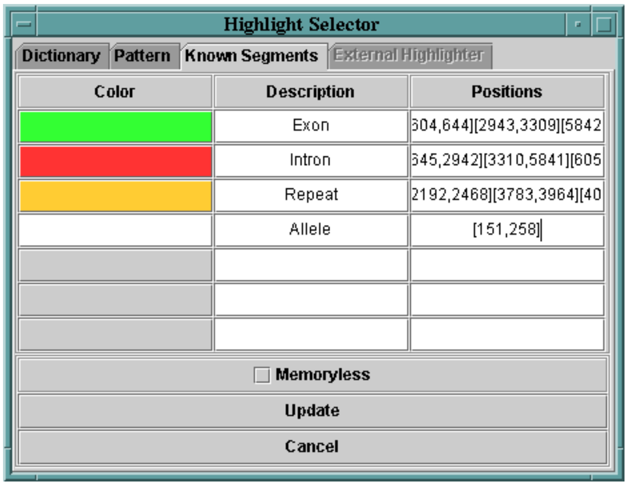


Figure 11-11: Known Segments Highlight Selector

The user is free to add highlights to a particular sequences to light up interesting areas of activity:

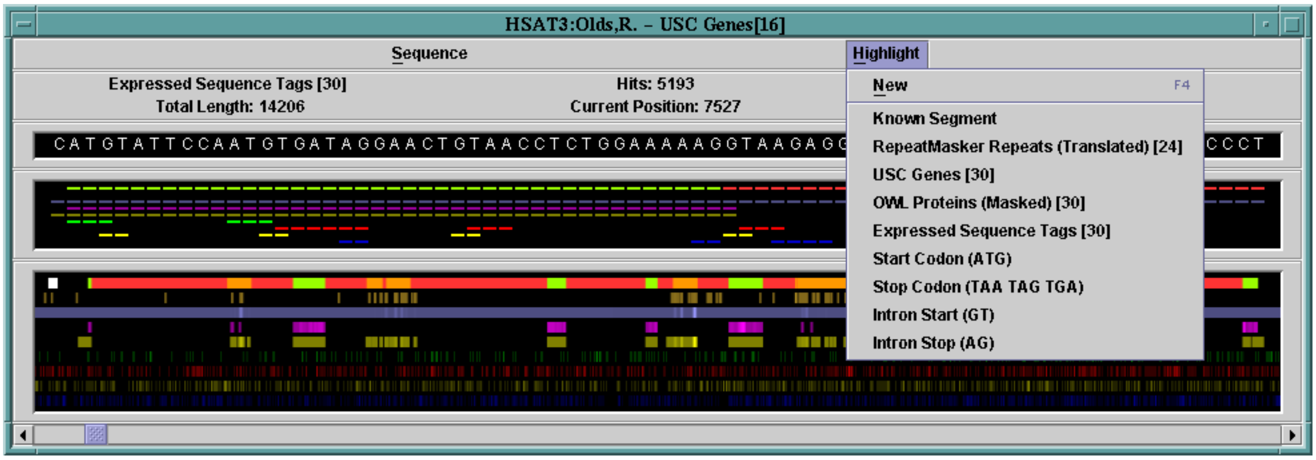
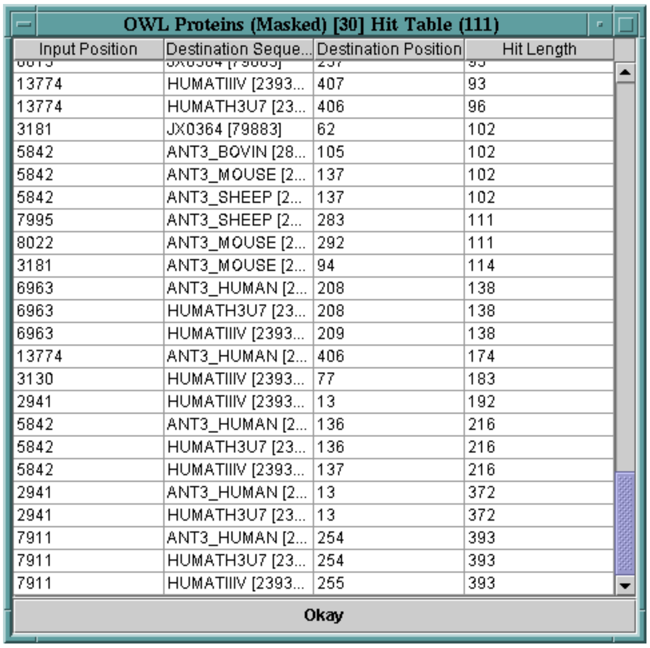


Figure 11-12: Highlighted Areas of Activity

The sequence depicted above is a known gene (`HSAT3`), see appendix B. Using the known segments interface, we've marked the exons, introns, and repeats in this sequence. We also highlighted the standard patterns: `ATG`, {`TAA`, `TAG`, `TGA`}, `GT`, and `AG`. In addition, we screened this sequence against four known dictionaries of masked proteins, translated repeats, genes, and expressed sequence tags. The user is free to load a list of actual matches — the program pulls up a table of hits — which the user can sort by all the values in any of the columns. To a biologist, the names of the sequences with long matches actually suggest the function performed by the proteins encoded by the exons of this particular sequence:


OWL Proteins (Masked) [30] Hit Table (111)

Input Position	Destination Seque...	Destination Position	Hit Length
13774	HUMATIIIV [2393...	407	93
13774	HUMATH3U7 [23...	406	96
3181	JX0364 [79883]	62	102
5842	ANT3_BOVIN [28...	105	102
5842	ANT3_MOUSE [2...	137	102
5842	ANT3_SHEEP [2...	137	102
7995	ANT3_SHEEP [2...	283	111
8022	ANT3_MOUSE [2...	292	111
3181	ANT3_MOUSE [2...	94	114
6963	ANT3_HUMAN [2...	208	138
6963	HUMATH3U7 [23...	208	138
6963	HUMATIIIV [2393...	209	138
13774	ANT3_HUMAN [2...	406	174
3130	HUMATIIIV [2393...	77	183
2941	HUMATIIIV [2393...	13	192
5842	ANT3_HUMAN [2...	136	216
5842	HUMATH3U7 [23...	136	216
5842	HUMATIIIV [2393...	137	216
2941	ANT3_HUMAN [2...	13	372
2941	HUMATH3U7 [23...	13	372
7911	ANT3_HUMAN [2...	254	393
7911	HUMATH3U7 [23...	254	393
7911	HUMATIIIV [2393...	255	393

Okay

Figure 11-13: Hit Table

Finally, the user can easily pull out a matching sequence and run sequence-to-sequence comparison on the two strands.

Our final product works; it is a simple implementation of functionality that is demanded. Yet it is disappointing how few features we actually managed to implement in the short time we had. Some blatant omissions come to mind: animations of the amount of activity in a region as a function of the cutoff length, various representations of the amount of activity — not just a gradient of colors — graphs, nonlinear scaling of the graphs / colors, etc. A million ideas come to mind just thinking of the table of hits: we did implement the sorting routine, but there are other features just begging to be introduced. For example, clicking on a hit could move the local viewing region of the sequence to the position where the hit occurs, information could be gathered not just based on hits but incorporating their sources: which sequence produced the most or the longest hits? Which sequence had the most total or consecutive area covered? Two-sequence representation could get away from flatland and attempt to draw surface maps with peaks where the sequences are similar. The list goes on and on. We didn't get a chance to allow for user-defined highlights, which would let the user plug in his or her own class files for annotating sequences. However, we developed a framework for a `HitSet` class that would make such a future extension very simple. This would be useful for annotating structures that have variation (for example, pyrimidine tracks — regions that are rich in `G`s and `C`s — that occur frequently near the ends of introns) as well as the presentation of various statistical analyzers (for instance frame-tests which compute conditional probabilities — if this segment were in an exon, what is the probability that it would be in frame zero, one, or two).

A basic aim that we met was to put the applet on the web. To do this, we needed to learn about `.jar` files. However, we failed to learn how to sign applets and to create certificates to allow users to set up their browsers to trust our product. Despite these holes, we are not too upset. The number of ideas on our wish-list is growing and we are psyched enough to follow through to produce a more robust and a more useful applet, getting away from version 0.01.

First of all, the distributed networking model must be improved. As of now, we've got a very simple blocking server in C++ which serves as a wrapper for the dictionary code RPC. We also have a Java client model for interacting with the server. The server is rather fault-tolerant and reliable; the client is also mostly fine, as we have come up with a good interface for passing data (we use compression on top of that to utilize the bandwidth fully). Still, expanding the server model to efficiently interact with multiple clients simultaneously poses an interesting theoretical and practical challenge.

Second, the user interface needs to be designed with the biologist in mind. Although we meant to, we didn't get a chance to ask our friends in course seven what they would want in a visualization tool — the sort of representations of genetic material they found most expressive in their bio books, etc. We had enough trouble just trying to get the GUI to work, seeing as how neither of us programmed GUIs before.

Finally, we suspect that our usage of the Java model is highly suboptimal. Our interfaces probably generate objects right and left, sucking off memory and valuable resources from the actual visualization methods. Now that we have some idea of how Swing works, redesigning the code could produce a faster and more light-weight program.

The major source of both help and trouble was Java. The great things about it are the tremendous wealth of specifications, example code, and tutorials available on the web for this language. It was nice not to have to worry about platform-dependent issues and just know that our code will run anywhere. It was also great to avoid dealing with motif and Unix network programming, for the most part. Finally, it was very useful to be able to go home for Thanksgiving and to continue to work on the project from there, now running the applet in Windows instead of Linux. But the troubles with Java almost outweighed the benefits. It's buggy... it's slow. Compiling the code takes forever, even with well-designed makefiles. Running it takes even longer. Various bugs creep up, driving you nuts trying to pin them down, only to find out that everything works fine on a different platform, meaning that it's a Java bug. Some features of our programs fail on the Suns — sometimes the networking blocks, sometimes the menus don't get drawn correctly. On the SGIs, the networking almost always fails and the rightmost menu-bar is always drawn wrong. In Linux, the program just hangs every now and then. Surprisingly enough, Windows has been the most stable platform... We can only hope that Java 1.3, to be released soon, will fix most of these bugs. When will Java compile to faster code is another question.

11.5 Acknowledgements

The most useful resources have been our TA Kari-Anne Kjølaas who guided us via weekly checkpoint meetings, a great search engine Google which let us find relevant info without fail, an extensive Java API Reference and a set of Java Tutorials. Finally, we ended up using a modified version of the image from The National Human Genome Research Institute as a front-page for our program.

11.6 Conclusion

A new visualization tool is now available for highlighting common regions between sequences. The standard applications for the dictionary tool are augmented with a graphical user interface: a high-bandwidth channel for absorbing the results. The implementation is relatively fast, light, elegant, and informative. It should be extremely useful to biologists in general and to researchers in computational biology, especially in the fields of gene recognition and gene annotation.

Chapter 12

Summary

We have presented the fast genomic dictionary, a new tool for rapidly finding common regions between strings. The fully automated approach that the dictionary provides has already proved itself in a number of applications in computational biology, most prominently in the area of gene recognition. Our tool presents a full-fledged suite of general string matching algorithms which should be incredibly useful to anyone dealing with large databases of strings and especially to researchers actively involved in the Human Genome Project.

More importantly, we hope to have sparked the reader's interest in computational biology. It is a beautiful and hot new field, full of surprises and exciting work to be done. This author found it very rewarding to find ways to apply mathematics and computer science to a seemingly unrelated realm of biology. Knowing that tight asymptotics mean not only efficient code but also speedier discoveries in molecular biology is very fulfilling: indirectly, this work may expedite cures for various genetic disorders and diseases.

Appendix A

Source Code

A.1 Common Files

A.1.1 Alphabet Class with Examples

Alphabet.h

Alphabet.cc

dna.alpha

aa.alpha

aanostops.alpha

A.1.2 Biology Class

Biology.h

A.1.3 iTools Class

iTools.h

A.1.4 Other Sources

Header.h

Header.cc

SymMatrix.t

charPile.java

intPile.java

longPile.java

A.2 Communication Files

A.2.1 Uncompressed Low-Level I/O

CommTCP.h

CommTCP.cc

CommReader.java

CommWriter.java

A.2.2 Compressed Low-Level I/O

CommZTCP.h

CommZTCP.cc

CommZReader.java

CommZWriter.java

A.2.3 High-Level I/O

Comm.h

Comm.cc

A.3 Dictionary Files

A.3.1 Builder

Builder.h

Builder.cc

A.3.2 DicFiles Class

DicFiles.h

DicFiles.cc

A.3.3 Dictionary Class

Dictionary.h

A.3.4 Remote Procedure Calls

Server.cc

DictionaryUser.java

Channel.java

Dictionary.java

A.4 Support Files

A.4.1 Scanners

owl2plain.flex

est2plain.flex

usc2plain.flex

A.4.2 Utilities

Mask.cc

A.4.3 Scripts

runServer.sh

owlPlain.sh

A.4.4 Tools

dDiff.cc

Appendix B

USC Gene #16 (HSAT3)

This is the entry for USC Gene #16 (HSAT3) which we visualized using GDV. The picture seems more informative.

```
LOCUS       HSAT3      14206 bp    DNA             PRI       25-JUN-1997
DEFINITION  H.sapiens gene for antithrombin III.
ACCESSION   X68793 S52236 S52240
VERSION     X68793.1  GI:28906
KEYWORDS    antithrombin; antithrombin III gene; AT3 gene; plasma protein;
            serine proteinase inhibitor; serpin.
SOURCE      human.
  ORGANISM  Homo sapiens
            Eukaryota; Metazoa; Chordata; Vertebrata; Mammalia; Eutheria;
            Primates; Catarrhini; Hominidae; Homo.
REFERENCE   1  (bases 1 to 14206)
  AUTHORS   Olds,R.
  TITLE     Direct Submission
  JOURNAL   Submitted (12-OCT-1992) R. Olds, Institute of Molecular Medicine,
            John Radcliffe Hospital, Oxford OX3 9DU, UK
REFERENCE   2  (bases 1 to 14206)
  AUTHORS   Olds,R.J., Lane,D.A., Chowdhury,V., De Stefano,V., Leone,G. and
            Thein,S.L.
  JOURNAL   Biochemistry In press
REFERENCE   3  (bases 1 to 14206)
  AUTHORS   Bock,S.C., Marrinan,J.A. and Radziejewska,E.
  TITLE     Antithrombin III Utah: proline-407 to leucine mutation in a highly
            conserved region near the inhibitor reactive site
  JOURNAL   Biochemistry 27 (16), 6171-6178 (1988)
  MEDLINE   89050967
  REMARK    Erratum:[[published erratum appears in Biochemistry 1989 Apr
            18;28(8):3628]]
REFERENCE   4  (bases 1 to 14206)
  AUTHORS   Olds,R.J., Lane,D.A., Ireland,H., Leone,G., De Stefano,V.,
            Wiesel,M.L., Cazenave,J.P. and Thein,S.L.
  TITLE     Novel point mutations leading to type 1 antithrombin deficiency and
            thrombosis
```

```
  JOURNAL   Br. J. Haematol. 78 (3), 408-413 (1991)
  MEDLINE   91337920
COMMENT     related sequences: M21636-M21645.
FEATURES             Location/Qualifiers
     source          1..14206
                     /organism="Homo sapiens"
                     /db_xref="taxon:9606"
                     /chromosome="1"
                     /map="q23-25"
     variation       152..259
                     /note="polymorphism"
                     /citation=[3]
     exon            <605..645
                     /gene="AT3"
                     /number=1
     mRNA            join(<605..645,2944..3310,5843..6058,6964..7101,
                     7912..8302,10336..10400,13775..>13951)
                     /gene="AT3"
     gene            605..13951
                     /gene="AT3"
     CDS             join(605..645,2944..3310,5843..6058,6964..7101,7912..8302,
                     10336..10400,13775..13951)
                     /gene="AT3"
                     /codon_start=1
                     /product="antithrombin"
                     /protein_id="CAA48690.1"
                     /db_xref="GI:28907"
                     /db_xref="SWISS-PROT:P01008"
                     /translation="MYSNVIGTVTSGKRKVYLLSLLLIGFWDCVTCHGSPVDICTAKP
                     RDIPMNPMCIYRSPEKKATEDEGSEQKIPEATNRRVWELSKANSRFATTFYQHLADSK
                     NDNDNIFLSPLSISTAFAMTKLGACNDTLQQLMEVFKFDTISEKTSDQIHFFFAKLNC
                     RLYRKANKSSKLVSANRLFGDKSLTFNETYQDISELVYGAKLQPLDFKENAEQSRAAI
                     NKWVSNKTEGRITDVIPSEAINELTVLVLVNTIYFKGLWKSKFSPENTRKELFYKADG
                     ESCSASMMYQEGKFRYRRVAEGTQVLELPFKGDDITMVLILPKPEKSLAKVEKELTPE
                     VLQEWLDELEEMMLVVHMPRFRIEDGFSLKEQLQDMGLVDLFSPEKSKLPGIVAEGRD
                     DLYVSDAFHKAFLEVNEEGSEAAASTAVVIAGRSLNPNRVTFKANRPFLVFIREVPLN
                     TIIFMGRVANPCVK"
     intron          646..2943
                     /gene="AT3"
                     /number=1
     repeat_region   complement(2193..2469)
                     /note="Alu repeat element"
     exon            2944..3310
                     /gene="AT3"
                     /number=2
     intron          3311..5842
                     /gene="AT3"
                     /number=2
     repeat_region   complement(3784..3965)
```

```
                /partial
                /note="Alu repeat element"
repeat_region   complement(4007..4284)
                /note="Alu repeat element"
exon            5843..6058
                /gene="AT3"
                /note="A"
                /number=3
intron          6059..6963
                /gene="AT3"
                /note="A"
                /number=3
exon            6964..7101
                /gene="AT3"
                /note="B"
                /number=3
intron          7102..7911
                /gene="AT3"
                /note="B"
                /number=3
repeat_region   complement(7277..7558)
                /note="Alu repeat element"
exon            7912..8302
                /gene="AT3"
                /note="4"
variation       8131
                /gene="AT3"
                /note="polymorphism"
                /replace="a"
variation       8160
                /gene="AT3"
                /note="polymorphism; Pst1 site"
                /replace="a"
intron          8303..10335
                /gene="AT3"
                /number=4
variation       8521
                /gene="AT3"
                /note="Nhe1 site; polymorphism"
                /replace="t"
repeat_region   8670..8687
                /rpt_family="att"
repeat_region   complement(8692..8951)
                /note="Alu repeat element"
repeat_region   complement(8994..9264)
                /note="Alu repeat element"
repeat_region   8994..9264
                /note="Alu repeat element"
repeat_region   complement(9362..9633)
```

```
                     /note="Alu repeat element"
     repeat_region   9823..9867
                     /rpt_family="att"
     repeat_region   complement(9871..10152)
                     /note="Alu repeat element"
     exon            10336..10400
                     /gene="AT3"
                     /number=5
     intron          10401..13774
                     /gene="AT3"
                     /number=5
     variation       10427
                     /gene="AT3"
                     /note="Dde1 site; polymorphism"
                     /replace="c"
     repeat_region   complement(10732..11011)
                     /note="Alu repeat element"
     repeat_region   complement(12124..12405)
                     /note="Alu repeat element"
     exon            13775..13951
                     /gene="AT3"
                     /number=6
BASE COUNT     3715 a   3338 c   3217 g   3936 t
ORIGIN
        1 ccacaggtgt aacattgtgt tttccttgtc tgtgccaggg acaccttggc atcagatgcc
       61 tgaaggtagc agcttgtccc tctttgcctt ctctaattag atatttctct ctctctctcc
      121 ctctctccat aaagaaaact atgagagagg gaattacagg tagagggcta gaagtttttg
      181 gacattaact atttctatct tctgatttag ttaacgagaa acaaaaaatc ctgcagacaa
      241 gtttctcctc agtcaggtat ttcctaacca agtttgaggg tatgaacata ctctcctttt
      301 ccttttctat aaagctgagg agaagagtga gggagtgtgg gcaagagagg tggctcaggc
      361 tttccctggg cctgattgaa ctttaaaact tctctactaa ttaaacaaca ctgggctcta
      421 cactttgctt aaccctggga actggtcatc agcctttgac ctcagttccc cctcctgacc
      481 agctctctgc cccaccctgt cctctggaac ctctgcgaga tttagaggaa agaaccagtt
      541 ttcaggcgga ttgcctcaga tcacactatc tccacttgcc cagccctgtg gaagattagc
      601 ggccatgtat tccaatgtga taggaactgt aacctctgga aaaaggtaag aggggtgagc
      661 tttccccttg cctgccccta ctgggttttg tgacctccaa aggactcaca ggaatgacct
      721 ccaacacctt tgagaagacc aggccctctc cctggtagtt acagtcaaag acctgtttgg
      781 aagacgtcat ttcaagtgct ctccctccca ccccacctct tggggtaagg cctttcctaa
      841 gctacccctt gggtccctag cctaagaaac aagggggatg tcatccctgg tgtaaagatg
      901 ctgtgcagga agtcagcact cacgggatcc aggggacgct ccaaggggaa tccccagggc
      961 ctgccatcca tccgggaaga gagcaaatgc tacccatgag gacctcctca ctccctttt
     1021 gctctttctt ccactcagat ccaccccact ccaccccac ccaaatccca gtgacctttg
     1081 actaaagggc caaaactgct tccttttctc acaatgagag ttgtccctcc ctcaatgcca
     1141 cacacactcc cttcttcatc tgagttgtca caggaggcta gaaacggggt ggtggcacaa
     1201 ctgtcttggt tttaatttgt gcttcatagc cctcccaggt cctctcagcc tcaaattgca
     1261 tttccaaatg tagttgaagg gacagagtgg gcaaccgaag cagcagtgga gatgggaaga
     1321 tgaatggcag ggtcctctcc tctctctctc tgcttcttca gcctgccttc cacatctccc
     1381 ttggtgccgc tgcttctctc cggctttgca cctctgttct tgaaagggct gcagaactgg
     1441 actcagacca cgcaagaagg caagtccccc tcagctgccc cagcttccag ccagccccag
```

1501 gcttgcccaa cggaccacgt ccgtgaatct gcactgggtg cctgtctttc tctcccagga
1561 gaagatggga agatccagta cccacacaca gaccccccttg tgtacacgca ggaaccataa
1621 accagctgga ggcagcccct gccccaccct gtcttatcta caaaaaatat tacaagagac
1681 tttatctctt gatttgcttc atcgagtgtc ccaactacct cattttttta aaatgtgaaa
1741 ttagcttcat ttaccttcat tgaatccatg ttggcgacta ttaaaaattc caggcaataa
1801 aaagggatga gagcctgaac taaagcagtg gcaataactg gtgaaagagt aaaaaaacag
1861 aactgattga ctctggggtg aactgattga ctctggggtt tgactaaatg aggaggagag
1921 agggaggaat ccagggtgat tctcaggttt ctgtacggga ttcactgagc ccactcacag
1981 gagcaggcct gtgggggaga attaattacc agttcagttt ggtcctgttt ccctgaagaa
2041 cttgtaggag ttcctggtgg aactgtccag caaatagtca gtctggagct cagtggaagg
2101 gttagggctg gagctagaga tgtaggaatc ttcagcacac agatattgcc attgtttttg
2161 tttgtttgtt tgtttgttgt tgctgttttg aagacacagt ctcactttgt cacccaggtt
2221 ggaggtcagt ggcacaatct cagctcactg caaccttcgc ctcctgggtt caagtgattc
2281 ttctgcctca gcctccctag tagcttggga ctacagtgtg cgccaccaca cccagctaat
2341 ttttgtattt ttagtagaga cagggtttca ccatgttgtc caggctgatc tcgaacaccc
2401 aacctcaagt gatctgcatg cctcagcctc caaagtgctg ggattacagc gtgagccacg
2461 cacccggcca gatattgcct ttgctccatc catttcttct ttttctcttg tgttgctgaa
2521 atctctctgc ctgcatctat cagagtcctt ccccaaacag tttctgtaga tggctccccc
2581 taccaccctg actcttcact gggcactaaa gccgattttt taggcatgca cattccatgt
2641 cacaaacagg aagcttctca ttcttttttc tcccagcgtg gggaattgag cacataatac
2701 tccaaataac catcagatga ttctaattcc aacatgacca cgtccaggca actgaactgt
2761 cccctggcaa gaagtctagg actgaacctg tcccgggccc ctgtacttgg ttcaaaggat
2821 ttagcctttc tcttggccac accaggtggg ctggaatcct ctgctttact ggggcaaccc
2881 tgtggtgggc agtgggggcta ggggttgcag cctagcttaa cttggcattt tgtctccttg
2941 caggaaggtt tatcttttgt ccttgctgct cattggcttc tgggactgcg tgacctgtca
3001 cgggagccct gtggacatct gcacagccaa gccgcgggac attcccatga atcccatgtg
3061 catttaccgc tccccggaga agaaggcaac tgaggatgag ggctcagaac agaagatccc
3121 ggaggccacc aaccggcgtg tctgggaact gtccaaggcc aattcccgct ttgctaccac
3181 tttctatcag cacctggcag attccaagaa tgacaatgat aacattttcc tgtcacccct
3241 gagtatctcc acggcttttg ctatgaccaa gctgggtgcc tgtaatgaca ccctccagca
3301 actgatggag gtacgaccaa aggtcttctg cccagccacc ttgttaggag cacctttggg
3361 cttccatagg cccaagtcca atgattcctc aaccaacact gcagccacta ggggcgctca
3421 ttatgcatta cgattccctt tgaacatcac tgtgttataa ttccctttga aaatcatttt
3481 ttaaaaaatt agccaaggaa tcttggctat ctacttttta aatcctggtt tcctcttttg
3541 agcaccttaa aatgggggaa ggcttgtatc ttctctcaac ttcttttcag taattctttc
3601 atctatatgt ttactcatta atttgatcat ttatttattt attcattcag cacttcctct
3661 gtgccaggca atgtgtagtg ccagtccctc ctctggtgga agaagagtag ctttaccata
3721 tggtgacatc aggcatatag gctctcgtgg aaaaaaattc taggatagta tttttttttt
3781 tttgagatgg aatctcgctc tattgcccag gctggagtgc agtggtgcag tctcggctca
3841 ctccaaactc tgcctcccag gttcaagcaa ttctcccacc tcagcctcct gagtagatgg
3901 gattacaggc acacgccatc acgcccagct aatttctata tttttagtag agatggggtt
3961 tcaccacgtg gccagactgg tctcaaactt tttttttttt ttttttgaga cggagtctcg
4021 ctctgtcatc caggctggag cgcagtgcac gatctcagct cactgcaacc tctacctccc
4081 gggttcaagc aattctcagc ctcagcctcc cgagtagctg ggattacagg cccccggcac
4141 catgcctggc taattttttt tcttcttagt agagatgggg tttcaccatg ttggccaggc
4201 tagtcttgaa ctcctgacct cgtgatccac ctgccttggc ctcccaaagt gctgtgatta
4261 caggcgtaac gaccgcgcct ggcctcaaac tcttgacttc aagggatcgg cctgacttgg
4321 cttcccaaag tgctgggatt acaagcatga gccactgcac ggggcctagg atggtatatt
4381 gagaccaggg gcccaggaaa gccaagagaa gcctcaagga cgtgagagtg tttctggctc

4441 tgggaagtat ggatcatttc agctcagtga cttagttccc accccccttcc ccccactgcc
4501 ttttgtggga gggaagtagg gcatgataag atgaaatgtc atagattgat tgatcactgt
4561 tggcctctgg ggctatgaca agtcatggat ggaaacacta gatctttaat ctgtccttgg
4621 cttggctgca tgacagtctt tcttcaagtt ggatcacact ttggaagcag agttcatcaa
4681 tagggaggca tgagtccctt caagatggta tacggtgctt atttgaaact tggacactaa
4741 agtctgtggg tcttaggagg gttccttcta ttctagtggt caatttccat ggaacttcat
4801 cacctttgct cagggctctg gggtgagtta acccaagtct tcactctttg aaagaaattg
4861 tagatttaaa aactctgaag acacataata ctgccttctc tgggcccttc agtcattttt
4921 gtatacattg gtactggtct caaagtactt ccatatcact catgtctctg tccccaggta
4981 agatcttaaa tgtaaccctt cctacaagaa gaacagaaca gaacactgct tccaaaccac
5041 acatgttcct ttggtcctcc cctctacaca aacgccatgt gttgggaaag cagggtgaga
5101 ctaaatctct ctggagaaaa gagaaattca gcaccaagct tttgatcaaa agcataatcc
5161 ccccctaaaa aaagtgccta ttggagcaaa atcaggaaaa ccaaaggcag agaacagata
5221 aaaccaaaag gccttttgta gcctgaggag agagcatgga aagggcagga ggggaacagc
5281 ctcacccatt ttgccttggg gatggtgaag gtgggcattg ggggattcca acttcaaagc
5341 atggatgact tctaagtcct tttcagccct gagctcttag attctgagcc tgtttaatcc
5401 cttgctgata gattcactct tcctttttca cccctaccac cagtatccca gagcctccat
5461 gagcagctgg ccccagtaga tgccacaaaa gtgtttgtta cgagaaggac accgtctgat
5521 tctcttctct gtccagaatc accaagagga cttttcccat tccagcaaga aaacgtctgt
5581 gtgttgatct agaggcgttt agagacttta ggtggcaacc tagtctctct ttttcccttt
5641 atccttccta cccttcattc ttcttttatc cttttattca tcagaacaca agagttgagc
5701 atttatgctg tcccaggtac tgtgcttgaa ggagttaaca actgaggtgg ctattagtca
5761 gagactgacc agcatgtgct caccacccat gttaactagg cagcccacca aacccaccac
5821 cattttttttt tgacttctat aggtatttaa gtttgacacc atatctgaga aaacatctga
5881 tcagatccac ttcttctttg ccaaactgaa ctgccgactc tatcgaaaag ccaacaaatc
5941 ctccaagtta gtatcagcca atcgcctttt tggagacaaa tcccttacct tcaatgagac
6001 ctaccaggac atcagtgagt tggtatatgg agccaagctc cagcccctgg acttcaaggt
6061 gagttgcaga tgttacccct gacctccgag ttcttcctct ccactcagag attgaggagg
6121 tggagaaaca gcatccaaat tcacactgct ttgctgctga agactgctgg agggctgact
6181 aaaagttaga acccctgcaa tagttattct tacttgaaac ctgagaaatc aaaggtatcc
6241 atgcttggat tgtagtgact gcccagaaaa catgaattaa taatcaattc ttcattccat
6301 ccaccaactt caaatatata ccaaagggtg ttttgaagat gccagttcta caagatatct
6361 tacttaattt gaactgttat catggtcaaa taaagttggt acatgatgca tgttacattc
6421 tcctcttgga gattcatgaa gcacatgggc ctatgaaggt tctgagaaac tctgcaacaa
6481 agaaatctgt tggctttatt caatcggcat tcctcaaatg tatttgactg catgggcatt
6541 tctctcctcc atataacctg ccaaccccat ataacctgcc aaccccatat aacctgcaat
6601 cattcattgc ttcccctggc acatgccttg gaaattctac ttttgtgagt taaggttttc
6661 caaagtcaga gaaaataata ttttatcttc tttttcccag actattttcg ccttccttct
6721 tttcatttat ttcttcctat ttctttttgt ctttttcttc tgataatatt tattaactac
6781 aggaaagatt catggaacta tattagatat gtgaggcttc ccaatttggg ttagagcaat
6841 ggcttcttaa tcaaatggtg ggaaaggaca gagggatggt gagaaaaata aaatgctgcc
6901 tgggaaaatg gagaagccaa ttgaatagca caggtgagta ggtttatttt ctgttctcct
6961 caggaaaatg cagagcaatc cagagcggcc atcaacaaat gggtgtccaa taagaccgaa
7021 ggccgaatca ccgatgtcat tccctcggaa gccatcaatg agctcactgt tctggtgctg
7081 gttaacacca tttacttcaa ggtactcaga atggccctgg agagacccca gggacttcct
7141 cttgctcttc agcttacccc cttttttttt aaatggcgag accgaagccc tgagagggca
7201 aatggactgc cgaaagctac acaggtacag gtcagcaggg caggtcaatc tattatttat
7261 ttatttattt atttttgaca gagtctcgct ctgtcgccca ggctggagtg cagtggcgtg
7321 atctcggctc actgcaagct tcgcctcctg ggttctcggc attctcctgc ctcagcctcc

7381 caagtagctg ggaatacagg cacccaccac catgcctggc taatttttttg gtttttttttt
7441 agtaaagacg gggtttcacc gtgttagcca ggatagtctt gatctcctga cctcgtgatc
7501 tgcccacctc ggcctcccaa agtgctggga ttacaggcat gagccaccgc gcccggcaga
7561 ttggcttctt tcacctagta aaatgcattt actgttcctt tgtgttttcg tggctttgtc
7621 actcatttct tcttagcatg gaatagcatt acatttggtc tggatgtacc acagtctgtc
7681 tattcatcta ctgaaggaca ttttggctgc ttccaaggtt tgacagttat gaataaacct
7741 actcataatt ccatcattct gacacagcca ttgttaacct ttttgtgcat atcccgccag
7801 tctttttttcc gaataattat atattaatgt aacactataa tatggatatg tctgtgtcaa
7861 taactatcct cctatgaatg tttgtgttct tactttgtga ttctcttcca gggcctgtgg
7921 aagtcaaagt tcagccctga gaacacaagg aaggaactgt tctacaaggc tgatggagag
7981 tcgtgttcag catctatgat gtaccaggaa ggcaagttcc gttatcggcg cgtggctgaa
8041 ggcacccagg tgcttgagtt gcccttcaaa ggtgatgaca tcaccatggt cctcatcttg
8101 cccaagcctg agaagagcct ggccaaggtg gagaaggaac tcaccccaga ggtgctgcag
8161 gagtggctgg atgaattgga ggagatgatg ctggtggtcc acatgccccg cttccgcatt
8221 gaggacggct tcagtttgaa ggagcagctg caagacatgg gccttgtcga tctgttcagc
8281 cctgaaaagt ccaaactccc aggtttgtct aggaaggagt ttcctccctt ctccacccgc
8341 aaggtagtct gaccaaaagt ggaagagttg gagaaagaat agaaaggagc aacaagtcag
8401 gactcctgga tactgatcct agtttctact gctaatttgt ggaaatctct tttccttttg
8461 agacctcagt ttcctcttct gtaaaaggga agtttgttct tggatctcca tgggcccagc
8521 cagcactggt gccctgtgag tctgtatcag gtagaggaga tgggaccagg tggagaggaa
8581 tttgaaaggg cattggaatt cagagcaaag agacagatat taagagctgg ggaaatgtgg
8641 ttcccattac acaggcctca ctgacattta ttattattat tattattact tgagacagag
8701 tcttactctg ttgcccaggc tggagtgcag cggtgcgatc tcggctcact gcaacctctg
8761 cctcccgggt tcaagcgatt ctcatgcctc agcctcctga gtagctggga ttacaggcac
8821 acgtcaccat acctggtaat ttctgtattc ttagtagaga tgggtttcac catgttggcc
8881 aggatggtct tgaactcttg accttgtgat ccgcctgcct tggcttccca aagtgctggg
8941 attacaggcg tcacgaccgc acctggcaca ttaaaatatc ttttaaagaa gttggctggc
9001 cagggtggct cacgcctgta ataccagcac tttgggaggc tgaggtggga ggatcgtcta
9061 agcccatgag ttcgagacca gcctggacaa catagtgaga tggtctctac aaaaaataaa
9121 aaaaattagc caggcatggt gacgcacacc tgtagtccta gcttcttggg aggcagagct
9181 gggaggattg cctgagtccg ggaggtcaac gctgtggtgt actgtgatca caccactgca
9241 ctccagcctg agcaacagag tgaggtccta tcactaaata aataaataaa taataaaata
9301 gtttacgatg ttaagtaatt agatttatct ttattgacct tttttttttt tttttttttt
9361 tgagacgaag tcttgctctt gtcccccagg ctggagtgca gtggtgcaat cttggctcac
9421 tgcaacctcc accccccaga ttcaagtgat tctcctgcct cagcctccca aggagctagg
9481 attacaggcg cctgccacca cgcccggcta atttttgcat ttttagtaga aacggggttt
9541 cactatgttg gccaggctgg tcttgaactc ctgacctcag gcgatctacc tgccttggcc
9601 tcccaaagtg ctgggattac aggcgtgagc cactgtgcta ttgggctgtc tttaagctag
9661 ttttgaaaac taaaaatgtt gccagactgg aaagaaagat gttccttctg gatggagtga
9721 gtttttttctg taagaacaga gtcttgccgt tctctctcca caaaaagctg aagcctgaga
9781 atgaattatc aggagccatg ctgaacaagc ccaaagtact ttattattat tattattatt
9841 attattatta ttattattat tattattttt gagatgcagt tttgctcttg ttgcccaggc
9901 tggagtgcag tggcgtgatc ttggctcact gcaacctcca cctcccgagt tcaagcgatc
9961 tcctgcctca gtcttccaag tagctgggat tacacgatgc gccaccacac ctggctaatt
10021 tttgtatttt cacgatagag acaaggtttc accatattag ttagagtgtc tccaactcct
10081 gacctcaggt gatctgtaca ccttggcctc ccgaagtgct gggattacag gtgtgagcca
10141 ctgcacccag cccccaaagt actttattat ttttaacaca tattcattgt gagagtatga
10201 ttaggtgaag atttaggatt tcttcttatg tttcaaaaag ccccaaagga tctcttaatc
10261 caaactgaat tcccatctgt ggattgaagc caactttctc ccatctcaca aagacttctc

10321 cggtcttcct tccaggtatt gttgcagaag gccgagatga cctctatgtc tcagatgcat
10381 tccataaggc atttcttgag gtgagtacac cttccccact ctcttagggt acagaaagga
10441 gatgcatgaa cagcaggaac acgtggaaaa ggcctgtttc cagtgttaag gcatgcaaaa
10501 ggcctccaca ggctgctata atacagccct ctccaaaacc ttcatggtgt gattgttctg
10561 ccttccctcc cactacctct tctgtagcag gtcaagcggg aacacaaaca tttagggagg
10621 gtgatatagg aaaagaagcc agcaaaggcc atcaagaaga aatttacagc atgaggagaa
10681 ccagaagagt atggggtcgc agaaacccag ggagaatttt tttttttttt tgagacagag
10741 cttcgttcgc tcgttgccca ggctagagtg caatggtgcg acctcactac aacttctgcc
10801 tcccgcgttc aagcgattct cctgcctcag cctcctgggt agctgggatg acaggcatgt
10861 gccatcacgc ccggctaatt tttgtatttt tagtagaaac agggtttctc catgttggtc
10921 aggcaggtct tgaactcccg atctcaggtg atccacctcc ttggctcccc caaagtgctg
10981 ggattacagg catgagccac tgcacccggc catacctagg gagaagtttt aagaaaatgg
11041 atagcatcta gtaagaagac tcctgggctg ggcatggtgg ttcacacctg taaccccagc
11101 accttgggag gctgagatgg aagatcactt tgagcccagg agtttgcaac cagccctgag
11161 caacatagtg agaccctgtc tctaccaaaa aaatcttaaa aaaaaaaaaa aaagtttgga
11221 gactgcccat agtttacctt tccctgagga cagaatagtg tggccacatg cctaattgta
11281 atggatgaag agcaaatgga aggtaagaaa gggaagctgg tgagtgtgca tcagtgtctt
11341 aaagtgtgct ccaactagag cactagacta cactggagga aacgaaaagg tggtcaaata
11401 aatgcatatc ctctcatggg agatgaacag tacacactga catgctgagg tctgacaagt
11461 cccacagtaa agaagacggt tgaatatcac ttaacgtgtt cccccaaatg agatgtgcat
11521 ggaaccctgt gttagagtaa tagcgtgtac agcctgtgga acttctggtt ctcaagtaaa
11581 cactaaacta tggaccaaca gcagtagtta tctggggagc tttatctttg gagattctgg
11641 atgggccctg agaatctctg tatgtactaa actcctcagg ggattcttat gcaaacaatg
11701 agatttggga accactggta tagactattt tttgcgggag ccaggctgtg agggatagga
11761 gattggacaa tggtagagat gttcctagaa tcaaagaaat cgaaaagaat gaaggttgta
11821 gtcaaagaga aaggtttcag aggatggtgc taagtaaata tagatccagg gatcaaactc
11881 agaggaaagt ggaattttaa cgggagcgag gaaatgtgat agcttgaaag aacccagtta
11941 taatctgaga aagatgctat taatataatt tcaaaggtag agtagttctg agatggcaag
12001 tccaaggtat agccatggac aggtttgctt aagtggaata aggcaatgct cattaggttt
12061 gatgaaagaa atgggagact agggtgttga acgggtctcc taatgagatt tttttttttc
12121 attgaaatgg agtcttgctc tctcgcctag gctggaatgc agtggtatga tctcagttca
12181 ctgcaacctc cgcctcctgg gttcaagcga tcctctcgcc tcagcctccc aagtagctgg
12241 gactacaggt gcccgccacc acgcccgact aaatttttgt atttttagta gagacggggt
12301 ttcaccatgt tggccaggct ggtctcaaac tcctgacctt aagtgatcca cctgactcgg
12361 cctcccaaag tgctgggatt acaggcatga gccacgtgcc cggcctactg agatattttt
12421 aattgcctca aatgatagca ggagttggag tggacagaaa ggctaagtgc aaaaatcatc
12481 agtgtgggga tataatctat aggacaatga atgtcaatga cctttaagac aatagcaaga
12541 gtagaggtat tgaggtcaga acaagggatt ttacaagagt gctgtattaa tggttttgga
12601 agttaagatg acactgctca caccctcttt cacatggatt tttggaagaa agaacactta
12661 ggaagactgc aagggaaatt gagtcctcag ggttttaact ctcattgaat atcctctggt
12721 aaggactcca gttagaagtg gtcaactcag acctccttga ggggtctgag ttactattag
12781 gaagaagcag aggtctggat tcattttatc cacctgagcc cagtacacaa tatgtaagat
12841 ttttccatgg ttcttacaac aaagccgttt tctttgaaaa ccttgggaat ccttaataaa
12901 caggacccca actaattgta gacctgagaa gccattaaaa accagaatct gatttgaata
12961 aaaggatcct ggtcatgcaa acactctagt ctgctaataa gctaataatt tagtgctgga
13021 atgagcatga aataggtaat atggggagat agcgggtaag gaagggagga acaaaggaag
13081 gggaaggaag agtgagaagg aaggagaaga catcatcaac cagctccaca aaacccaggg
13141 agccggttaa tcatgtgctt tcattaagag cagaaacaga gttttagtga tattctgggt
13201 cctgaggcaa aattttctga aggtgtttcc ctctagatcg ctatcagcca tgttcaaata

13261 ccattgtttc agtctattac tccaagaaaa tggcatctcg tccagccaga gaacccacct
13321 cttttcatag gccctaggtc ctgagtggct ctttggagta gctgtatctt ggatcttgat
13381 gctccaagag tgaaactgtt tctttcaact atggagttca gatcttgagc caaaaatctt
13441 tcagcggctg gtacaaaaaa aatccgctgt aaaaccattt acaatggtac cagccagaaa
13501 tgtgataacc tgtgtacatt cagatttctg ggttacctga atggaactct tacacttatt
13561 acctagcaca aggcttggac aaacacaagt accttacatt ctctgcatga aagaatgagt
13621 gaaagtagga ttctggaggg aatccaacct gacccaaatg tactttttac tggaaaacaa
13681 aagcatttga ggaattgctg tgtctgtgga tgatttacct gccaaaatga acggcagagt
13741 ggctaattta gttttattcc catgtgacct gcaggtaaat gaagaaggca gtgaagcagc
13801 tgcaagtacc gctgttgtga ttgctggccg ttcgctaaac cccaacaggg tgactttcaa
13861 ggccaacagg cctttcctgg tttttataag agaagttcct ctgaacacta ttatcttcat
13921 gggcagagta gccaaccctt gtgttaagta aaatgttctt attctttgca cctcttccta
13981 tttttggttt gtgaacagaa gtaaaaataa atacaaacta cttccatctc acattataaa
14041 tggactctgc atttgaaatg aagacaagga aaggggaaac atgctattgg ggcacatggt
14101 aaaatgatgc cttcaagttg ttctttaccc agtaaccaca tctggatcaa gaaaatgagg
14161 gagagagcga taaaagatgg tagacagcca gaaagggaag ggagag

Bibliography

[1] S. F. Altschul, S. F. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403-410, 1990.

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

[3] S. Batzoglou, B. A. Berger, D. J. Kleitman, E. S. Langer, and L. Pachter. Recent developments in computational gene recognition. *Documenta Mathematica, Extra Volume ICM 1998*, 1:649-658, 1998.

[4] S. Batzoglou, L. Pachter, J. P. Mesirov, B. Berger, and E. S. Lander, "Human and Mouse Gene Structure: Comparative Analysis and Application to Exon Prediction." In *Proceedings of the Fourth Annual International Conference on Computational Molecular Biology*, Tokyo, Japan, April 2000 (NY:ACM Press, 2000), 46-53.

[5] G. Benson. Sequence alignment with tandem duplication. *Journal of Computational Biology*, 4(3):351-367, 1997.

[6] C. Burge and S. Karlin. Prediction of complete gene structures in human genomic DNA. *Journal of Molecular Biology*, 268:78-94, 1997.

[7] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms* (MIT Press/McGraw-Hill, 1990).

[8] M. S. Gelfand, A. A. Mironov, and P. A. Pevzner. Gene recognition via spliced sequence alignment. *Proc. Natl. Acad. Sci. USA*, 93:9061-9066, 1996.

[9] D. Haussler, D. Kulp, and M. Reese. A representative benchmark gene data set. `http://www-hgc.lbl.gov/inf/genesets.html`, 1996.

[10] J. Henderson, S. Salzberg, and K. H. Fasman. Finding genes in DNA with a hidden markov model. *Journal of Computational Biology*, 4(2):127-141, 1997.

[11] X. Huang, M. D. Adams, H. Zhou, and A. Kerlavage. A tool for analyzing and annotating genomic sequences. *Genomics*, 46:37-45, 1997.

[12] Human Genome Project Information.
http://www.ornl.gov/TechResources/Human_Genome/home.html, 1999.

[13] R. J. Huskey. Overview of Transcription and Translation.
http://wsrv.clas.virginia.edu/~rjh9u/trtrrev.html and
http://wsrv.clas.virginia.edu/~rjh9u/trtrpict.html, 1997.

[14] D. Kulp, D. Haussler, M. G. Reese, and F. H. Eeckman. A generalized Hidden Markov Model for the recognition of human genes in DNA. *Proceedings of the 4th Conference on Intelligent Systems in Molecular Biology*, 1996.

[15] M. Laub and D. W. Smith. Finding intron/exon splice junctions using INFO, INTerruption Finder and Organizer. *Journal of Computational Biology*, 5(2):307-321, 1998.

[16] A. V. Lukashin and M. Borodovsky. GENEMARK.HMM: new solutions for gene finding. *Nucleic Acids Research*, 26(4):1107-1115, 1998.

[17] A. Mironov, M. A. Roytberg, P. A. Pevzner, and M. S. Gelfand. Performance-guarantee gene prediction via spliced alignment. *Genomics*, 51:332-339, 1998.

[18] National Center for Biotechnology Information Database of Expressed Sequence Tags.
http://www.ncbi.nlm.nih.gov/dbEST/index.html, 1998.

[19] Non-Redundant Protein Sequence Database.
http://bmbsgi11.leeds.ac.uk/bmb5dp/owl.html, 1998.

[20] L. Pachter. Domino tiling, Gene recognition, and Mice. *Ph.D. dissertation, MIT, Department of Mathematics*, 1999.

[21] L. Pachter, S. Batzoglou, E. Banks, W. Beebee, N. Feamster, V. I. Spitkovsky, T. Tyan, B. Wallis, E. S. Lander, D. J. Kleitman, and B. A. Berger. Unpublished manuscript. 1998.

[22] L. Pachter, S. Batzoglou, V. I. Spitkovsky, E. Banks, E. S. Langer, D. J. Kleitman, and B. A. Berger. " A Dictionary Based Approach for Gene Annotation." *Journal of Computational Biology* (Fall 1999), in press.

[23] L. Pachter, S. Batzoglou, V. I. Spitkovsky, W. Beebee, E. S. Langer, B. A. Berger, and D. J. Kleitman. "A Dictionary Based Approach for Gene Annotation." In *Proceedings of the Third Annual International Conference on Computational Molecular Biology*, Lyon, France, April 1999 (NY:ACM Press, 1999), 285-294.

[24] RepeatMasker.
http://ftp.genome.washington.edu/cgi-bin/RepeatMasker, 1998.

[25] I. Rigoutsos and A. Califano. Searching in parallel for similar strings. *Computational Science and Engineering*, 1(2):60-75, 1994.

[26] V. V. Solovyev, A. A. Salamov, and C. B. Lawrence. Identification of human gene structure using linear discriminant functions and dynamic programming. *Proceedings of ISMB*, pages 367-375, 1995.

[27] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195-197, 1981.

[28] S. Sze and P. A. Pevzner. Las Vegas algorithms for gene recognition: Suboptimal and error-tolerant spliced alignment. *Journal of Computational Biology*, 4(3):297-309, 1997.

[29] M. S. Waterman. *Introduction to Computational Biology: Maps, sequences and genomes*. Chapman and Hall, 1998.

[30] Y. Xu, R. J. Mural, M. Shah, and E. C. Uberbacher. Recognizing exons in genomic sequences using GRAIL II. In J. Setlow, editor, *Genetic Engineering: Principles and Methods*, volume 16. Penum Press, 1994.

[31] Z. Zhang, W. R. Pearson, and W. Miller. Aligning a DNA sequence with a protein sequence. *Journal of Computational Biology*, 4(3):339-349, 1997.

[32] zlib 1.1.3
`http://www.info-zip.org/pub/infozip/zlib/`, 2000.