New Jersey Institute of Technology

Digital Commons @ NJIT

Dissertations

Electronic Theses and Dissertations

Summer 8-31-2002

Knowledge discovery and modeling in genomic databases

Michael M. Yin
New Jersey Institute of Technology

Follow this and additional works at: https://digitalcommons.njit.edu/dissertations

Part of the Databases and Information Systems Commons, and the Management Information Systems Commons

Recommended Citation

Yin, Michael M., "Knowledge discovery and modeling in genomic databases" (2002). *Dissertations*. 551.
https://digitalcommons.njit.edu/dissertations/551

This Dissertation is brought to you for free and open access by the Electronic Theses and Dissertations at Digital Commons @ NJIT. It has been accepted for inclusion in Dissertations by an authorized administrator of Digital Commons @ NJIT. For more information, please contact digitalcommons@njit.edu.

Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be "used for any purpose other than private study, scholarship, or research." If a, user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of "fair use" that user may be liable for copyright infringement,

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation

Printing note: If you do not wish to print this page, then select "Pages from: first page # to: last page #" on the print dialog screen

NJIT
New Jersey's Science & Technology University

The Van Houten library has removed some of the personal information and all signatures from the approval page and biographical sketches of theses and dissertations in order to protect the identity of NJIT graduates and faculty.

ABSTRACT

KNOWLEDGE DISCOVERY AND MODELING IN GENOMIC DATABASES

by
Michael M. Yin

This dissertation research is targeted toward developing effective and accurate methods for identifying gene structures in the genomes of high eukaryotes, such as vertebrate organisms. Several effective hidden Markov models (HMMs) are developed to represent the consensus and degeneracy features of the functional sites including protein-translation start sites, mRNA splicing junction donor and acceptor sites in vertebrate genes. The HMM system based on the developed models is fully trained using an expectation maximization (EM) algorithm and the system performance is evaluated using a 10-way cross-validation method. Experimental results show that the proposed HMM system achieves high sensitivity and specificity in detecting the functional sites.

This HMM system is then incorporated into a new gene detection system, called GeneScout. The main hypothesis is that, given a vertebrate genomic DNA sequence S, it is always possible to construct a directed acyclic graph G such that the path for the actual coding region of S is in the set of all paths on G. Thus, the gene detection problem is reduced to the analysis of paths in the graph G. A dynamic programming algorithm is employed by GeneScout to find the optimal path in G. Experimental results on the standard test dataset collected by Burset and Guigo indicate that GeneScout is comparable to existing gene discovery tools and complements the widely used GenScan system.

KNOWLEDGE DISCOVERY AND MODELING IN GENOMIC DATABASES

by
Michael M. Yin

A Dissertation
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy in Computer and Information Science

Department of Computer Science

August 2002

Copyright © 2002 by Michael M. Yin

ALL RIGHTS RESERVED

APPROVAL PAGE

KNOWLEDGE DISCOVERY AND MODELING IN GENOMIC DATABASES

Michael M. Yin

Dr. Jason T. L. Wang, Dissertation Advisor — Date
Professor of Computer Science, NJIT

Dr. James A. McHugh, Committee Member — Date
Professor of Computer Science, NJIT

Dr. Frank Shih, Committee Member — Date
Professor of Computer Science, NJIT

Dr. Vincent Oria, Committee Member — Date
Assistant Professor of Computer Science, NJIT

Dr. Xiaoan Ruan, Committee Member — Date
Senior Scientist in Molecular Biology, Abbott Laboratories, Inc

BIOGRAPHICAL SKETCH

Author:	Michael M. Yin
Degree:	Doctor of Philosophy
Date:	August 2002

Undergraduate and Graduate Education:

- Doctor of Philosophy in Computer and Information Science
  New Jersey Institute of Technology, Newark, NJ, 2002
- Master of Science in Computer Science
  New Jersey Institute of Technology, Newark, NJ, 1997
- Master of Science in Microbiology
  Huazhong Agriculture University, Wuhan, China, 1985
- Bachelor of Science in Microbiology
  Huazhong Agriculture University, Wuhan, China, 1982

Major: Computer Science

Publications:

M. M. Yin and J. T. L. Wang,
"GeneScout: a data mining system for predicting vertebrate genes in genomic DNA sequences", To appear in *Information Sciences*, Special Edition of Soft Computing Data Mining, 2002.

M. M. Yin and J. T. L. Wang,
"Effective hidden Markov models for detecting splicing junction sites in DNA sequences", *Information Sciences*, 139(1-2):139–163, 2001.

M. M. Yin and J. T. L. Wang,
"Application of hidden Markov models to biological data mining: a case study", *Data Mining and Knowledge Discovery: Theory, Tools, and Technology II. Proceedings of SPIE*, B. V. Dasarathy (ed), Vol. 4057, SPIE-The International Society for Optical Engineering, USA, pages 352–358, 2000.

M. M. Yin and J. T. L. Wang,
"Application of hidden Markov models to gene prediction in DNA", *Proceedings of the IEEE International Conference on Information, Intelligence and Systems*, pages 40–47, Bethesda, Maryland, November 1999.

M. M. Yin and J. T. L. Wang,
"Recognizing splicing junction acceptors in Eukaryotic genes using hidden Markov models and machine learning methods", *Proceedings of the Fifth Joint Conference on Information Sciences*, P. P. Wang (ed), Vol. 2, The Association for Intelligent Machinery, pages 786–789, Atlantic City, New Jersey, February, 2000.

J. T. L. Wang, S. Rozen, B. A. Shapiro, D. Shasha, Z. Wang, M. M. Yin
"New techniques for DNA sequence classification", *Journal of Computational Biology*, 6(2):209–218, 1999.

M. M. Yin and J. T. L. Wang,
"Algorithms for splicing junction donor recognition in genomic DNA sequences", *Proceedings of the IEEE International Joint Symposia on Intelligence and Systems*, pages 169–176, Rockville, Maryland, May 1998.

M. M. Yin,
"Algorithms and tools for splicing junctions donor recognition in genomic DNA sequences", *Master of Science Thesis*, New Jersey Institute of Technology, Newark, NJ, October, 1997.

This Dissertation is dedicated
to my wonderful wife, Yuxian,
my children Andy and Sally for
their love, support and
patience while I worked nights and weekends
on this project instead of spending time with them.

ACKNOWLEDGEMENT

The author wishes to express his sincere gratitude to his advisor, Professor Jason T. L. Wang, for his guidance and support throughout this research.

The author is grateful to Dr. James A. McHugh, Dr. Frank Shih, Dr. Vincent Oria, Dr. Byount-Kee Yi and Dr. Xiaoan Ruan for serving as members of the dissertation committee.

Special thanks go to Dr. Robert Mitchell for his precious encouragement and support to this and other related research when I worked in his laboratory for about seven years.

Finally, the author wants to thank his wife, Yuxian; without her love and support this accomplishment would not be possible.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Biological Background

A DNA (deoxyribonucleic acid) chain is a long, unbranched polymer composed of four types of nucleotides or bases: adenine (A), cytosine (C), guanine (G) and thymine (T). Genes, made of deoxyribonucleic acid, are the invisible information-containing elements that are distributed to each daughter cell when a cell divides. In general, genes are divided into two categories: *eukaryotic* and *prokaryotic.* Eukaryotic genes are from *eukaryotic cells* and prokaryotic genes are from *prokaryotic cells.* "Eu" means "good, well or true". "Karyote" (or "caryote") means "nucleus" ("caryon" in Greek). Eukaryotic cells, by definition, have a nucleus that contains the cell's DNA for all of the genes, enclosed by a double layer of membrane [1]. So, the eukaryotic gene category includes all kinds of genes from cells with a nucleus, such as those from any kind of animals, even yeast.

In the bioinformatics field, eukaryotic DNA (or gene) means the kind of genomic DNA with *introns*, such as the DNA from high level animals and human. Prokaryotic cells, in contrast to eukaryotic cells, have relatively simple internal structures, specifically, without membrane enclosed nuclei [1]. Prokaryotic cells include the various types of bacteria such as *E. coli. E. coli* has simple genomic DNA and its cells are very easy to culture. So, *E. coli* is often used for research on prokaryotic DNA.

The basic gene structure for higher eukaryotes includes promoter, start codon, introns, exons, and stop codon, etc. (see Figure 1.1). The exon sequences of a gene are called the coding sequences of this gene, and the region covers all the exon sequences of a gene are called the coding region of the gene (which is the region for making protein). In contrast, prokaryotic genes have no introns, and the gene structure includes only promoter, start codon, coding region and stop codon.

Normally, if one can detect the promoter in the prokaryotic DNA sequences, one is able to find the gene coding region. Intron sequences range in size from about 80 nucleotides to 10,000 nucleotides or more. Introns in genes are of no function at all and are actually the genetic "junk" [1]. They differ dramatically from exons in that their exact nucleotide sequences seem to be unimportant. The only highly conserved sequences in introns are those required for intron removal.

The genetic information present in genes is expressed in the organisms (*Gene expression*) through the processes of *Transcription* and *Translation* (see Figure 1.1). Transcription is the process for the production of a specific molecular of messenger RNA (mRNA) from a given sequence of DNA in a gene. In this process the genetic information (message) carried in the DNA is transcribed to (or written into) the mRNA. As its name implies, messenger RNA carries a message. mRNA transmits the genetic message in the sequence of its own bases. The process by which mRNA directs the synthesis of a specific protein is called translation. In this process, the information (message) carried in the base sequences of the mRNA is translated into the amino acid sequence of the protein.

In the eukaryotic gene transcription process, the intermediate product is called pre-mRNA. Pre-mRNA is the direct copy of the DNA sequences in the eukaryotic gene and it contains the exon and intron sequences from the gene. The intron sequences will be removed from pre-mRNA, so a mature mRNA only consists of exon sequences, which will be translated into protein. The process for intron removal is called RNA splicing, and the positions for intron removal and RNA rejoining are called splicing junction sites (see Figure 1.1). The consensus sequences at each end of an intron are nearly the same in all known intron sequences, and these can not be changed without affecting the splicing process. The conserved boundary sequence at the 5' splice site is called a *donor site*, and the one at the 3' splice site is called an *acceptor site*. The RNA breaking and rejoining (splicing) must be carried out

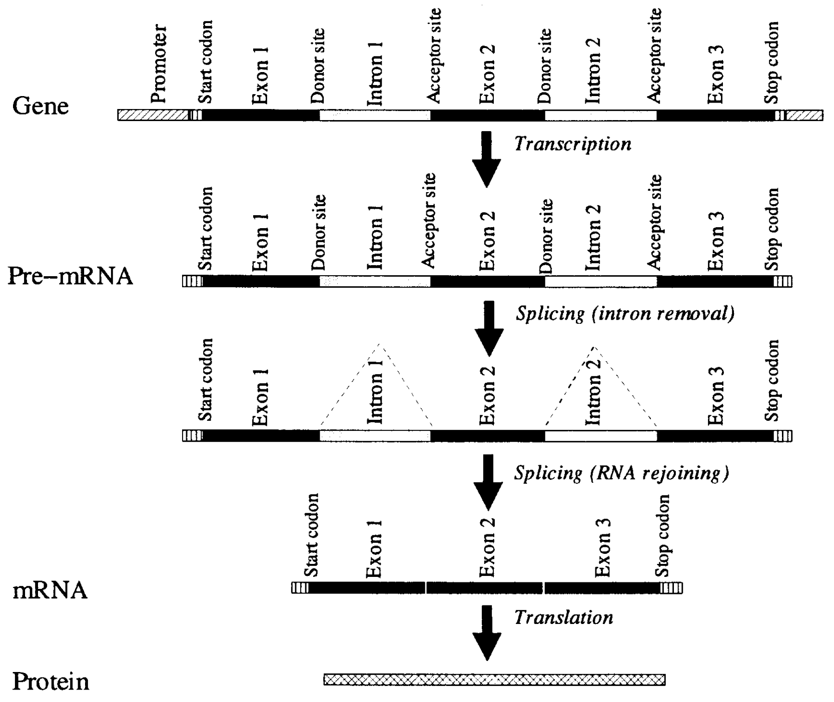


Figure 1.1 Eukaryotic gene structure and gene expression processes.

The basic gene structure for higher eukaryotes includes promoter, start codon, exons, introns and stop codon, etc. The boundaries between the exons and the introns are called 5' donor sites, and the boundaries between the introns and the exons are called 3' acceptor sites. During the DNA transcription process, the gene sequences (excluding the promoter region) are first transcribed into pre-mRNA. Then, the intron sequences in the pre-mRNA are removed and the RNA fragments are rejoined together by the RNA splicing process to get mRNA. The mRAN directs protein synthesis through the gene translation process.

precisely because an error of even one nucleotide would shift the reading frame in the resulting mRNA molecule and make nonsense of its message [1].

Identification or prediction of coding sequences from within genomic DNA has been a major rate-limiting step in the pursuit of genes. For eukaryotic gene detection, researchers have to detect the start codon, exons, introns and stop codon. How to find out exons/introns? The most important step is to detect the splicing junction sites including donor and acceptor sites, because once the splicing junction sites are detected, the exon/intron boundaries are found. Then the introns can be removed from the DNA sequence to get the coding regions. Biologists study gene structures based on lab experiments such as PCR on cDNA libraries, Northern blot, sequencing, etc. However, characterizing the 60,000 to 100,000 genes thought to be hidden in the human genome by means of merely experiments is not feasible. A current trend is to complement the lab study with bioinformatics approaches. using computer programs to elucidate a gene structure from DNA sequence signals, including start codon, splicing junction donor sites and acceptor sites, stop codon, etc.

1.2 Current Status and Progress

Although methods to predict potential gene coding regions on genomic DNA sequences have existed since the 1980s, the first programs to assemble potential DAN coding sequences into translatable mRNA sequences were not available until the early 1990s [7]. Recently there are several programs available for biologists, such as GeneID [14], GenLang [10] and GRAIL [43], etc. GRAIL is the one now widely used by researchers and it is available on the BLAST web site (http://www.ncbi. nlm.nih.gov) for gene structure detection. The approaches used for the function sites detection include:

- **Consensus Search [12]** This approach considers an aligned set of site sequences. At each position with non-uniform distribution of nucleotides, researchers retain the preferred nucleotide and obtain the *consensus* word. It is possible to account for degeneracies and to distinguish between strongly and weakly conserved positions, dependent on the degree of the non-uniformity. People write the consensus of mammalian gene donor sites as 'maG/**GT**RAGu', where the boldface denotes invariant positions, capital and lower case letters denote, respectively, strongly and weakly conserved positions. 'R' denotes 'A or G' and 'm' denotes 'a or c'. '/' is the splice point. A formal determination of conserved positions can be made using standard statistical criteria or computation of the information content of positional nucleotide distribution [17, 27, 31]. The consensus methods are tools to summarize the distribution of an aligned set of molecular sequences. Typically the methods make three simplifying assumptions:

1. Analysis of molecular sequences is a multistage process in which sequence alignment precedes the identification of consensus sequences.
2. An alignment of the molecular sequences has already been obtained.
3. Alignment and the identification of consensus sequences can be treated independently.

Thus, the problem to find a consensus sequences of k aligned molecular sequences, in which n aligned positions have been identified, can be viewed as a set of n simpler problems, each to find a consensus of k symbols (*i.e.* nucleotides) at an aligned position [9]. The comparison with the consensus is the simplest form of the site prediction algorithm, but consensus analysis only does a very rough functional mapping of a sequence and its results should be interpreted with caution [12].

- **Weight Matrices** The next level of sophistication is provided by weight matrices. Each nucleotide b ($b =$ A, C, G, T) in the site position p ($p = 1, 2, \ldots,$

L) is set in correspondence with the weight W $(b,\ p)$. The score of a potential site is defined as the sum of the position weights of the constituent nucleotides. R. Staden applied the weight matrix method to obtain the relative importance of each nucleotide in the consensus sequence [34]. Another approach used for multivariate statistical analysis was to perform categorical discriminant analysis, where nucleotide sequences were transformed into categorical data. Categorical weights on the variables were estimated in such a way that the two classes of the 5' splice site sequences and sequences other than the 5' splice site might be discriminated most distinctly [22]. It has been demonstrated that site strengths estimated by this theory to some extent agree with the experimental data [12]. Like consensus search, the weight matrices can be used for fast database searches.

- **Pattern Recognition and Neural Networks [12]** Algorithms of the pattern recognition theory are based on the (implicit) assumption that in the genome there is a tendency to avoid non-functioning signal-like sites. Thus, a learning sample consists of two classes, sites and non-sites. The non-sites class is usually formed by random fragments of the natural DNA. The basic steps in application of pattern recognition techniques are as follows:

  1. Creation of a learning sample.

  2. Choice and encoding of signal features.

  3. Iterative correction of recognition rules according to results of discrimination between the two classes at the previous round.

  4. Testing on an independent sample.

  One of the diverse pattern recognition algorithms is *neural networks* [43]. The neural networks consist of a layer of input neuron, several layers of hidden neurons, and an output neuron. When the network is presented with a candidate site, the input neurons check whether the site possesses the corresponding features and send binary signals to the neurons of the first hidden layer. Each hidden neurons sums the weighted signals coming by connections from the lower level, compares the result with the threshold, and sends a binary signal to the upper level neurons. The output neuron provides the final site/non-site detection. Programs such as GenViewer [26] and GRAIL [43] employ a procedure that scores candidate exons using some combination of the sites scores and the coding potential. They then perform an exhaustive search over the set of structures generated by the remaining high-scoring exons.

Recently, Moises Burset and Roderic Guigo evaluated a number of computer programs designed to predict the structure of gene coding regions in genomic DNA sequences [7]. The programs analyzed were uniformly tested on a large set of vertebrate sequences with a standard gene structure. Their carefully selected test set included 570 sequences, totaling 2649 coding exons. All the sequences in the test set had the start codon and stop codon. All the donor sites contain the **GT** dinucleotide and all the acceptor sites contain the **AG** dinucleotide at the right positions. Some of their data was shown in Table 1.1. The results indicated that the predictive accuracy of the programs analyzed was really low. For example, for the widely used GRAIL program, the sensitivity (Sn) and specificity (Sp) were just 36% and 43% [7]. So they claimed that although programs currently available may still be of great use in pinpointing the regions likely to contain exons, they are far from being powerful enough to elucidate its genomic structure completely [7].

The vertebrate DNA sequence signals involved in gene determination are usually ill defined, degenerate and highly unspecific. Given the current detection methods, it is usually impossible to distinguish the signals truly processed by the cellular machinery from those that are apparently non-functional [13]. Furthermore, the inherent conservatism of the currently popular methods such as similarity

Table 1.1 Performance of the Programs Evaluated by M. Burset and R. Guigo

Programs	*Sensitivity*	*Specificity*
FGENEH	0.61	0.64
GeneID+	0.73	0.70
GeneParser3	0.56	0.58
GeneLang	0.51	0.52
GRAIL2	0.36	0.43
SORFIND	0.42	0.47
Xpound	0.15	0.18

Note: M. Burset and R. Guigo defined Sensitivity (Sn) and Specificity (Sp) as follows:

$$S_n = \frac{Number\ of\ Correct\ Exons}{Number\ of\ Actual\ Exons} \tag{1.1}$$

$$S_p = \frac{Number\ of\ Correct\ Exons}{Number\ of\ Predicted\ Exons} \tag{1.2}$$

search, GRAIL, etc. will greatly limit the capacity for making unexpected biological discoveries from increasingly abundant genomic data. Except for a very limited subset of trivial cases, the automated interpretation without experimental validation of genomic data is still a myth [8]. Unlike the situation in bacteria and yeast organisms, in which computer systems have substantially contributed to the automatic analysis of genomes, automatic sequence analysis and annotomatic elucidation of their structure from the genomes of high eukaryotic organisms are far from being a reality [13].

This research is targeted toward developing effective and accurate methods for identifying gene structures in the genomes of high eukaryotic organisms. The first phase of the research is for splicing junction sites detection. Then, during the second phase, the gene structure signal information will be integrated with global gene structure information together to develop a full gene structure detection system.

Splicing junction donor and acceptor sites are the most important functional gene structure signals. Earlier, a donor Motif model was developed and pattern matching techniques were used for donor classification [40, 44, 45]. The case studies and preliminary data for splicing junction donor and acceptor site classifications were also reported [46, 47, 48]. Here, the approaches using hidden Markov modes (HMM) will be introduced to represent the degeneracy features of splicing junction sites. Then the 10-way cross-validation method is used to evaluate the system for splicing junction sites detection in unlabeled test DNA sequences.

Hidden Markov models (HMMs) have been used extensively to describe sequential data or processes such as speech recognition. An HMM model is a process in which some of the details are unknown, or *hidden*. A general description of a Markov model is that it models a stochastic process using a number of states and probabilistic state transitions. An HMM is defined by a set of states and transitions, usually represented by a graph where states correspond to vertices and transitions to edges. Each state s is associated with a discrete output probability distribution, $P(s)$. Similarly, each transition has a probability, which represents the probability that a generating process makes that transition. Thus, the sum of the probabilities of all the transitions from a given states s to all other states must be 1. Hidden Markov models have been remarkably successful in the field of speech recognition [21], where they are used in most state-of-the-art systems. Researchers in computational biology have recently started to use HMMs for biological sequence analysis. Lukashin, Borodovsky [23] and their colleagues [5] successfully applied HMMs to the detection of protein coding regions in prokaryotes. Audic and Claverie [2] reported their using of Markov transition matrices to detect eukaryotic promoters. Salzberg [29] has used HMM for identifying splice sites and translational start sites in eukaryotic genes. Salzberg's group also developed an HMM system, called VEIL (Viterbi Exon-Intron Locator), for finding eukaryotic genes [19]. The approach used in this study differs from Salzberg's by using a different topology of HMM and by employing two modules in the HMM model: one for true sites, and the other for false sites.

There are also many other pioneers in this field. Even though the current systems are far from being powerful enough for gene structure elucidation, the information these researchers provide is valuable, and research on automated gene detection using HMM is of great potentiality.

CHAPTER 2

SPLICING JUNCTION SITES MODELING AND DETECTING

2.1 Using HMMs to Model Splicing Junction Sites

2.1.1 The Donor Model

Splicing junction sites in vertebrate DNA include donor and acceptor sites. Donor sites are conserved boundary sequences at the 5' splicing sites in DNA. The conserved sequences include nine nucleotide bases with GT (GU in mRNA) almost invariable to all donor sites [1]. An example of a donor site is shown below:

$$\overbrace{\cdots CAG}^{exon}\overbrace{\mathbf{GT}GAGA\cdots}^{intron}$$

The nucleotide G occurs at position 4 and the nucleotide T occurs at position 5 in a donor site. Here refer to a 9-base sequence that exists as a donor in a real gene sequence as a *true donor site.* Note that in all true donor sites, G and T occur at position 4 and position 5, respectively. Similarly, refer to a 9-base non-donor sequence in which G and T also occur at position 4 and position 5, respectively, as a *false donor site.* Notice that it is not necessary to consider those sequences without G, T being at position 4 and position 5, respectively, because they are deemed to be non-donor sequences. Given an unlabeled 9-base sequence with G, T being at position 4 and position 5, respectively, referred to as a *candidate donor site*, the algorithm tries to determine whether the candidate sequence is a true donor site or a false donor site. A Donor Model is designed based on HMMs to describe the consensus and degenerate properties occurring in true donor sites.

An HMM with nine states and a set of transitions is used for modeling a true donor site, which is represented as a digraph where states correspond to vertices and transitions to edges. At each state, the HMM generates a base b in {A, G, C, T} according to the state and transition probabilities, with the exception of state 4 and state 5. At state 4, the HMM constantly generates base b = G, and at state 5, the

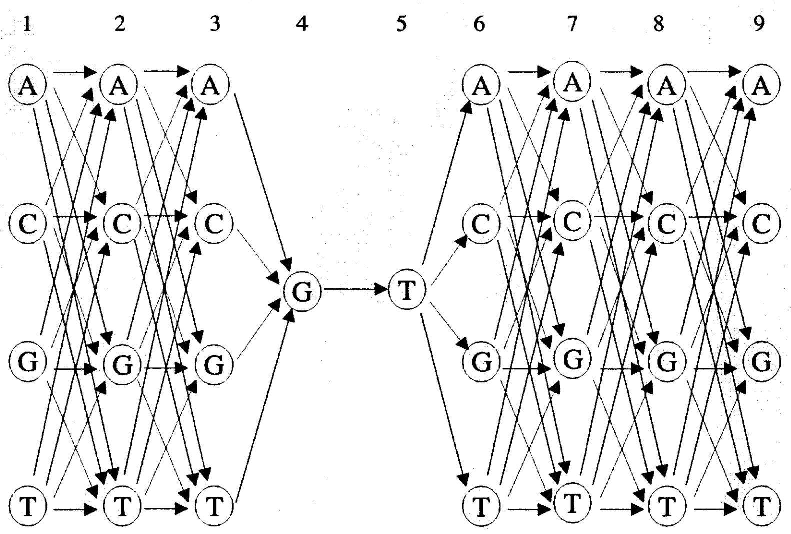


Figure 2.1 The Donor Model for splicing junction donor site detection.

HMM constantly generates base b = T. Each state s is associated with a discrete probability distribution, $P(s)$. For state 4 and state 5, $P(s) = 1$. Except at state 3 and state 4, each base b at a state has four possible transitions to the next state. Each transition has a probability, $P(t)$, which represents the probability that the HMM makes that transition. Each base at state 3 has a fixed transition, namely $P(t) = 1$, to the base G at state 4. Similarly, at state 4, the base G has a fixed transition, namely $P(t) = 1$, to the base T at state 5. Figure 2.1 illustrates the Donor Model.

2.1.2 The Acceptor Model

Acceptor sites are conserved boundary sequences at the 3' splicing sites in DNA. The conserved sequences include 16 nucleotide bases with AG almost invariable to all acceptor sites [1]. An example of an acceptor site is shown below:

$$\overbrace{\cdots CTATCCTTCTCAC\mathbf{AG}}^{intron}\overbrace{G\cdots}^{exon}$$

The nucleotide A occurs at position 14 and the nucleotide G occurs at position 15 in an acceptor site. Refer to a 16-base sequence that exists as an acceptor in a real gene sequence as a *true acceptor site*. Note that in all true acceptor sites, A and G occur at position 14 and position 15, respectively. Similarly, refer to a 16-base non-acceptor sequence in which A and G also occur at position 14 and position 15, respectively, as a *false acceptor site*. Given an unlabeled 16-base sequence with A, G being at position 14 and position 15, respectively, referred to as a *candidate acceptor site*, the algorithm tries to determine whether the candidate sequence is a true acceptor site or a false acceptor site. The Acceptor Model, defined below, is used to describe the consensus and degenerate properties occurring in true acceptor sites.

An HMM with 16 states and a set of transitions is developed for modeling a true acceptor site, which is represented as a digraph where states correspond to vertices and transitions to edges. At each state, the HMM generates a base b in {A, G, C, T} according to the state and transition probabilities, with the exception of state 14 and state 15. At state 14, the HMM constantly generates base b = A, and at state 15, the HMM constantly generates base b = G. Each state s is associated with a discrete probability distribution, $P(s)$. For state 14 and state 15, $P(s) = 1$. Except at state 13 and state 14, each base b at a state has four possible transitions to the next state. Each transition has a probability, $P(t)$, which represents the probability that the HMM makes that transition. Each base at state 13 has a fixed transition, namely $P(t) = 1$, to the base A at state 14. Similarly, at state 14, the base A has a fixed transition, namely $P(t) = 1$, to the base G at state 15. Figure 2.2 illustrates the Acceptor Model. There are 16 states in this model. Except state 14 and state 15, there are four possible bases at each state, and a base at one state may have four possible ways to transit to the next state. States 14 and 15 are a constant, and the transition from state 14 to state 15 is also a constant with a probability of 1. In a

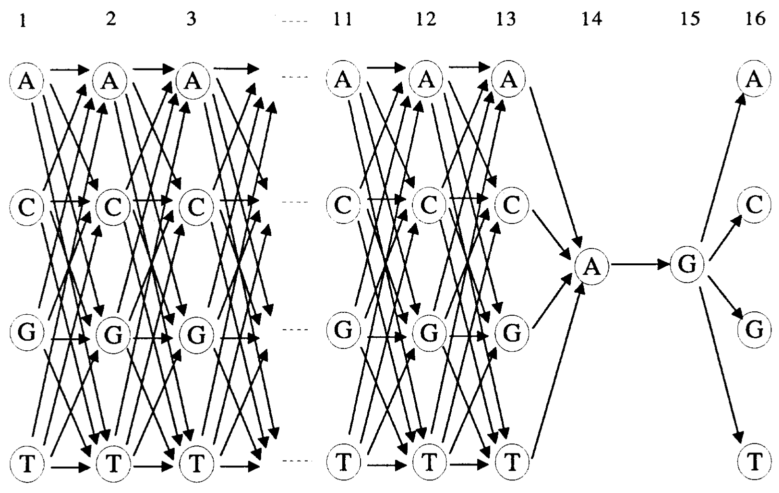


Figure 2.2 The Acceptor Model for splicing junction acceptor site detection.

gene sequence, states 1 through 15 belong to an intron and state 16 is the first base of an exon.

2.1.3 Two Modules for Each Model

In vertebrate DNA sequences, there are much more false splicing junction sites than true sites. The ratio between the number of false sites and the number of true sites is about 100 to 1. In order to mine out the differences between the true sites and false sites, two programs are implemented: True Donor Module and False Donor Module, based on the Donor Model and another two programs, True Acceptor Module and False Acceptor Module, based on the Acceptor Model. The True Donor Module and True Acceptor Module are collectively referred to as *true site modules.* The False Donor Module and False Acceptor Module are collectively referred to as *false site modules.* The true site modules are trained using the true sites in the training data set, and train the false site modules using the false sites in the training data set. A given candidate site is tested by these modules. Let S_{cand} be a candidate site. Let $P(Y = 1|S_{cand}, M^{(t)})$ be the probability of S_{cand} being a donor (acceptor) sequence given that it is processed by a true site module. Let $P(Y = 0|S_{cand}, M^{(f)})$ be the probability of S_{cand} being a non-donor (non-acceptor) sequence given that it is processed by a false site module. In the above specification, $M^{(t)}$ is for the true site modules and $M^{(f)}$ is for the false site modules. In the splicing junction detection phase, these true site modules and false site modules are used to classify candidate sequences into right categories. For example, for a candidate donor site, it is first passed through True Donor Module to get $P(Y = 1|S_{cand}, M^{(t)})$, the probability of this candidate site being a donor sequence. It is then passed through False Donor Module to get $P(Y = 0|S_{cand}, M^{(f)})$, the probability of this candidate site being a non-donor sequence. Comparing these two values, a score is assigned to the candidate sequence. This candidate sequence is assigned to the true donor category or false donor category depending on its score obtained.

2.2 Algorithms

The algorithms described in this section can be used for both the Donor Model and the Acceptor Model. For illustration purposes, this section focus on Donor Model and its corresponding modules, True Donor Module and False Donor Module. The algorithms for the Acceptor Model are essentially the same.

2.2.1 Training Algorithm

A modified expectation maximization (EM) algorithm, called **TEM**, is developed for training the modules. The original EM method takes, as the input, a set of unaligned sequences and a motif length, and returns a probabilistic model for the motif [3]. Because the data set contains splicing junction sites with the same length, and all these sites can be aligned to each other, **TEM** is designed specifically for training a hidden Markov model with fixed topology.

Let M represent the set of sequences that are randomly picked from the positive training data set and negative training data set. (In the study presented here, M contains about 200 true donor sites and 14,000 false donor sites.) Each sequence in M is labeled as *positive* or *negative* depending on whether it is from the positive training data set or the negative training data set. Let E^t be the set containing the remaining sequences in the positive training data set, and let E^f represent the set containing the remaining sequences in the negative training data set. There are much more true (false, respectively) donor sites in E^t (E^f, respectively) than those in M. (In this study presented here, the total number of the sequences in E^t and E^f is about 9 times of the number of sequences in M.) Let P be a subset of M.

In the training phase, the **TEM** algorithm proceeds iteratively to converge. At each iteration, the algorithm removes some sequences from E^t and E^f and inputs those sequences into True Donor Module and False Donor Module. The algorithm then uses these modules to determine which sequences are placed in P as it will be explained later. S_n^{em} is used to represent the *sensitivity* and S_p^{em} is used to represent the *specificity* during the **TEM** training. S_n^{em} is the ratio between the number of true donor sites in P and the total number of true donor sites in M; note that $P \subseteq M$. S_p^{em} is the ratio between the number of true donor sites in P and the total number of sequences in P. The goal of the **TEM** training is, given a fixed value of S_n^{em}, the modules are trained iteratively to get a maximal value of S_p^{em}, or until E^t and E^f become empty. In this research, $S_n^{em} = 0.90$ is used for training the modules.

Specifically, let T_{states} represent the total number of states in the Donor Model. Let b_i ($b_i \in \{A, G, C, T\}$) be the base at state i, $1 \leq i \leq T_{states}$. Let $tr_i(b_i, b_{i+1}), 1 \leq i \leq T_{states} - 1$, be the transition from state i to state $i+1$. The topology for the Donor Model is fixed, and all of the transition probabilities and state probabilities are initialized to random values. Then, one tenth of the sequences are picked from E^t and they are inputted into the True Donor Module. At the same time, one tenth of the sequences from E^f are picked and they are inputted into False Donor Module. The number of the individual bases, b_i, is recorded at each state and the number of individual transitions, $tr_i(b_i, b_{i+1})$, is also recorded from one state to the next state. Then compute the post probabilities for all the states and transitions in True Donor Module and False Donor Module are computed. Let $T^{(t)}tr_i(b_i, b_{i+1})$ be the total number of transitions from a base b_i at state i to a base b_{i+1} at state i +1 in True Donor Module. Let $T_{in}^{(t)}$ be the total number of true donor sites that have been input into True Donor Module. The state transition probabilities, $ftr_i^{(t)}(b_i, b_{i+1})$, in True Donor Module can be calculated as follows:

$$ftr_i^{(t)}(b_i, b_{i+1}) = \frac{T^{(t)}tr_i(b_i, b_{i+1})}{T_{in}^{(t)}} \tag{2.1}$$

Similarly, let $T^{(f)}tr_i(b_i, b_{i+1})$ be the total number of transitions from a base b_i at state i to a base b_{i+1} at state i +1 in False Donor Module. Let $T_{in}^{(f)}$ be the total number of false donor sites that have been input into False Donor Module. The state transition probabilities, $ftr_i^{(f)}(b_i, b_{i+1})$, in False Donor Module can be calculated as follows:

$$ftr_i^{(f)}(b_i, b_{i+1}) = \frac{T^{(f)}tr_i(b_i, b_{i+1})}{T_{in}^{(f)}} \tag{2.2}$$

Next, all the sequences in M are treated as unlabeled sequences and input them into True Donor Module and False Donor Module. Let $P(True \mid S,\ M^{(t)})$ denote the probability of a sequence S in set M being a donor sequence, and let $P(False \mid S,\ M^{(f)})$ denote the probability of S being a non-donor sequence. In order to calculate $P(True \mid S,\ M^{(t)})$, the probability of the sequence S must be calculated given it is a true donor site using True Donor Module. This can be written as

$$P(S \mid True,\ M^{(t)}) = \prod_{i=1}^{T_{States}-1} ftr_i^{(t)}(b_i, b_{i+1}),\ \ b_i \in \{A, G, C, T\} \tag{2.3}$$

The **TEM** algorithm uses Bayes' rule to estimate $P(True \mid S,\ M^{(t)})$ from $P(S \mid True,\ M^{(t)})$. Bayes' rule states that

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \tag{2.4}$$

so,

$$P(True \mid S,\ M^{(t)}) = \frac{P(S \mid True,\ M^{(t)})P(True)}{P(S)} \tag{2.5}$$

$P(True)$ is the prior probability that is assumed to be a constant, and $P(S)$ is the product of the individual base probabilities in the sequences. $P(S)$ can be written as

$$P(S) = \prod_{i=1}^{T_{States}} P(b_i \mid True,\ M^{(t)}) \tag{2.6}$$

In the same way, equations can be written for calculating $P(False \mid S,\ M^{(f)})$ as follows:

$$P(S \mid False,\ M^{(f)}) = \prod_{i=1}^{T_{States}-1} ftr_i^{(f)}(b_i, b_{i+1}),\ b_i \in \{A, G, C, T\} \tag{2.7}$$

$$P(False \mid S,\ M^{(f)}) = \frac{P(S \mid False,\ M^{(f)})P(False)}{P(S)} \tag{2.8}$$

$$P(S) = \prod_{i=1}^{T_{States}} P(b_i \mid False,\ M^{(f)}) \tag{2.9}$$

Let *pratio* be the probability ratio of sequence S in set M, and

$$pratio = \frac{P(True \mid S,\ M^{(t)})}{P(False \mid S,\ M^{(f)})} \tag{2.10}$$

The *pratio* is calculated for each sequence in set M. Then the sequences in set M is sorted, in the descending order, according to their *pratio* values. Suppose the total number of positive sequences in set M is N. Then select the *pratio* value for the $N \times S_n^{em}$th positive sequence and use that *pratio* value as the positive lower bound, denoted L_p. (In the study presented here, there are 200 positive sequences in set M and the sensitivity S_n^{em} is 0.9. Therefore, L_p is the *pratio* value of the 180th positive sequence in set M.) The **TEM** algorithm assigns a sequence $S \in M$ into set P if the *pratio* value for S is greater than or equal to L_p. Let $T_{(TP)}$ be the number of positive sequences in set M that are assigned into set P. Let $T_{(P+N)}$ be the total number of positive sequences in M. Then, by definition,

$$S_n^{em} = \frac{T_{(TP)}}{T_{(P+N)}} \tag{2.11}$$

Let $T_{(PP)}$ be the total number of sequences in M that are assigned into P. Then, by definition,

$$S_p^{em} = \frac{T_{(TP)}}{T_{(PP)}} \tag{2.12}$$

The re-estimation procedure then adjusts all of the probabilities hidden in the Donor Model in order to increase S_p^{em}. New sequences in E^t and E^f are picked and removed from E^t and E^f. These sequences are then run through True Donor Module and False Donor Module again and the probabilities are further refined. This process is iterated until the S_p^{em} is maximized or until E^t and E^f become empty. This algorithm is guaranteed to converge to a locally optimal estimate of all the probabilities in the Donor Model. The positive lower bound L_p that maximizes S_p^{em} will be an output and used in the splicing junction sites detection phase. Figure 2.3 summarizes the **TEM** algorithm used in the training phase.

2.2.2 Algorithm for Detecting Splicing Junction Sites

As described in Section 2.1, a candidate donor site refers to a 9-base sequence fragment with bases G, T being at position 4 and position 5, respectively. The input of the site detection algorithm is a fragment, denoted S_{cand}, of nine bases extracted from a genomic DNA sequence S with a minimum length of nine bases. In this research, the longest DNA sequence used is about 50,000 bases long. The S_{cand} has G, T at position 4 and 5, respectively, and is considered as a candidate donor site. The nine bases in S_{cand} are referred as $b_1, b_2, \ldots, b_9$, respectively. The output of the site detection algorithm is a flag, $KIND_i$, indicating whether the S_{cand} starting at positing i of the genomic DNA sequence S is a true donor site or not.

Let $ftr_j^{(t)}(b_j, b_{j+1})$ be the probability of a transition from base b_j to base b_{j+1}, $1 \leq j \leq 8$, of S_{cand} using True Donor Module. Define a flag variable Y to be 1 if S_{cand} belongs to a true site category, and 0 otherwise. Let n be the length of the candidate site S_{cand} (n is 9 for donor sites and 16 for acceptor sites). Let $P(S_{cand}|Y = 1, M^{(t)})$ be the probability of the candidate site S_{cand} given that it is a donor site processed

```
INPUT:
    Untrained HMM site modules
    (including a true site module and a false site module);
    Positive training data set, $E^t$;
    Negative training data set, $E^f$;
    TEM testing data set, $M$;
OUTPUT:
    Fully trained HMM site modules and $L_p$;
ALGORITHM:
    unmaximized := true;
    while unmaximized do begin
        unmaximized := false;
        if $E^t$ is not empty then begin
            remove one tenth of the sequences from $E^t$
            and input them into the true site module;
            for $i = 1$ to $T_{states} - 1$
                calculate $ftr_i^{(t)}(b_i, b_{i+1})$ as in Equation (2.1);
        end;
        if $E^f$ is not empty then begin
            remove one tenth of the sequences from $E^f$
            and input them into the false site module;
            for $i = 1$ to $T_{states} - 1$
                calculate $ftr_i^{(f)}(b_i, b_{i+1})$ as in Equation (2.2);
        end;
        for each sequence $S \in M$ do begin
            calculate $P(True \mid S,\ M^{(t)})$ as in Equation (2.5);
            calculate $P(False \mid S,\ M^{(f)})$ as in Equation (2.8);
            calculate $pratio$ as in Equation (2.10);
        end;
        select $L_p$;
        calculate $S_p^{em}$ according to $L_p$;
        if ($S_p^{em}$ is not maximized) and (either $E^t$ or $E^f$ is non-empty) then
            unmaximized := true;
    end;
```

Figure 2.3 The **TEM** algorithm used in the training phase.

by True Donor Module. Then

$$P(S_{cand}|Y=1, M^{(t)}) = \prod_{j=1}^{n-1} ftr_j^{(t)}(b_j, b_{j+1}), b_j \in \{A, G, C, T\} \tag{2.13}$$

As defined before, $P(Y = 1|S_{cand}, M^{(t)})$ is the probability of S_{cand} being a donor site given that it is processed by True Donor Module. According to Bayes' rule, cf. Equation (2.4):

$$P(Y=1|S_{cand}, M^{(t)}) = \frac{P(S_{cand}|Y=1, M^{(t)})P(Y=1)}{P(S_{cand})} \tag{2.14}$$

When examining a set of sequences to detect true donor sites, the underlying prior $P(Y = 1)$ can be treated as a constant [29]. $P(S_{cand})$ is the product of the individual base probabilities for $b_1, b_2, \ldots, b_n$ in S_{cand}:

$$P(S_{cand}) = \prod_{j=1}^{n} P(b_j|Y=1, M^{(t)}), b_j \in \{A, G, C, T\} \tag{2.15}$$

Similarly, False Donor Module is used to compute $P(Y = 0|S_{cand}, M^{(f)})$, the probability of S_{cand} being a false donor site given that it is processed by False Donor Module. So, the false donor site counterparts of the above equations can be written as:

$$P(S_{cand}|Y=0, M^{(f)}) = \prod_{j=1}^{n-1} ftr_j^{(f)}(b_j, b_{j+1}), b_j \in \{A, G, C, T\} \tag{2.16}$$

$$P(Y=0|S_{cand}, M^{(f)}) = \frac{P(S_{cand}|Y=0, M^{(f)})P(Y=0)}{P(S_{cand})} \tag{2.17}$$

$$P(S_{cand}) = \prod_{j=1}^{n} P(b_j|Y=0, M^{(f)}), b_j \in \{A, G, C, T\} \tag{2.18}$$

Given the candidate donor site S_{cand} starting at position i in the genomic DNA sequence S, the algorithm will find the two most likely sets of states through the two HMM modules for S_{cand}. Then the algorithm calculates $P(Y = 1|S_{cand}, M^{(t)})$ and

```
INPUT:
    A candidate donor site $S_{cand}$ starting at position $i$
    of an unlabeled genomic DNA sequence;
OUTPUT:
    /* $KIND_i$ is a flag indicating whether $S_{cand}$ is a true donor site or not. */
    $KIND_i$;
ALGORITHM:
    present $S_{cand}$ to True Donor Module and calculate
        $P(Y = 1 \mid S_{cand}, M^{(t)})$ as in Equation (2.14);
    present $S_{cand}$ to False Donor Module and calculate
        $P(Y = 0 \mid S_{cand}, M^{(f)})$ as in Equation (2.17);
    calculate $sratio$ as in Equation (2.19);
    calculate $KIND_i$ as in Formula (20);
```

Figure 2.4 Algorithm for classifying splicing junction donor sequences.

$P(Y = 0|S_{cand}, M^{(f)})$. A score, *sratio*, is assigned to the candidate site based on the scoring function below:

$$sratio = \frac{P(Y = 1|S_{cand}, M^{(t)})}{P(Y = 0|S_{cand}, M^{(f)})} \tag{2.19}$$

Comparing *sratio* with the L_p obtained from the training phase, a flag, $KIND_i$, is assigned to the candidate site S_{cand} based on the following formula:

$$KIND_i = \begin{cases} 1 & \text{if } sratio \geq L_p \\ 0 & \text{otherwise} \end{cases} \tag{2.20}$$

The candidate site S_{cand} is classified as a true donor site if $KIND_i$ has a value of 1. S_{cand} is classified as a false donor site if $KIND_i$ has a value of 0. Figure 2.4 illustrates the site detection algorithm.

2.3 Experiments and Results

2.3.1 Sequence Data and Evaluation Method

In order to evaluate the accuracy of the HMM system for splicing junction site detection, the database of DNA sequences originally collected by Burset and Guigo [7] is used, This database was used to compare a number of major gene-finding programs [7]. The sequences in this database were obtained from the vertebrate divisions of

GenBank release 85.0 (October, 1994). There are 570 vertebrate sequences in the database, and they all have simple and standard gene structures. Each entry contains a complete protein coding sequence with no in-frame stop codons. There are at least one exon and one intron in all entries in the database. There are 2,079 true donor sites and 2,079 true acceptor sites, all of which are standard splicing junction sites. This means that all the donor sites have 'GT' and all the acceptor sites have 'AG' at the right positions. This database now becomes the standard data set for evaluating gene-finding programs.

The 10-way cross-validation method [46] is applied to evaluate how well the HMM system performs when tested on data that are not in the training data set. Cross validation is a standard experimental technique for determining how well a classifier performs on unseen data [19]. Specifically, the 570 sequences at hand are randomly partitioned into 10 sets. These sets have roughly the same number of true donor sites; the sets also have roughly the same number of true acceptor sites. For each iteration in the 10-way cross-validation experiment, nine out of the ten sets are used as the training data set, and the remaining one set is used as the test data set. The HMM system is trained using the training data set (i.e., all sequences excluding those in the test data set are used as the training data). The system is then tested on the sequences in the test data set. Thus, the training data set consists of 90% and the test data set consists of 10% of the sequences. Each time in the 10-way cross-validation experiment, the HMM system is trained with sequences containing about 1,871 true sites and 135,000 false sites. The HMM system is tested on the sequences containing about 208 true sites and 14,000 false sites.

2.3.2 Experimental Results

The state transition probabilities for the Donor Model and the Acceptor Model are shown in Tables 2.1-2.4. Comparing the state transition probabilities of the true site modules with those of the false site modules, the results show that the proposed HMM system maximizes the differences between the true sites and false sites. The results for detecting splicing junction sites are summarized in Table 2.5 and Table 2.6. The results for each of the 10 test sets of the cross validation are shown, so are the average results for all the 10 test sets. In Table 2.5 and Table 2.6, *TP* is the number of true positives. *FP* is the number of false positives. *TN* is the number of true negatives. *FN* is the number of false negatives. A true positive is a true donor (true acceptor, respectively) site that is also classified as a true donor (true acceptor, respectively) site. A false positive is a false donor (false acceptor, respectively) site that is mis-classified as a true donor (true acceptor, respectively) site. A true negative is a false donor (false acceptor, respectively) site that is also classified as a false donor (false acceptor, respectively) site. A false negative is a true donor (true acceptor, respectively) site that is mis-classified as a false donor (false acceptor, respectively) site. S_n^{true} is the ratio between the number of correctly classified true donor (true acceptor, respectively) sites and the total number of true donor (true acceptor, respectively) sites in the test data set, i.e.,

$$S_n^{true} = \frac{TP}{TP + FN} \tag{2.21}$$

The similar calculations are also used to evaluate the performance of the proposed HMM system in predicting the false splicing junction sites. S_n^{false} is the ratio between the number of correctly classified false donor (false acceptor, respectively) sites and the total number of false donor (false acceptor, respectively) sites in the test data set, i.e.,

$$S_n^{false} = \frac{TN}{TN + FP} \tag{2.22}$$

S_n is the proportion of the candidate sites in the test data set that are classified correctly. S_n tells how well the proposed HMM system can assign true sites and false

Table 2.1 State Transition Probabilities for True Donor Module

	$ftr_i^{(t)}(b_i, b_{i+1})$							
i	1	2	3	4	5	6	7	8
A → A	0.21	0.04	null	null	null	0.32	0.04	0.01
A → G	0.05	0.51	0.08	null	null	0.06	0.63	0.03
A → C	0.02	0.01	null	null	null	0.06	0.02	0.01
A → T	0.04	0.04	null	null	null	0.06	0.02	0.02
G → A	0.13	0.02	null	null	null	0.37	0.01	0.12
G → G	0.02	0.11	0.81	null	null	0.04	0.10	0.13
G → C	0.03	0.00	null	null	null	0.02	0.01	0.12
G → T	0.02	0.01	null	1.00	null	0.01	0.00	0.46
C → A	0.23	0.02	null	null	null	0.02	0.02	0.01
C → G	0.02	0.07	0.02	null	null	0.00	0.03	0.00
C → C	0.04	0.01	null	null	null	0.00	0.02	0.02
C → T	0.05	0.02	null	null	null	0.01	0.02	0.02
T → A	0.02	0.00	null	null	0.50	0.01	0.00	0.00
T → G	0.04	0.12	0.08	null	0.44	0.02	0.07	0.02
T → C	0.03	0.01	null	null	0.03	0.00	0.01	0.01
T → T	0.03	0.01	null	null	0.03	0.00	0.00	0.01

Note: The state transition probability values are of 'long int' type in the computer programs. In order to save space, the values are rounded to the second position following the decimal point to fit into this table. For example, a probability value of 0.13293 is shown in this table as 0.13, but 0.13593 is shown here as 0.14. Theoretically, the sum of the transition probabilities from one state to the next state should equal to 1.00. Because of the rounding, the sum of the values in each column in this table may be slightly smaller or greater than 1.00. This holds in Tables 2.2 - 2.4 as well.

Table 2.2 State Transition Probabilities for False Donor Module

	$ftr_i^{(f)}(b_i, b_{i+1})$							
i	1	2	3	4	5	6	7	8
A → A	0.08	0.08	null	null	null	0.05	0.06	0.06
A → G	0.07	0.08	0.28	null	null	0.05	0.07	0.07
A → C	0.05	0.02	null	null	null	0.04	0.05	0.04
A → T	0.05	0.08	null	null	null	0.05	0.05	0.06
G → A	0.07	0.07	null	null	null	0.09	0.05	0.06
G → G	0.07	0.08	0.27	null	null	0.10	0.07	0.07
G → C	0.06	0.02	null	null	null	0.07	0.05	0.05
G → T	0.05	0.09	null	1.00	null	0.09	0.05	0.07
C → A	0.08	0.08	null	null	null	0.06	0.07	0.07
C → G	0.02	0.02	0.08	null	null	0.01	0.02	0.02
C → C	0.07	0.02	null	null	null	0.07	0.07	0.07
C → T	0.06	0.12	null	null	null	0.08	0.08	0.09
T → A	0.05	0.04	null	null	0.18	0.04	0.05	0.05
T → G	0.09	0.09	0.37	null	0.35	0.06	0.09	0.08
T → C	0.07	0.02	null	null	0.22	0.06	0.07	0.06
T → T	0.07	0.09	null	null	0.25	0.09	0.08	0.08

sites into the right categories; it is calculated by the following formula:

$$S_n = \frac{N_c}{N_t} \tag{2.23}$$

where N_c is the number of the candidate sites in the test data set that are classified correctly and N_t is the total number of the candidate sites in the test data set.

The results in Table 2.5 show that, on average, the system can correctly detect 92% of the true donor sites in the test data set, and 95% of the false donor sites in the test data set are predicted as false sites. Overall, 95% of the candidate donor sites are classified into the right categories. The results for acceptor classification are shown in Table 2.6. The proposed HMM system can correctly predict 91.5% of the true acceptor sites in the test data set and 93% of the false acceptor sites in the test data set. In general, the system can assign 93% of the candidate acceptor sites into the right categories.

Table 2.3 State Transition Probabilities for True Acceptor Module

	$ftr_i^{(t)}(b_i, b_{i+1})$							
i	1	2	3	4	5	6	7	8
A $\rightarrow$ A	0.01	0.01	0.00	0.01	0.02	0.01	0.00	0.01
A $\rightarrow$ G	0.01	0.01	0.00	0.01	0.00	0.00	0.00	0.00
A $\rightarrow$ C	0.04	0.04	0.03	0.02	0.03	0.03	0.03 i	0.03
A $\rightarrow$ T	0.05	0.03	0.03	0.03	0.03	0.03	0.05	0.01
G $\rightarrow$ A	0.02	0.01	0.02	0.00	0.01	0.01	0.00	0.01
G $\rightarrow$ G	0.03	0.03	0.04	0.03	0.03	0.02	0.03	0.03
G $\rightarrow$ C	0.04	0.04	0.05	0.03	0.03	0.03	0.03	0.05
G $\rightarrow$ T	0.05	0.05	0.04	0.05	0.05	0.04	0.06	0.04
C $\rightarrow$ A	0.04	0.03	0.03	0.04	0.03	0.05	0.02	0.04
C $\rightarrow$ G	0.03	0.03	0.01	0.01	0.02	0.01	0.03	0.01
C $\rightarrow$ C	0.14	0.14	0.14	0.14	0.14	0.16	0.17	0.18
C $\rightarrow$ T	0.18	0.15	0.17	0.21	0.18	0.16	0.15	0.19
T $\rightarrow$ A	0.02	0.02	0.03	0.02	0.02	0.02	0.02	0.03
T $\rightarrow$ G	0.07	0.07	0.05	0.06	0.05	0.09	0.06	0.04
T $\rightarrow$ C	0.14	0.15	0.18	0.17	0.18	0.16	0.19	0.19
T $\rightarrow$ T	0.15	0.19	0.17	0.16	0.20	0.17	0.14	0.14

A $\rightarrow$ A	0.01	0.00	0.02	0.01	0.02	null	null
A $\rightarrow$ G	0.00	0.00	0.02	0.00	null	1.00	null
A $\rightarrow$ C	0.04	0.03	0.02	0.17	null	null	null
A $\rightarrow$ T	0.03	0.01	0.00	0.04	null	null	null
G $\rightarrow$ A	0.00	0.01	0.01	0.00	0.01	null	0.30
G $\rightarrow$ G	0.01	0.02	0.03	0.00	null	null	0.48
G $\rightarrow$ C	0.03	0.01	0.02	0.23	null	null	0.14
G $\rightarrow$ T	0.04	0.01	0.01	0.03	null	null	0.09
C $\rightarrow$ A	0.02	0.03	0.13	0.01	0.79	null	null
C $\rightarrow$ G	0.01	0.01	0.06	0.00	null	null	null
C $\rightarrow$ C	0.24	0.25	0.16	0.24	null	null	null
C $\rightarrow$ T	0.17	0.21	0.08	0.07	null	null	null
T $\rightarrow$ A	0.01	0.02	0.06	0.00	0.18	null	null
T $\rightarrow$ G	0.03	0.04	0.15	0.00	null	null	null
T $\rightarrow$ C	0.18	0.14	0.12	0.14	null	null	null
T $\rightarrow$ T	0.16	0.21	0.11	0.06	null	null	null

Note: top, probabilities for the first 8 transitions; bottom, probabilities for the remaining 7 transitions.

Table 2.4 State Transition Probabilities for False Acceptor Module

	$ftr_i^{(f)}(b_i, b_{i+1})$							
i	1	2	3	4	5	6	7	8
A → A	0.05	0.05	0.05	0.06	0.06	0.06	0.06	0.08
A → G	0.08	0.10	0.10	0.08	0.11	0.08	0.08	0.09
A → C	0.05	0.05	0.04	0.05	0.07	0.03	0.06	0.06
A → T	0.05	0.04	0.04	0.03	0.05	0.04	0.04	0.03
G → A	0.08	0.07	0.08	0.11	0.07	0.08	0.08	0.07
G → G	0.10	0.10	0.13	0.11	0.11	0.13	0.12	0.12
G → C	0.08	0.07	0.07	0.07	0.07	0.08	0.07	0.05
G → T	0.06	0.05	0.05	0.06	0.05	0.05	0.06	0.05
C → A	0.08	0.07	0.07	0.07	0.06	0.07	0.07	0.07
C → G	0.03	0.02	0.04	0.03	0.03	0.03	0.02	0.03
C → C	0.06	0.07	0.06	0.06	0.06	0.08	0.07	0.08
C → T	0.08	0.07	0.08	0.06	0.07	0.06	0.07	0.06
T → A	0.04	0.03	0.02	0.04	0.03	0.03	0.04	0.03
T → G	0.07	0.10	0.08	0.09	0.09	0.09	0.07	0.08
T → C	0.04	0.06	0.05	0.04	0.04	0.05	0.05	0.06
T → T	0.04	0.04	0.04	0.04	0.04	0.04	0.03	0.04

	$ftr_i^{(f)}(b_i, b_{i+1})$						
i	9	10	11	12	13	14	15
A → A	0.05	0.06	0.05	0.04	0.28	null	null
A → G	0.13	0.11	0.09	0.06	null	1.00	null
A → C	0.04	0.04	0.05	0.04	null	null	null
A → T	0.03	0.03	0.03	0.01	null	null	null
G → A	0.09	0.07	0.04	0.07	0.36	null	0.19
G → G	0.12	0.19	0.12	0.12	null	null	0.43
G → C	0.06	0.07	0.05	0.08	null	null	0.18
G → T	0.05	0.04	0.19	0.03	null	null	0.20
C → A	0.07	0.05	0.04	0.07	0.28	null	null
C → G	0.04	0.03	0.03	0.03	null	null	null
C → C	0.07	0.06	0.08	0.10	null	null	null
C → T	0.06	0.06	0.06	0.02	null	null	null
T → A	0.03	0.03	0.03	0.09	0.08	null	null
T → G	0.09	0.07	0.07	0.15	null	null	null
T → C	0.03	0.04	0.04	0.05	null	null	null
T → T	0.04	0.03	0.03	0.01	null	null	null

Note: top, probabilities for the first 8 transitions; bottom, probabilities for the rest of the 7 transitions.

To investigate how well the proposed HMM system can discriminate true splicing junction sites from false splicing junction sites when a group of candidate sequences are presented to the system, some statistic analyses are performed on the scores the HMM system assigned to each candidate site in the 10-way cross-validation experiment. Figure 2.5 shows the score distribution of true donor sites and false donor sites in one test data set. Figure 2.6 shows the score distribution of true acceptor sites and false acceptor sites in the same test data set. Striking differences can be observed by comparing the curves in these figures. The scores for the true donor sites can be higher than 10000, with about 85% of the true donor sites scoring more than 10.0. For the false donor sites, only about 5% of the sequences score more than 1.0, with the majority of the false donor sites scoring between 0.1 and 0.00001. More than 10% of the false donor sites score less than 0.00001. The score distribution for the true acceptor scores in Figure 2.6 shares a similar pattern as the one for the true donor sites shown in Figure 2.5. The scores for the false acceptor sites are more scattered, but again, there are only 5% to 6% of the sequences scoring more than 1. The results suggest that the proposed HMM system can be used to discover the degenerate features of the splicing junction sites to a great degree.

2.4 Conclusions

In this chapter, hidden Markov models (HMMs) are developed to represent the consensus and degenerate features of splicing junction sites in eukaryotic genes. The proposed Donor Model and Acceptor Model have a different topology from those previously reported for splicing junction site detection. To capture the consensus and degenerate features of the splicing junction sites, the constant states and constant state transitions are introduced into the models. This innovative approach conceptually simplifies the splicing junction site models and the computation process of using the models. The results from the 10-way cross-validation experiment show

Table 2.5 Performance Evaluation of the HMM System for Detecting Donor Sites

Set	*true sites*	*false sites*	*TP*	*FP*	*TN*	*FN*	S_n^{true}	S_n^{false}	S_n
1	209	16259	191	634	15625	18	0.914	0.961	0.960
2	210	13411	191	643	12768	19	0.910	0.952	0.951
3	203	12942	185	677	12265	18	0.911	0.948	0.947
4	200	15473	183	654	14819	17	0.915	0.958	0.957
5	208	17245	192	815	16430	16	0.923	0.952	0.952
6	213	15817	205	809	15008	8	0.962	0.948	0.949
7	206	15895	191	762	15133	15	0.927	0.951	0.952
8	212	13206	194	748	12458	18	0.915	0.942	0.953
9	209	14334	192	702	13632	17	0.919	0.950	0.951
10	209	14651	190	702	13949	19	0.909	0.951	0.952
Average							0.921	0.951	0.952

Table 2.6 Performance Evaluation of the HMM System for Detecting Acceptor Sites

Set	*true sites*	*false sites*	*TP*	*FP*	*TN*	*FN*	S_n^{true}	S_n^{false}	S_n
1	209	21553	198	1428	20125	11	0.947	0.933	0.934
2	210	19169	197	1371	17798	13	0.938	0.928	0.929
3	203	19995	183	1404	18591	20	0.901	0.929	0.929
4	200	22683	181	1364	21319	19	0.905	0.939	0.940
5	208	24721	194	1416	23305	14	0.933	0.942	0.943
6	213	23871	194	1392	22479	19	0.911	0.941	0.941
7	206	22877	186	1388	21489	20	0.903	0.938	0.939
8	212	19012	192	1400	17612	20	0.906	0.925	0.926
9	209	20798	189	1398	19400	20	0.904	0.932	0.932
10	209	18221	189	1377	16844	20	0.904	0.924	0.923
Average							0.915	0.934	0.934

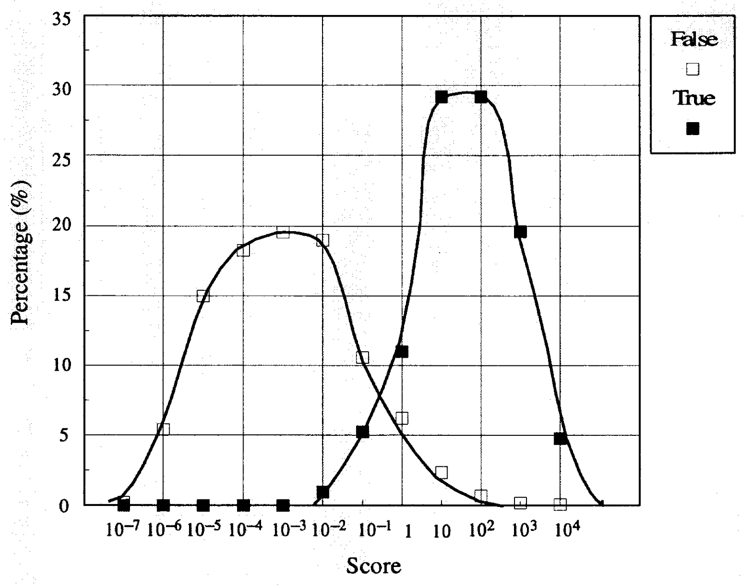


Figure 2.5 Score distributions for true donor sites and false donor sites.

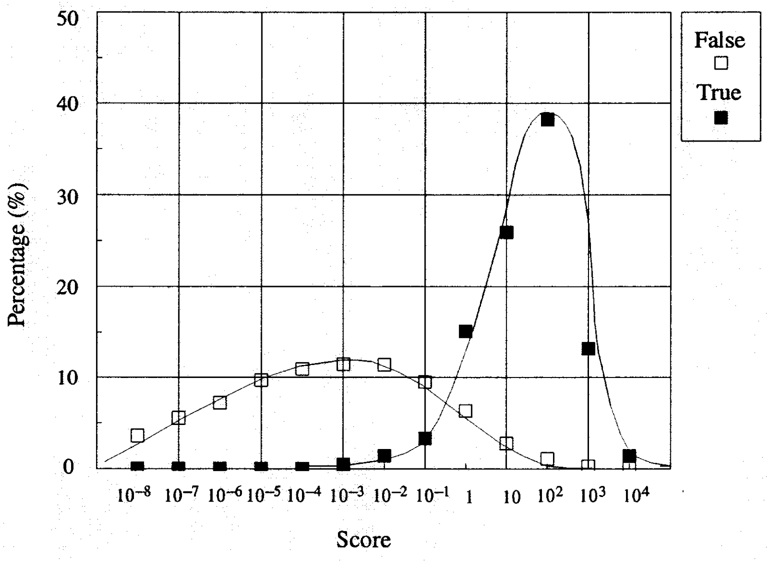


Figure 2.6 Score distributions for true acceptor sites and false acceptor sites.

that the proposed HMM system can correctly detect 92% of the true donor sites and 91.5% of the true acceptor sites in the standard sequence data set composed by Burset and Guigo. The following chapters will introduce the integration of the HMM system with other gene structure information to develop a system for full gene structure detection.

CHAPTER 3

TRANSLATIONAL START SITE MODELING AND DETECTING

The *Start* codon is always *ATG* and it is the start position in mRNA for protein translation, so the start codon is the first three bases of the coding region of a gene. *ATG* is also the codon for Methionine, a regular amino acid occurring at many positions in all of the known proteins. This means one can not just detect the start codon by simply searching for ATG in the genomic DNA. It is reported that there are some statistic relations between the start codon and the nucleotides before (13 bases) and after (three bases) it [29]. In this study, this 19 bases with a start codon is referred to as *start site* or *true start site.* In contrast to true start site, *false start site* refers to a 19-bases sequence containing no start codon but with nucleotides 'A', 'T' and 'G' at position 14, 15 and 16, respectively.

3.1 The Start Site Model

An HMM with 19 states and a set of transitions is defined for modeling a true start site, represented as a digraph where states correspond to vertices and transitions to edges. At each state, the HMM will generate a base b in $\{A, G, C, T\}$ according to the state and transition probabilities, with the exception of states 14, 15 and 16. The HMM constantly generates base b = A at state 14, b = T at state 15 and b = G at state 16. Each state s is associated with a discrete output probability distribution, $P(s)$. Obviously, for state 14, state 15 and state 16, $P(s) = 1$. Except at states 13, 14 and 15, each base b at a state has four possible transitions to the next state. Each transition has a probability, $P(t)$, which represents the probability that the HMM makes that transition. Each base at state 13 has a fixed transition, namely 1 to the base A at state 14. At state 14, the base A has a fixed transition, namely one to base T at state 15. Similarly, at state 15, the base T has a fixed transition, namely

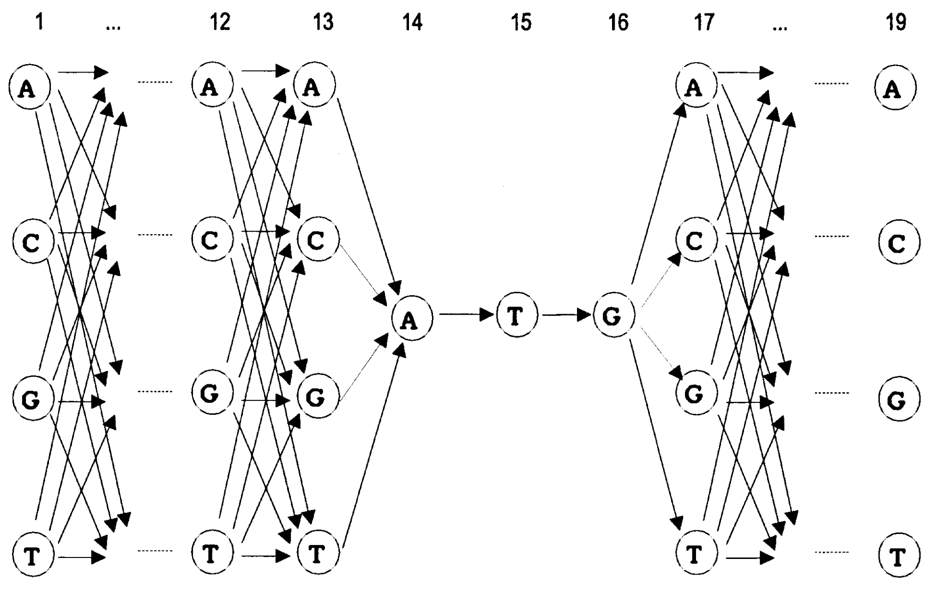


Figure 3.1 The Start Site Model for translational start site detection.

one to base G at state 16. Figure 3.1 shows the graph model of the HMM Start Site Model.

In the vertebrate DNA sequences, there are much more false Start sites than true Start sites. The ratio between the number of false sites and the number of true sites is about 100 to 1. In order to mine out the differences between the true sites and false sites, as presented in Chapter 2 for the donor and acceptor models, two programs, True Start Module and False Start Module are implemented for the Start Site Model.

3.2 Start Site Detecting Algorithms

The HMM algorithms described in Section 2.2 (see also the published paper [49]) are used to train the start site modules and to detect start sites in vertebrate genomic DNA. The final state transition probabilities for Start Site Model are shown in Table 3.1 and 3.2. Comparing the state transition probabilities from the true start site module with those from the false start site module, this HMM system maximized the differences between the true start sites and false sites.

3.3 Experiments and Results

The experimental sequence data and evaluation method are the same as described in Section 2.3. Briefly, the DNA sequence database originally collected by Burset and Guigo [7] is used to evaluate the accuracy of the HMM system for start site detection; and the 10-way cross-validation method [46, 49] is applied to evaluate how well HMM systems will perform when tested on data that are not in the training data set.

The start sites detection results are summarized in Table 3.3. The results for each of the ten test sets of the cross-validation are shown, so are the average results for all the ten test sets. The results in Table 3.3 show that, on average, this system can correctly detect about 90% of the true start sites, and 94% of the candidate start sites are correctly classified into the right categories.

Table 3.1 State Transition Probabilities for True Start Site Module

	$ftr_i^{(t)}(b_i, b_{i+1})$								
i	1	2	3	4	5	6	7	8	9
A → A	0.06	0.10	0.09	0.04	0.05	0.05	0.04	0.04	0.10
A → G	0.10	0.06	0.09	0.05	0.05	0.20	0.09	0.04	0.10
A → C	0.06	0.06	0.10	0.13	0.05	0.02	0.04	0.17	0.01
A → T	0.01	0.02	0.05	0.01	0.02	0.02	0.02	0.01	0.01
G → A	0.03	0.11	0.04	0.06	0.05	0.05	0.19	0.04	0.10
G → G	0.04	0.04	0.05	0.05	0.03	0.05	0.07	0.04	0.07
G → C	0.11	0.05	0.06	0.10	0.06	0.05	0.13	0.11	0.01
G → T	0.02	0.05	0.02	0.04	0.03	0.01	0.06	0.01	0.01
C → A	0.14	0.10	0.07	0.05	0.16	0.09	0.02	0.10	0.44
C → G	0.04	0.03	0.06	0.03	0.04	0.07	0.01	0.11	0.09
C → C	0.11	0.10	0.07	0.16	0.14	0.09	0.10	0.15	0.02
C → T	0.12	0.08	0.08	0.10	0.14	0.06	0.08	0.01	0.00
T → A	0.02	0.02	0.02	0.01	0.04	0.01	0.01	0.03	0.03
T → G	0.05	0.04	0.05	0.04	0.04	0.14	0.04	0.04	0.02
T → C	0.09	0.07	0.11	0.09	0.06	0.05	0.06	0.12	0.01
T → T	0.03	0.05	0.03	0.04	0.05	0.04	0.04	0.01	0.00

	$ftr_i^{(t)}(b_i, b_{i+1})$								
i	10	11	12	13	14	15	16	17	18
A → A	0.18	0.07	0.15	null	null	null	0.10	0.03	0.04
A → G	0.10	0.13	null	null	null	null	0.11	0.14	0.03
A → C	0.30	0.04	null	null	null	null	0.04	0.04	0.03
A → T	0.08	0.01	null	1.00	null	null	0.02	0.03	0.01
G → A	0.05	0.02	0.22	null	null	0.27	0.08	0.02	0.11
G → G	0.02	0.05	null	null	null	0.49	0.06	0.08	0.10
G → C	0.18	0.04	null	null	null	0.15	0.17	0.06	0.20
G → T	0.02	0.00	null	null	null	0.09	0.18	0.10	0.03
C → A	0.02	0.04	0.58	null	null	null	0.05	0.05	0.09
C → G	0.01	0.03	null	null	null	null	0.00	0.05	0.03
C → C	0.02	0.40	null	null	null	null	0.04	0.06	0.06
C → T	0.00	0.03	null	null	null	null	0.05	0.10	0.05
T → A	0.01	0.01	0.05	null	null	null	0.00	0.01	0.03
T → G	0.00	0.00	null	null	1.00	null	0.03	0.19	0.05
T → C	0.00	0.09	null	null	null	null	0.04	0.04	0.09
T → T	0.01	0.00	null	null	null	null	0.02	0.03	0.03

Note: top, probabilities for the first 9 transitions; bottom, probabilities for the remaining 9 transitions.

Table 3.2 State Transition Probabilities for False Start Site Module

	$ftr_i^{(t)}(b_i, b_{i+1})$								
i	1	2	3	4	5	6	7	8	9
A → A	0.08	0.08	0.08	0.08	0.08	0.08	0.08	0.08	0.08
A → G	0.07	0.07	0.07	0.07	0.08	0.07	0.08	0.08	0.08
A → C	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05
A → T	0.07	0.06	0.07	0.07	0.07	0.07	0.07	0.06	0.07
G → A	0.06	0.06	0.07	0.06	0.06	0.07	0.07	0.06	0.07
G → G	0.07	0.07	0.02	0.06	0.07	0.07	0.06	0.07	0.08
G → C	0.05	0.05	0.06	0.05	0.05	0.05	0.05	0.04	0.05
G → T	0.06	0.05	0.06	0.06	0.06	0.06	0.06	0.05	0.06
C → A	0.07	0.07	0.07	0.07	0.07	0.07	0.06	0.08	0.07
C → G	0.02	0.02	0.02	0.01	0.01	0.02	0.01	0.11	0.02
C → C	0.05	0.06	0.06	0.05	0.05	0.05	0.05	0.06	0.05
C → T	0.07	0.07	0.07	0.07	0.07	0.07	0.07	0.06	0.07
T → A	0.05	0.06	0.05	0.06	0.06	0.06	0.06	0.06	0.06
T → G	0.08	0.09	0.07	0.08	0.09	0.08	0.08	0.09	0.08
T → C	0.06	0.06	0.07	0.06	0.06	0.05	0.06	0.06	0.05
T → T	0.08	0.08	0.08	0.08	0.08	0.08	0.09	0.08	0.07

	$ftr_i^{(f)}(b_i, b_{i+1})$								
i	10	11	12	13	14	15	16	17	18
A → A	0.09	0.10	0.28	null	null	null	0.08	0.08	0.07
A → G	0.08	0.09	null	null	null	null	0.07	0.08	0.07
A → C	0.05	0.07	null	null	null	null	0.05	0.05	0.09
A → T	0.06	0.05	null	1.00	null	null	0.06	0.07	0.04
G → A	0.08	0.07	0.25	null	null	0.26	0.07	0.06	0.06
G → G	0.07	0.07	null	null	null	0.27	0.08	0.07	0.07
G → C	0.05	0.06	null	null	null	0.20	0.06	0.05	0.09
G → T	0.05	0.04	null	null	null	0.26	0.06	0.06	0.03
C → A	0.08	0.06	0.27	null	null	null	0.06	0.07	0.05
C → G	0.01	0.01	null	null	null	null	0.01	0.01	0.06
C → C	0.05	0.08	null	null	null	null	0.06	0.06	0.07
C → T	0.06	0.05	null	null	null	null	0.07	0.07	0.03
T → A	0.06	0.05	0.20	null	null	null	0.06	0.05	0.07
T → G	0.08	0.08	null	null	1.00	null	0.08	0.08	0.07
T → C	0.05	0.06	null	null	null	null	0.05	0.05	0.10
T → T	0.07	0.05	null	null	null	null	0.07	0.07	0.04

Note: top, probabilities for the first 9 transitions; bottom, probabilities for the remaining 9 transitions.

Table 3.3 Performance Evaluation for the HMM Start Site Model

Set	*True Site*	*False Site*	*TP*	*FP*	*TN*	*FN*	S_n^{true}	S_n^{false}	S_n
1	57	4492	53	253	4239	4	0.930	0.944	0.944
2	57	4546	52	210	4336	5	0.912	0.954	0.953
3	57	6726	51	491	6235	6	0.895	0.927	0.927
4	57	4269	50	300	3969	7	0.877	0.930	0.929
5	57	6506	51	436	6070	6	0.895	0.933	0.933
6	57	4367	51	248	4119	6	0.895	0.943	0.943
7	57	3660	53	218	3442	4	0.930	0.940	0.940
8	57	2892	50	157	2735	7	0.877	0.946	0.944
9	57	3977	51	241	3736	6	0.895	0.939	0.939
10	57	4441	53	199	4242	4	0.930	0.955	0.955
Average							0.904	0.941	0.941

CHAPTER 4

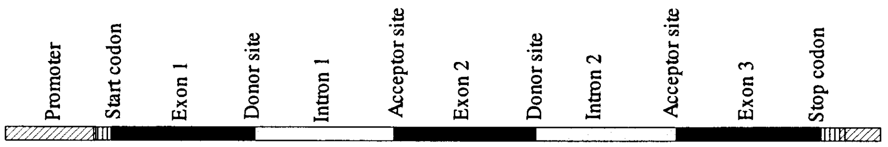
A DATA MINING SYSTEM FOR PREDICTING VERTEBRATE GENES

4.1 Introduction

Data mining, or knowledge discovery from data, refers to the process of extracting interesting, non-trivial, implicit, previously unknown and potentially useful information or patterns from data [15]. In life sciences, this process could refer to finding clustering rules for gene expression, discovering classifications rules for proteins, detecting associations between metabolic pathways, predicting genes in genomic DNA sequences, etc. [39, 40, 41, 42]. This chapter presents a data mining system for automated gene discovery.

A genomic DNA sequence is comprised of four types of nucleotides, or bases, represented by English letters `A`, `C`, `G`, and `T`. Identification or prediction of coding regions from within a genomic DNA sequence has been a major rate-limiting step in the pursuit of genes. The Human Genome Project has produced millions of nucleotides of sequences, and it becomes increasingly important to rapidly identify genes in these sequences [30]. The basic structure for a vertebrate gene includes a promoter, a start codon, introns, exons, donors, acceptors, and a stop codon; cf. Figure 4.1(A). The exon sequences of a gene are also called the *coding sequences* of this gene, and the whole exon sequences of a gene are called the *coding region* of the gene (which is the region for making proteins); cf. Figure 4.1(B). Intron sequences range in size from about 80 nucleotides to 10,000 nucleotides or more. Introns in genes have no function at all and are actually the genetic "junk" [1, 4]. They differ dramatically from exons in that their exact nucleotide sequences seem to be unimportant. The only highly conserved sequences in introns are those required for intron removal from DNA.

A.



B.

```
                                             ↓
CTGGTGAGGCTCCTTCTAACTCCTAACTTTTCCCTTCTCTGGCAGGCCCCTTTGAAGGTC
CAAACTATCACATTGCGCCACGCTGGGTCTACAATATCACTTCTGTCTGGATGATTTTTG
TGGTCATCGCTTCAATCTTCACCAATGGTTTGGTATTGGTGGCCACTGCCAAATTCAAGA
AGCTACGGCATCCTCTGAACTGGATTCTGGTAAACTTGGCGATAGCTGATCTGGGTGAGA
CGGTTATTGCCAGTACCATCAGTGTCATCAACCAGATCTCTGGCTACTTCATCCTTGGCC
ACCCTATGTGTGTGCTGGAGGGATACACTGTTTCAACTTGTGGTAAGAGACAAATTGATG
                                         ↑
```

Figure 4.1 Vertebrate gene structure (A) and a sequence fragment (B).

This sequence fragment contains an exon of 296 nucleotides. The `AG` bases preceding the first arrow are the conserved nucleotides in a splicing junction acceptor site. The `GT` bases preceding the second arrow are the conserved nucleotides in a splicing junction donor site. The nucleotides between the two arrows constitute the exon.

In this chapter, the gene structure signal information is combined with global gene structure information and GeneScout, a full-scale gene structure detection system is presented.

The rest of the chapter is organized as follows. Section 4.2 surveys related work. Section 4.3 presents the approach used in this study for gene discovery. Section 4.4 reports experimental results. Section 4.5 concludes the chapter.

4.2 Related Work

Although methods for predicting coding regions in genomic DNA sequences have existed since the 1980s, the programs for assembling coding sequences into translatable mRNA sequences were not available until the early 1990s [7]. Recently, there have been several programs available for biologists, such as GenViewer [26], GeneID [14], GenLang [10], GeneParser [32], FGENEH [33], SORFIND [20], Xpound [35], GRAIL [43], VEIL [19], GenScan [6], etc. Among the tools, GRAIL and GenScan are widely used in academia and industry. GRAIL is available on the BLAST Web site (http://www.ncbi.nlm.nih.gov) and GenScan is available at MIT's Web site (http://genes.mit.edu/license.html). Algorithms employed by these various tools are based on *consensus search* [12], *weight matrices* [22], *pattern recognition* [12], etc., though the most popular approaches are based on *neural networks* (NNs) [43] and, more recently, *hidden Markov models* (HMMs) [23, 29, 49]. The basic algorithms using NN techniques are described below.

4.2.1 NN-Based Techniques

A learning sample of NN-based techniques consists of two classes: sites and non-sites. The non-sites class is usually formed by randomly choosing fragments from the natural DNA. The basic steps of the NN-based techniques are as follows [24, 25, 37, 38]:

1. Creation of a learning sample;
2. Choice and encoding of signal features;
3. Iterative correction of recognition rules according discrimination between the two classes obtained in the previous round;
4. Testing on an independent sample.

Often, an NN consists of a layer of input neurons, several layers of hidden neurons, and an output neuron. When an unlabeled candidate site is presented to the NN, the input neurons check whether the site possesses the corresponding features and send binary signals to the neurons of the first hidden layer. Each hidden neuron sums the weighted signals coming, by connection, from lower-level neurons, compares the result with some thresholds, and sends a binary signal to upper-level neurons. The output neuron provides the user with a final site/non-site decision. Several research groups implemented NNs for gene structure prediction, and they adopted two different techniques in combination with NNs [12]: (i) the use of *ad hoc* heuristic procedures and (ii) the use of combinatorial algorithms.

The first technique was adopted in GRAIL [36, 43]. The exon recognition module of GRAIL employs an NN that integrates values of various coding potentials and other statistics. The first step is to perform splicing site prediction by similar multisensor networks. The second step is to perform exon prediction by Bayesian analysis of frame-specific coding potentials. Finally, all legitimate combinations of predicted exons are used for gene assembly. At this step, GRAIL employs a heuristic procedure that scores candidate exons using some combination of the site scores and the coding potentials obtained in the previous steps. Low-scoring exons are eliminated from further consideration. The software then performs an exhaustive search over the set of structures generated by the remaining high-scoring exons.

GeneParser [32] is an example using the second technique in which dynamic programming is applied to get rid of the exhaustive search over exon-intron structures. GeneParser combines the classical dynamic programming with an NN, and uses both coding potentials and weight matrix scores of splicing sites for exon prediction. The learning stage consists of a series of iterations. At each iteration, a dynamic programming procedure is applied to find the highest-scoring structures for all sequences in the learning sample. Then another dynamic programming module is used to find the optimal structures for new sequences [12].

4.3 The Approach Used in This Study

The proposed GeneScout system contains several specially designed HMM models for predicting functional sites as well as an HMM model for calculating coding potentials. The functional sites are some sequence signals common for all sites of a given type, which are recognized by corresponding DNA- and RNA-binding proteins [12]. Basic units of such functional sites in a vertebrate gene sequence include translational start sites, splicing junction donor and acceptor sites, etc. Effectively predicting these function sites is the first and crucial step for gene finding.

4.3.1 HMM Models for Predicting Functional Sites

Functional sites include sites used in the transcription process such as splicing junction donor sites, acceptor sites, and sites used in the translation process such as a start site, etc. Often, the functional sites include (almost) invariant (consensus) nucleotides and other degenerate features. Thus, the invariant nucleotides themselves do not completely characterize a functional site. For example, a start codon is always a sequence of `ATG` and it is the start position in mRNA for protein translation, so the start codon is the first three bases of the coding region of a gene. ATG is also the codon for Methionine,[1] a regular amino acid occurring at many positions in all of the known proteins. This means one is unable to detect the start codon by simply searching for ATG in a genomic DNA sequence.

It is reported that there are some statistic relations between a start codon ATG and the 13 nucleotides immediately preceding it and the three bases immediately following it [29]. As presented in Chapter 3, call these 19 bases containing a start codon a *start site*. An HMM model, the Start Site Model, is built to model the start site. As shown in Figure 3.1, there are 19 states in the Start Site Model. Except for states 14, 15 and 16, there are four possible bases at each state, and a base at one state may have four possible ways to transit to the next state. States 14, 15 and 16 are constant states (representing a start codon), and the transitions from state 14 to 15 and from state 15 to 16 are also constant with a probability of 1.0. With the Start Site Model, the HMM algorithms described previously can be used to detect a start site. The HMM models and algorithms for detecting splicing junction donor and acceptor sites are presented in Chapter 2. The HMM model for a donor site contains nine states whereas the HMM model for an acceptor site contains 16 states. The algorithms used for training the HMM models for start sites, donor sites and acceptor sites and for detecting these functional sites are similar.

4.3.2 Graph Representation of the Gene Detection Problem

The goal of GeneScout is to find coding regions as illustrated in Figure 4.1. Like all other major gene-finding systems surveyed in Section 4.2, GeneScout does not find the promoter at the very beginning of a gene structure as well as the beginning or the end of transcription. A more accurate term for this process might be "coding

[1]A codon is a triplet of contiguous bases in mRNA that code for specific amino acids, which in turn are used for building proteins [41].

region detection", though traditionally the term "gene detection" has been used when referring to these systems [29].

The main hypothesis used in this work is that, given a vertebrate genomic DNA sequence S, it is always possible to construct a directed acyclic graph G such that the path for the actual coding region of S is in the set of all paths on G. Thus, the gene detection problem is reduced to the analysis of paths in the graph G. Dynamic programming algorithms are used to find the optimal path in G.

Define a *candidate exon* to be a sequence fragment whose left boundary is an acceptor site or a start codon, and whose right boundary is a donor site or a stop codon. A *candidate intron* is a sequence fragment with a donor site at its left boundary and an acceptor site at its right boundary. Define a *candidate gene* as a chain of non-intersecting alternating exons and introns that satisfy the following biological consistency conditions [28]:

1. the total length of exons is divisible by 3;
2. there are no in-frame stop codons in exons;
3. the first intron-exon boundary is a start codon, and the last exon-intron boundary is a stop codon.

However, like most of the existing gene detection programs, this algorithm can be easily generalized for incomplete genes violating condition 3 and possibly condition 1.

Referring to Figure 4.1 again, if one could detect the start codon and all the splicing junction donor and acceptor sites correctly, the coding region would be found immediately. Unfortunately, there is no program that can correctly and precisely detect the start codon and all the splicing junction sites without any error. Even for the effective hidden Markov models developed in this study for detecting splicing junction sites, there are still false positives mistaken as donor or acceptor sites [49]. Suppose one starts with a vertebrate genomic DNA sequence with marked positions for the start codons, donor sites and acceptor sites that are detected by the HMM algorithms described previously. Then these sites generate a set of candidate exons and candidate introns, and their combinations form a set of candidate genes. Assuming all the true functional sites have been detected, one (and only one) subset of the candidate exons must constitute the real coding region.

Consider a directed acyclic graph G where vertices are functional sites, and edges are exons and introns (Figure 4.2). All the edges from the top vertices to the bottom vertices in the graph G are candidate exons, and the edges from the bottom vertices to the top vertices are candidate introns. There must be a path on G representing real exons-introns as shown by the boldface edges in Figure 4.2. So, given a vertebrate genomic DNA sequence with detected sites, it is always possible to construct a directed acyclic graph G such that the path for real exons-introns is in the set of all paths on G. Thus, the gene detection problem is reduced to the analysis of paths in the graph G.

4.3.3 A Dynamic Programming Algorithm

Consider again the graph G in Figure 4.2. A candidate gene is represented by a path in G. Let S_G denote the set of all paths in G. A score is assigned to each functional site based on the HMM models and algorithms described in Section 4.3.1 [49]. The score is used as the weight of the corresponding vertex v in S_G, and denote that weight as $W(v)$. Each edge (v_1, v_2) in S_G is associated with a weight $W(v_1, v_2)$. The weight $W(v_1, v_2)$ equals the coding potential of the candidate exon or intron corresponding to the edge (v_1, v_2) (the calculation of the coding potential will be described in Section 4.3.4). Let $p = \ell_1 \odot \cdots \odot \ell_n$ be a path in G corresponding to a candidate gene. The path weight can be written as:

$$W(p) = W(\ell_1 \odot \cdots \odot \ell_n) = W(\ell_1) \oplus \cdots \oplus W(\ell_n) = \sum_{i=1}^{n} W(\ell_i) \qquad (4.1)$$

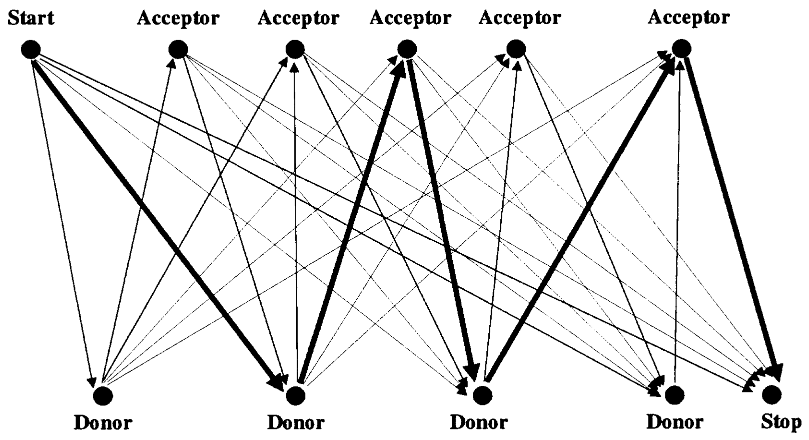


Figure 4.2 The site graph used for vertebrate gene detection.

Note: The boldface edges represent real exons-introns.

If S is a set of paths, the weight of the optimal path in S is:

$$\Theta(S) = \max_{p \in S} W(p) \tag{4.2}$$

Now, let v be a vertex in S_G and let $(v_1, v), \ldots, (v_k, v)$ be all edges entering v. Let $S(v)$ be the set of all paths entering the vertex v. Let $\odot$ represent the concatenating operation, and $\oplus$ represent the simple addition. The weight of the optimal path in $S(v)$, denoted $\Theta(S(v))$, can be calculated as follows:

$$\begin{aligned} \Theta(S(v)) &= \max_{p \in S(v)} W(p) = \max_{i=1}^{k} \left(\max_{p \in S(v_i)} W(p \odot (v_i, v)) \right) & (4.3) \\ &= \max_{i=1}^{k} \left(\left(\max_{p \in S(v_i)} W(p) \right) \oplus W(v_i) \oplus W(v_i, v) \right) & (4.4) \\ &= \max_{i=1}^{k} (\Theta(S(v_i)) \oplus W(v_i) \oplus W(v_i, v)) & (4.5) \end{aligned}$$

This recurrence formula can be used for computing $\theta(S(v))$ given the set of weights $\theta(S(v_i))$, $i = 1, \ldots, k$. Thus, a dynamic programming algorithm can be used to find the weight of the optimal path and locate the path itself in the graph G. This path indicates the real exons (coding region) in the given genomic DNA sequence.

4.3.4 An HMM Model for Computing Coding Potentials

Because vertebrate genes have coding regions and noncoding regions, *coding potential* is used here to measure the difference in statistical characteristics between coding and noncoding regions. The approach for computing coding potentials is based on the analysis of codon usage, which reflects the following phenomena [12]: The universal structure of the genetic code, the average amino acid composition of proteins, the genome-specific patterns of the usage of synonymous codons, and genes intend to use preferred codons in the coding regions. The Codon Model is developed as the basic unit for calculating coding potentials.

Figure 4.3 shows an HMM with 3 states and a set of transitions used for modeling a codon in a vertebrate gene. The HMM is represented as a digraph where

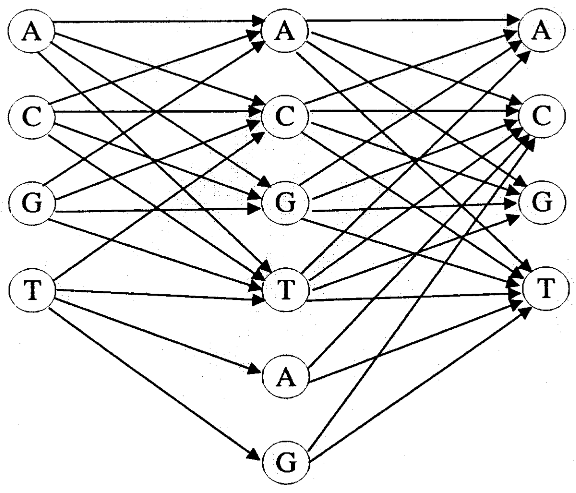


Figure 4.3 The Codon Model.

states correspond to vertices and transitions to edges. At each state, the HMM generates a base b in {A, C, G, T} according to the state and transition probabilities. Notice that, if the HMM generates base $b =$ T in the first state, and generates base $b =$ A or $b =$ G in the second state, the third state can only be $b =$ C or $b =$ T. In other words, if the first state is base T, the transitions (from state 2 to state 3) A $\rightarrow$ A, A $\rightarrow$ G, G $\rightarrow$ A and G $\rightarrow$ G are not defined. This is because TAA, TAG, TGA and TGG are stop codons.

Let $ftr(b_i, b_{i+1})$ be the state transition probability from state i to state $i+1$ (the calculation of $ftr(b_i, b_{i+1})$ will be described in Section 4.3.5). Let U^j_{codon} be the codon usage probability for a codon starting at position j in a coding region. This can be written as:

$$U^j_{codon} = \prod_{i=j}^{j+1} ftr(b_i, b_{i+1}) \tag{4.6}$$

Let P^m_{coding} be the coding potential for a sequence with m bases. Then the coding potential for the sequence can be calculated as follows:

$$P^m_{coding} = \sum_{j=1,4,7,\ldots}^{m-2} U^j_{codon} \tag{4.7}$$

In this study, the Equation (4.7) is used to calculate the coding potentials for candidate exons, which are represented by the edges starting at the top vertices and ending at the bottom vertices in the site graph G shown in Figure 4.2. Notice that if the transition from state i to $i+1$ does not exist, $ftr(b_i, b_{i+1})$ is not defined. If $ftr(b_i, b_{i+1})$ is not defined, U^j_{codon} is not defined, neither is P^m_{coding}. This means that, given a candidate exon represented as an edge in the graph G shown in Figure 4.2, if there are any undefined state transitions, that candidate exon has no coding potential and it is not a real exon. A coding potential value of 0 is assigned to all candidate introns that are represented by the edges starting at the bottom vertices and ending at the top vertices in the site graph G shown in Figure 4.2.

4.3.5 Training and Predicting Algorithms

Let M represent the set of sequences that are randomly picked from a training data set and let E be the set containing the remaining sequences in the training data set. (In the study presented here, M contains about 10% of the sequences in the training data set.) The Codon Model described in Section 4.3.4 is trained using an expectation maximization (EM) [3] algorithm with the exons in set E. The topology for the Codon Model is fixed, and all the transition probabilities and state probabilities are initialized to random values. The program first picks one tenth of the sequences from E and inputs them into the Codon Model; then, records the number of the individual bases at each state and the number of individual transitions from one state to another state; finally, calculates the post probabilities for all the states and transitions in the Codon Model.

Let $Ttr(b_i, b_{i+1})$ be the total number of transitions from a base b_i at state i to a base b_{i+1} at state i +1 in the Codon Model. Let T_{in} be the total number of codons that have been input into the model. The state transition probability $ftr(b_i, b_{i+1})$ in the model can be calculated as follows:

$$ftr(b_i, b_{i+1}) = \frac{Ttr(b_i, b_{i+1})}{T_{in}} \tag{4.8}$$

The re-estimation procedure then adjusts all of the probabilities hidden in the Codon Model. Newly picked sequences in E are then run through the model and the probabilities are further refined. This process is iterated until $ftr(b_i, b_{i+1})$ is maximized or until E becomes empty. This algorithm is guaranteed to converge to a locally optimal estimate of all the probabilities in the HMM [19].

Next, treat all the sequences in set M as unlabeled sequences and input each of the sequences into the HMMs described in Section 4.3.1 to detect start sites, donor sites and acceptor sites. Then the training algorithm constructs the site graph G shown in Figure 4.2 for an input sequence with the detected functional sites on it. The coding region for the input sequence is detected by dynamic programming as specified in Equation (4.5). Comparing the detected coding region with the known gene structures for each of the sequences in M, the approximation correlation (AC) value introduced by Burset and Guigo [7] can be obtained (the calculation of the AC value will be described in detail in Section 4.4.2). In a nutshell, the AC value is the measure that summarizes the prediction accuracy at the nucleotide level. AC ranges from -1 to 1. A value of 1 corresponds to a perfect prediction, while -1 corresponds to a prediction in which each coding nucleotide is predicted as a non-coding nucleotide, and vice versa. A value of 0 is expected for a random prediction [13]. The training process is iterated until the AC value is maximized or the training data set becomes empty.

In the testing (prediction, respectively) phase where an unlabeled test (new, respectively) sequence S is given, GeneScout first detects the functional sites on S and then builds a directed acyclic graph G using the detected functional sites as vertices. Next, GeneScout finds the optimal path on G and outputs the vertices (functional sites) and edges on the optimal path, which displays the coding region on S.

4.4 Experiments and Results

4.4.1 Data

In evaluating the accuracy of the proposed GeneScout system for detecting vertebrate genes, this study adopted the database of human DNA sequences originally collected by Burset and Guigo [7]. The authors used this database to compare a number of gene-finding programs. The sequences in this database were obtained from the vertebrate divisions of GenBank release 85.0 (October, 1994). There are 570 vertebrate sequences in the database and they all have simple and standard gene structures. Each entry contains a complete coding sequence with no in-frame stop codons. There are 28,992,149 nucleotides in these 570 sequences, and there are 2,649 exons, corresponding to 444,498 coding nucleotides. There are at least one exon and one intron in each entry in the database. All the functional sites mentioned in the chapter that appear in these sequences are standard sites. This means that all the start sites have `ATG` as the start codon, and all the donor sites have `GT` and all the acceptor sites have `AG` at appropriate positions [49]. This database now becomes the standard data set for evaluating gene-finding programs.

The 10-way cross-validation method [46, 49] is applied to evaluate how well GeneScout performs when tested on sequences that are not in the training data set. Cross validation is a standard experimental technique for determining how well a classifier performs on unseen data [19]. Specifically, the program randomly partition the 570 sequences at hand into 10 sets. These sets have roughly the same number of exons and introns. For each run in the 10-way cross-validation experiment, nine out of the ten sets are used as the training data set, and the remaining one set is used as the test data set. The GeneScout system is trained using the training data set (i.e., all sequences excluding those in the test data set are used as the training data) and then is tested on the sequences in the test data set. Thus, for each run, the training data set contains 90% of the total exons and the test data set contains 10% of the exons. Notice that each of the 570 sequences is used exactly once in the test data set.

4.4.2 Results

Table 4.1 shows the results obtained in each run of cross-validation, and the average over all the ten runs. The prediction accuracy is estimated at both the nucleotide level and the exon level. At the nucleotide level, let TP_c be the number of true positives, FP_c be the number of false positives, TN_c be the number of true negatives, and FN_c be the number of false negatives. A *true positive* is a coding nucleotide that is correctly predicted as a coding nucleotide. A *false positive* is a non-coding nucleotide that is incorrectly predicted as a coding nucleotide. A *true negative* is a non-coding nucleotide that is correctly predicted as a non-coding nucleotide. A *false negative* is a coding nucleotide that is incorrectly predicted as a non-coding nucleotide. The sensitivity (S_c^n) and specificity (S_c^p) at the nucleotide level described in Table 4.1 are defined as follows:

$$S_c^n = \frac{TP_c}{TP_c + FN_c} \tag{4.9}$$

$$S_c^p = \frac{TP_c}{TP_c + FP_c} \tag{4.10}$$

As mentioned before, the approximation correlation (AC) is the measure that summarizes the prediction accuracy at the nucleotide level. AC ranges from -1 to 1. A value of 1 corresponds to a perfect prediction, while -1 corresponds to a prediction

Table 4.1 Performance Evaluation of GeneScout System for Gene Detection

	Nucleotide			*Exon*	
Run	S_c^n	S_c^p	AC	S_e^n	S_e^p
1	0.86	0.78	0.77	0.51	0.49
2	0.85	0.79	0.77	0.50	0.48
3	0.86	0.80	0.78	0.52	0.50
4	0.85	0.78	0.75	0.49	0.51
5	0.87	0.78	0.78	0.53	0.48
6	0.85	0.79	0.77	0.53	0.49
7	0.84	0.80	0.77	0.52	0.49
8	0.87	0.77	0.76	0.49	0.47
9	0.86	0.78	0.77	0.51	0.48
10	0.86	0.80	0.77	0.52	0.50
Average	0.86	0.79	0.77	0.51	0.49

in which each coding nucleotide is predicted as a non-coding nucleotide, and vice versa. Formally, AC is defined as follows [7]:

$$ACP = \frac{1}{4}\left[\frac{TP_c}{TP_c+FN_c}+\frac{TP_c}{TP_c+FP_c}+\frac{TN_c}{TN_c+FP_c}+\frac{TN_c}{TN_c+FN_c}\right] \quad (4.11)$$

$$AC = (ACP-0.5)\times 2 \quad (4.12)$$

At the exon level, let TP_e be the number of true positives, FP_e be the number of false positives, TN_e be the number of true negatives, and FN_e be the number of false negatives. A *true positive* is an exon that is correctly predicted as an exon. A *false positive* is a non-exon that is incorrectly predicted as an exon. A *true negative* is a non-exon that is correctly predicted as a non-exon. A *false negative* is an exon that is incorrectly predicted as a non-exon. The sensitivity (S_e^n) and specificity (S_e^p) at the exon level described in Table 4.1 are defined as follows:

$$S_e^n = \frac{TP_e}{TP_e + FN_e} \quad (4.13)$$

$$S_e^p = \frac{TP_e}{TP_e + FP_e} \quad (4.14)$$

Table 4.2 Comparison Between GeneScout and Other Systems for Gene Detection

	Nucleotide			*Exon*	
System	S_c^n	S_c^p	AC	S_e^n	S_e^p
GeneScout	0.86	0.79	0.77	0.51	0.49
VEIL	0.83	0.72	0.73	0.53	0.49
FGENEH	0.77	0.88	0.78	0.61	0.64
GeneID	0.63	0.81	0.67	0.44	0.46
GeneParser 2	0.66	0.79	0.67	0.35	0.40
GenLang	0.72	0.79	0.69	0.51	0.52
GRAIL 2	0.72	0.87	0.75	0.36	0.43
SORFIND	0.71	0.85	0.73	0.42	0.47
Xpound	0.61	0.87	0.68	0.15	0.18

The result in Table 4.1 shows that, on average, GeneScout can correctly detect 86 percent of the coding nucleotides in the test data set. Among the predicted coding nucleotides, 79 percent are real coding nucleotides. At the exon level, GeneScout achieved a sensitivity of 51 percent and a specificity of 49 percent. This means GeneScout can detect 51 percent of exons in the test data set with both of their 5' and 3' ends being exactly correct.

Table 4.2 compares GeneScout with other gene finding tools on the same 570 vertebrate genomic DNA sequences. The performance data for the other tools shown in the table are taken from the paper authored by Burset and Guigo [7] except for the VEIL system, whose data is taken from the paper authored by Henderson *et al.* [19] It can be seen from Table 4.2 that GeneScout is comparable to these other programs. It is worth pointing out that, GeneScout beats the widely used neural network based system, GRAIL 2 (the successor of GRAIL). GRAIL 2 detects 36 percent of the 2,649 exons in the 570 vertebrate sequences with a specificity of 43 percent, while GeneScout can detect 51 percent of these exons with a specificity of 49 percent.

Table 4.3 Comparison Between GeneScout and GenScan Systems

	Nucleotide			*Exon*	
System	S_c^n	S_c^p	AC	S_e^n	S_e^p
GenScan	0.93	0.90	0.91	0.78	0.81
GeneScout	0.86	0.79	0.77	0.51	0.49

GeneScout is also compared with another widely used gene-finding program, GenScan, on the same 570 vertebrate genomic DNA sequences. A licensed copy of GenScan was obtained from MIT's Web site (http://genes.mit.edu/ license.html), and then ran the tool on the 570 sequences used in the experiments. Table 4.3 shows the result. It can be seen from the table that GenScan is more accurate than GeneScout at both the nucleotide level and the exon level. However, as indicated by GenScan's inventors Burge and Karlin [6], many of the 570 sequences collected by Burset and Guigo [7] were used to train the GenScan system. This means that a portion of the test sequences were used in GenScan's training process. In contrast, GeneScout is tested on the sequences that are completely unseen in the training phase. Table 4.4 shows the complementarity between GenScan and GeneScout. For the 570 sequences that contained 444,498 coding nucleotides totally, GenScan correctly predicted 93.2 percent of the coding nucleotides, while GeneScout correctly predicted 86.1 percent of the coding nucleotides. If both systems are used together, one can correctly predict (81.4% + 4.7% + 11.7%) = 97.8% of the total coding nucleotides. This is higher than the sensitivity of each individual system.

4.5 Conclusions

In this chapter, the GeneScout data mining system is presented for detecting gene structures in vertebrate genomic DNA. GeneScout uses hidden Markov models to detect functional sites, including start codon sites, splicing junction donor sites and acceptor sites. The main hypothesis is that, given a vertebrate genomic DNA

Table 4.4 Complementarity Between GenScan and GeneScout Systems

Prediction Results	Number of coding nucleotides	Percentage of coding nucleotides
GenScan predicted correctly	414,049	93.2%
GeneScout predicted correctly	382,712	86.1%
GeneScout and GenScan both predicted correctly	361,821	81.4%
GeneScout predicted correctly and GenScan missed	20,891	4.7%
GenScan predicted correctly and GeneScout missed	52,228	11.7%
GenScan and GeneScout both missed	9,558	2.2%

sequence S, it is always possible to construct a directed acyclic graph G such that the path for the actual coding region of S is in the set of all paths on G. Thus, the gene detection problem is reduced to the analysis of paths in the graph G. A dynamic programming algorithm is employed by GeneScout to find the optimal path in G. The system is trained using an expectation maximization algorithm, and its performance on vertebrate gene detection is evaluated using the 10-way cross-validation method on the data set collected by Burset and Guigo [7].

The experiment results show that, GeneScout can correctly detect 86% of the coding nucleotides in the data set with 79% of detected coding nucleotides being correct. The approximation correlation value is 0.77 in predicting coding nucleotides. At the exon level, GeneScout achieves 51% sensitivity and 49% specificity. This means that GeneScout can detect 51% of exons in the data set with both 5' and 3' ends being exactly correct. It was also shown experimentally that GeneScout is comparable to existing gene discovery tools and complements the widely used GenScan system.

CHAPTER 5

SUMMARY OF THE RESEARCH AND FUTURE STUDIES

5.1 Summary

The basic gene structure for higher eukaryotes includes promoter, start codon, introns, exons and stop codon, etc. The boundaries between exons and introns are called splicing junction sites. The exon sequences of a gene are called the coding region of the gene. Identification or prediction of coding sequences from within genomic DNA has been a major rate-limiting step in the pursuit of genes. Biologists study gene structures based on lab experiments such as PCR on cDNA libraries, Northern blot, sequencing, etc. However, characterizing the 60,000 to 100,000 genes thought to be hidden in the human genome by means of merely lab experiments is not feasible. A current trend is to complement the lab study with a bioinformatics approach.

The bioinformatics approach for gene detection means using computer programs to elucidate a gene structure from DNA sequence signals, including start codon, splicing junction donor sites and acceptor sites, stop codon, etc. Since the 1990s, a number of programs have been developed for locating gene coding regions. However, the higher eukaryotic DNA sequence signals involved in gene determination are usually ill defined, degenerate and highly unspecific. Given the current detection methods it is usually impossible to distinguish the signals truly processed by the cellular machinery from those that are apparently non-functional. So, as R. Guigo indicated, automatic sequence analysis and structure elucidation for the genomes of high eukaryotic organisms are far from being a reality.

This dissertation research is targeted toward developing effective and accurate methods for identifying gene structures in the genomes of high eukaryotic organisms. The first phase of this research is for functional sites modeling and detection as presented in Chapters 2 and 3. The second phase, as presented in Chapter 4, is to combine the gene structure signal information with global gene structure information to develop a full gene structure detection system.

Three effective hidden Markov models, the Donor Model, the Acceptor Model and the Start Site Model, have been developed to represent the consensus and degeneracy features of the functional sites in eukaryotic genes. In higher eukaryotic DNA sequences, there are much more false functional sites than true sties. In order to mine out the differences between the true sites and false sites, two programs have been implemented for each model: True Donor Module and False Donor Module based on the Donor Model, True Acceptor Module and False Acceptor Module based on the Acceptor Model, True Start Site Module and False Start Site Module based on the Start Site Model. To capture the consensus and degenerate features of the functional sites, constant states and constant state transitions are introduced into the hidden Markov models. This approach conceptually simplifies the functional site models and the computation process of using the models. The HMM system based on the developed models is fully trained using an expectation maximization (EM) algorithm and the system performance is evaluated using a 10-way cross-validation method. Experimental results show that the proposed HMM system can correctly detect 92% of the true donor sites, 91.5% of the true acceptor sites and 90% of the true start sites in the standard test data set containing real vertebrate gene sequences. The sensitivity and specificity obtained in detecting functional sites are higher than those previously reported. These results suggest that the proposed approach provides a useful tool in discovering the splicing junction sites and start sites in eukaryotic genes.

The GeneScout data mining system is developed in this study for detecting gene structures in vertebrate genomic DNA. GeneScout uses the lahidden Markov models to detect functional sites, including start codon sites, splicing junction donor sites and acceptor sites. The main hypothesis is that, given a vertebrate genomic DNA sequence S, it is always possible to construct a directed acyclic graph G such that the path for the actual coding region of S is in the set of all paths on G. Thus, the gene detection problem is reduced to the analysis of paths in the graph G. A dynamic programming algorithm is employed by GeneScout to find the optimal path in G. The system is trained using an expectation maximization algorithm, and its performance on vertebrate gene detection is evaluated using the 10-way cross-validation method on the data set collected by Burset and Guigo [7]. The experiment results show that, GeneScout can correctly detect 86% of the coding nucleotides in the data set with 79% of detected coding nucleotides being correct. The approximation correlation value is 0.77 in predicting coding nucleotides. At the exon level, GeneScout achieves 51% sensitivity and 49% specificity. This means that GeneScout can detect 51% of exons in the data set with both 5' and 3' ends being exactly correct. It was also shown experimentally that GeneScout is comparable to existing gene discovery tools and complements the widely used GenScan system.

5.2 Future Studies

Future work includes the incorporation of more parameters or criteria into GeneScout. One source of possible new parameters could be obtained from the analysis of potential coding regions, such as preferred exon and intron lengths [18], and positions of exon-intron junctions relative to the reading frame [11]. More functional sites such as those in the upstream or downstream of a coding region may be also modeled. These efforts will further improve GeneScout's performance to make it more accurate for vertebrate gene detection.

APPENDIX A

GENESCOUT TOOLKIT

A.1 Overview of GeneScout

GeneScout is a general-purpose vertebrate gene structure prediction / detection program. For each input genomic DNA sequence, GeneScout determines the most optimal "path" (gene structure) according to probabilistic functional site models and the global vertebrate gene properties. The goal of GeneScout is to find coding regions in the genomic DNA sequences. Like all other major gene-finding systems, GeneScout does not find the promoter at the very beginning of a gene structure as well as the beginning or the end of transcription. A more accurate term for this process might be "coding region detection", though traditionally the term "gene detection" has been used when referring to these systems [29].

For each input DNA sequence, GeneScout will predict the most possible gene structure. Then, the detected gene structure data will be output to a text file.

A.2 GeneScout Installation

GeneScout programs can be obtained in one of the format below:

1. GeneScout.tar:

Files included: GeneScout.exe (executable), sample.seq (the sample sequence file).

2. GeneScoutsrc.tar:

Files included: GeneScout.c (source code), genescout.h (the head file used by GeneScout.c), Makefile (make file for compiling GeneScout programs).

GeneScout installation should be very easy. Below are the suggested procedures for installation on a Linux system. The "hiland> " symbol shown below refers to the shell prompt.

1. Create a directory for installing GeneScout, and copy the tar file to the newly created directory. Extract the tar file:

```
hiland> mkdir GeneScout
hiland> cd GeneScout
hiland> cp <ORIDIR>/GeneScout*.tar .
```

(Note: <ORIDIR> refers to the original directory in which the GeneScout tar file is stored.)

```
hiland> tar -xvf GeneScout*.tar
```

If the tar file name is GeneScoutsrc.tar, compile the source code:

```
hiland> make
```

The make process will print the message below to the stdout if everything is OK:

```
gcc -c -o GeneScout.o GeneScout.c
rm -f GeneScout.exe
gcc -o GeneScout.exe GeneScout.o -lm
```

2. Set up the running environment:

```
hiland> chmod a+x GeneScout.exe
hiland> export PATH=$PWD:$PATH
```

3. The synopsis to run GeneScout is as follows:

```
GeneScout { -cds | -exon } -seq <INPUT FILE> -out [<RESULT FILE>]
```

- **-cds**: GeneScout output option (see below).
- **-exon**: GeneScout output option (see below).
- **-seq**: flag indicating the next parameter will be the sequence file.
- **<INPUT FILE>**: name of the input file where the DNA sequences are stored.
- **-out**: flag indicating the next parameter will be the output file name.

- [**<OUTPUT FILE>**]: name of the output file where the gene detection results will be stored. If this parameter is not provided, the output will print to the screen.

A.3 GeneScout Input Sequence File

The command line arguments for GeneScout programs are described above. The sequence file used by GeneScout must have the format as shown in Figure A.1. The sequence file should begin with a single-line for the sequence name and a brief description followed by lines of sequence data. The sequence name line always starts with a greater-then(">") symbol in the first column. The sequence data can be upper or lower case.

A.4 GeneScout Output

Depending on the first argument, GeneScout will print different content to the sequence output file. For example, if running GeneScout programs by entering the command below:

```
hiland> GeneScout.exe -cds -seq sample.seq -out testout
```

the output will be like the text shown in Figure A.2. And, if running GeneScout programs by entering the command below:

```
hiland> GeneScout.exe -exon -seq sample.seq -out testout
```

the output will be like the text shown in Figure A.3.

```
>ALOEGLOBIM
CCTTGACCAATGACTTTCAAGTACCACGGAAAACAGGGGGGCAGAACTTCAGCAGTAAAG
AATAAAAGGCCATGCAGAGAAGCAGCTGCACATATCTGCTTCCGACACAGCTACAATCAC
CAGCGAGCTCTCAGACCTGACATCATGGTGCATTTTACTGCTGAGGAGAAGGCTGCCATC
ACTAGCCTGTGGGGCAAGATGAATGTGGAAGAGGCTGGAGGTGAAGCCTTGGGCAGGTAA
GCAGTGGTTCTCAATGCATGGGAATAAAGGGTGAATATTACCTTTGCAAGTTGATTGGGA
AAGTCTTCAAGATTTGTTAGCATCTCTAATGTTGTATCTGATATGGTGCCATTTCATAGG
CTCCTGGTTGTTTACCCCTGGACCCAGAGATTTTTTGACAACTTTGGAAACCTGTCCTCT
CCCTCTGCCATCCTGGGCAACCCCAAGGTCAAGGCCCATGGCAAGAAGGTGCTGACATCC
TTTGGAGATGCTATTAAAAACATGGACAACCTCAAGACCACCTTTGCTAAGCTAAGTGAG
CTGCACTGTGACAAGTTGCATGTGGATCCTGAGAACTTCAGGGTGAGTTCAGGCGCGGGT
GATGTGATTGTTTTGGCTTTCTATATTGACATTAATTGAGGTTGAGAATCTTATTGGAAA
CACCAGCAAAGATCTCAGAAATCATGGGTCTAGCTTGATTTTAGAACAGCAGACTTCTAG
TGTGCATAACCAAGGCTACCTTGATTCAGAGCTAGTGACAGTAAAGGGCTACTTGGCTTA
ACTTTTCAAGAAATCTTGCCAGAACTTGATGTGTTTATCCTGGAGAATAGTATTATAAAA
TTGTAGACTTGTGCAAGGAGAATGAAATTTGGCTTTTGATAGATGAAAGCCTGTTTCAAG
GAAATAGAAATGCCTTATTTATGGGTCATGATAACTGAGGTTTAGGAAGAGATGTTTGAA
ACAAAAATTAAAAGATTTTCTCAAAGAAAAATAAGACTAATTTTCTAAAATAGATTAAAT
TTCCCATCAGTATTGTGACCAAGTGAAGGTTTGTTTCTGTATTTGTTAGGGATTTTAAAC
CTCCATGAGAACTCTTGCAGCACTAACATTCTAAGTTTACAGAAATTAGATAACTGGTTA
AAGAAAAATACTGGTAACATGAGGAGAGGGTGAGGGTATAGGTAGGCAGAATGTTGAATG
TAGGGCTCATAAAAATAAATTTGAACTCAAGCTCATCTGAACTTTTTGGGTAGGCACAAA
CCTTGAAACAGTTTGAGGTCCAGGTTGTCAAGGAATGTAGGTATAAAGCCTTTTTTTTTT
TTTTTCAGCAAATTGTTTTTGAAAACTTCTGTTCAACATGCCTATGTGTTATTTTGTCTT
TTGCCTAACAGCTCCTGGGTAACGTGATGGTGATTATTCTGGCTACTCACTTTGGCAAGG
AGTTCACTCCTGAGGTGCAGGCTGCCTGGCAGAAGCTGGTGTCTGCTGTGGCCATTGCCT
TGGGTCACAAGTACCACTGAGTTATCTCCCAGTTTGCCAGTGTTCCTGTGACCCTGACAC
CCTTCTTCTGCACATGAATACTGGGCTTGGCCTTGAGAGGAAGGTTTCTGTTTAATAAAG
TACATTTTCTTCAGTAATCAAAAATTGCAACTTCATCTTCTCCATCTTGTACTCTTGTGC
TAAAGGAAAAG
```

Figure A.1 Sample GeneScout input sequence file.

GeneScout output:
Format: Simple

Input sequence:

```
>ALOEGLOBIM
CCTTGACCAATGACTTTCAAGTACCACGGAAAACAGGGGGCAGAACTTCAGCAGTAAAG
AATAAAAGGCCATGCAGAGAAGCAGCTGCACATATCTGCTTCCGACACAGCTACAATCAC
CAGCGAGCTCTCAGACCTGACATCATGGTGCATTTTACTGCTGAGGAGAAGGCTGCCATC
ACTAGCCTGTGGGGCAAGATGAATGTGGAAGAGGCTGGAGGTGAAGCCTTGGGCAGGTAA
GCAGTGGTTCTCAATGCATGGGAATAAAGGGTGAATATTACCTTTGCAAGTTGATTGGGA
AAGTCTTCAAGATTTGTTAGCATCTCTAATGTTGTATCTGATATGGTGCCATTTCATAGG
CTCCTGGTTGTTTACCCCTGGACCCAGAGATTTTTTGACAACTTTGGAAACCTGTCCTCT
CCCTCTGCCATCCTGGGCAACCCCAAGGTCAAGGCCCATGGCAAGAAGGTGCTGACATCC
TTTGGAGATGCTATTAAAAACATGGACAACCTCAAGACCACCTTTGCTAAGCTAAGTGAG
CTGCACTGTGACAAGTTGCATGTGGATCCTGAGAACTTCAGGGTGAGTTCAGGCGCGGGT
GATGTGATTGTTTTGGCTTTCTATATTGACATTAATTGAGGTTGAGAATCTTATTGGAAA
CACCAGCAAAGATCTCAGAAATCATGGGTCTAGCTTGATTTTAGAACAGCAGACTTCTAG
TGTGCATAACCAAGGCTACCTTGATTCAGAGCTAGTGACAGTAAAGGGCTACTTGGCTTA
ACTTTTCAAGAAATCTTGCCAGAACTTGATGTGTTTATCCTGGAGAATAGTATTATAAAA
TTGTAGACTTGTGCAAGGAGAATGAAATTTGGCTTTTGATAGATGAAAGCCTGTTTCAAG
GAAATAGAAATGCCTTATTTATGGGTCATGATAACTGAGGTTTAGGAAGAGATGTTTGAA
ACAAAAATTAAAAGATTTTCTCAAAGAAAAATAAGACTAATTTTCTAAAATAGATTAAAT
TTCCCATCAGTATTGTGACCAAGTGAAGGTTTGTTTCTGTATTTGTTAGGGATTTTAAAC
CTCCATGAGAACTCTTGCAGCACTAACATTCTAAGTTTACAGAAATTAGATAACTGGTTA
AAGAAAAATACTGGTAACATGAGGAGAGGGTGAGGGTATAGGTAGGCAGAATGTTGAATG
TAAAGGAAAAG
```

Input sequence length: 1691.

Predicted Gene Structure:
(The numbers below are the positions on the input sequence for start, donor, acceptor, donor...)

ALOEGLOBIM 145 236 360 598 1070 1169 1329 1398

Figure A.2 Simple GeneScout output format.

GeneScout output:
Format: Exon

Input sequence:

```
>ALOEGLOBIM
CCTTGACCAATGACTTTCAAGTACCACGGAAAACAGGGGGGCAGAACTTCAGCAGTAAAG
AATAAAAGGCCATGCAGAGAAGCAGCTGCACATATCTGCTTCCGACACAGCTACAATCAC
CAGCGAGCTCTCAGACCTGACATCATGGTGCATTTTACTGCTGAGGAGAAGGCTGCCATC
ACTAGCCTGTGGGGCAAGATGAATGTGGAAGAGGCTGGAGGTGAAGCCTTGGGCAGGTAA
GCAGTGGTTCTCAATGCATGGGAATAAAGGGTGAATATTACCTTTGCAAGTTGATTGGGA
AAGTCTTCAAGATTTGTTAGCATCTCTAATGTTGTATCTGATATGGTGCCATTTCATAGG
CTCCTGGTTGTTTACCCCTGGACCCAGAGATTTTTTGACAACTTTGGAAACCTGTCCTCT
CCCTCTGCCATCCTGGGCAACCCCAAGGTCAAGGCCCATGGCAAGAAGGTGCTGACATCC
TTTGGAGATGCTATTAAAAACATGGACAACCTCAAGACCACCTTTGCTAAGCTAAGTGAG
CTGCACTGTGACAAGTTGCATGTGGATCCTGAGAACTTCAGGGTGAGTTCAGGCGCGGGT
GATGTGATTGTTTTGGCTTTCTATATTGACATTAATTGAGGTTGAGAATCTTATTGGAAA
CACCAGCAAAGATCTCAGAAATCATGGGTCTAGCTTGATTTTAGAACAGCAGACTTCTAG
TGTGCATAACCAAGGCTACCTTGATTCAGAGCTAGTGACAGTAAAGGGCTACTTGGCTTA
ACTTTTCAAGAAATCTTGCCAGAACTTGATGTGTTTATCCTGGAGAATAGTATTATAAAA
TTGTAGACTTGTGCAAGGAGAATGAAATTTGGCTTTTGATAGATGAAAGCCTGTTTCAAG
GAAATAGAAATGCCTTATTTATGGGTCATGATAACTGAGGTTTAGGAAGAGATGTTTGAA
ACAAAAATTAAAAGATTTTCTCAAAGAAAAATAAGACTAATTTTCTAAAATAGATTAAAT
TAAAGGAAAAG
```

Input sequence length: 1691.

Predicted Gene Structure:

Sequence Name: ALOEGLOBIM
Number of exons detected: 4

Exon Number	*Begin At*	*End At*	*Length*
1	145	236	92
2	360	598	239
3	1070	1169	100
4	1329	1398	67

Lenth of Coding Region: 501 bp

Figure A.3 GeneScout output: Exon format.

APPENDIX B

GENESCOUT PROGRAMS SOURCE CODE

B.1 GeneScout.c

```
/*****************************************************************
*                                                               *
*             (c) Copyright 2002                                *
*             All rights reserved                               *
*                                                               *
*             Program written by Michael M. Yin, Ph. D student  *
*             in the group of Professor Jason T.L. Wang         *
*             Department of Computer Science                    *
*             College of Computing Sciences                     *
*             New Jersey Institute of Technology                *
*             University Heights, Newark, NJ 07102, USA         *
*                                                               *
*      This is the main part of GeneScout program source code,  *
*      and it may not be the up-to-date version.                *
*      Programmer(s) makes no representations about the         *
*      suitability of this software for any purpose.            *
*      It is provided "as is" without express or implied        *
*      warranty.                                                *
*                                                               *
*****************************************************************/

/*
 * GeneScout.c
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "splicehmm.h"

#define START    1
#define DONOR    2
#define ACCEPTOR 3
#define STOP     4
typedef struct Trans
{
       long tcnt; //num of trans--edges
       /*
       tfreq: trans frequency.
```

```
        A->T, G->T, C->T, T->T
        tfreq of A->T = (tcnt of A->T)/#.of motif
         */
double tfreq;  //frequence of this trans
} trans;

typedef struct Base
{
        long bcnt;      //base count
        double bfreq;   //freq for this base
        trans toA, toG, toC, toT;  //edges out from this base
} base;

typedef struct State  //HMM state
{
        base A, G, C, T;
} state;

typedef struct SiteData
{
        int seqNum;
        int siteNum;
        double P_t;
        double P_f;
        double score;
} siteData;

state aHMM_T[amotifLen];      //true acceptor HMM
state aHMM_F[amotifLen];      //false acceptor HMM
state dHMM_T[dmotifLen];
state dHMM_F[dmotifLen];
state sHMM_T[smotifLen];
state sHMM_F[smotifLen];


int TotalSeq;
/*total number of false donor sites extracted*/
int totalF = 0;
int totalT = 0;
char seq[MAXLENGTH];
int totalTP = 0;
int totalFP = 0;
/******** In head file *****
double apA, apG, apC, apT,
      afpA, afpG, afpC, afpT,
      dpA, dpG, dpC, dpT,
      dfpA, dfpG, dfpC, dfpT,
      sfpA, sfpG, sfpC, sfpT,
      spA, spG, spC, spT;
********************/
double p_t, p_f;

typedef struct spsite
{
       int bp;   //base position
       int kind; //SART, DONOR ....
       int TP;   //True Positive Known
       struct spsite *pred;
       struct spsite *next;
       struct spsite *next_site;
       long HiScore; //coding base pares
       double score; //score for this site only
       int index; //index in the linked list
       int startFound;
       int nextStop; //only for Acceptor for next STOP codon
} spSite;

spSite *acc, *don, *sta, *tmp;


main(int argc, char** argv)
{
        char seq_file[50];
        char cds_out[50];
        FILE * CDS_Out;
        FILE * F_In;
        int type;  //1 for simple output, 2 for exon output
        if(argc < 5)
            printf("\n USAGE: GeneScout { -cds | -exon }
                -seq <INPUT FILE> -out [RESULT FILE]\n");
        printf("\n\n\tWelcome to use Gene Structure
                   Detecting Program!\n");
        if(strcmp(argv[1], "-cds") == 0)
           type = 1;
        else if(strcmp(argv[2], "-exon") == 0)
           type = 2;
        strcpy(seq_file, argv[3]);
        HmmGetFreq();
        if ((F_In = fopen (seq_file, "r")) == NULL)
        {
                printf("ERROR: %s file open.\n", seq_file);
                exit(0);
        }
        if(argc > 5)
        {
           strcpy(cds_out, argv[5]);
           if ((CDS_Out = fopen (cds_out, "w")) == NULL)
           {
                printf("ERROR: %s file open.\n", cds_out);
                exit(0);
           }
        }
        else
        {
           strcpy(cds_out, "sddout");
           CDS_Out = stdout;
        }
        printf("\n\n\tThe input sequence file is: %s", seq_file);
        printf("\n\tThe out put results will be in: %s\n", cds_out);
        printf("\n\n\tExecuting.......");
        SiteDetect(F_In, CDS_Out, type);
        printf("\n\n\tGeneScout finished gene structure
               detection successfully.");
        fclose(F_In);
        fclose(CDS_Out);
        return 0;
}

/*
* int HmmGetFreq() to initialize all the HMMs by
* the data in the header file
* This means no training but use the training
* data before
*/
int _HmmGetFreq(state *HMM, int motifLen, double *arry)
{
        int arrylen = motifLen * 16;
        int i, j;
        j = -1;
        for (i = 0; i < motifLen; i++)
        {
                HMM[i].A.toA.tfreq = arry[++j];
                HMM[i].A.toG.tfreq = arry[++j];
                HMM[i].A.toC.tfreq = arry[++j];
                HMM[i].A.toT.tfreq = arry[++j];

                HMM[i].G.toA.tfreq = arry[++j];
                HMM[i].G.toG.tfreq = arry[++j];
                HMM[i].G.toC.tfreq = arry[++j];
                HMM[i].G.toT.tfreq = arry[++j];

                HMM[i].C.toA.tfreq = arry[++j];
                HMM[i].C.toG.tfreq = arry[++j];
                HMM[i].C.toC.tfreq = arry[++j];
                HMM[i].C.toT.tfreq = arry[++j];

                HMM[i].T.toA.tfreq = arry[++j];
                HMM[i].T.toG.tfreq = arry[++j];
                HMM[i].T.toC.tfreq = arry[++j];
                HMM[i].T.toT.tfreq = arry[++j];


        }
}

int HmmGetFreq()
{
        _HmmGetFreq(aHMM_T,  amotifLen, t_acept);
        _HmmGetFreq(aHMM_F,  amotifLen, f_acept);
        _HmmGetFreq(dHMM_T,  dmotifLen, t_donor);
        _HmmGetFreq(dHMM_F,  dmotifLen, f_donor);
        _HmmGetFreq(sHMM_T,  smotifLen, t_start);
        _HmmGetFreq(sHMM_F,  smotifLen, f_start);
}

void SiteClassif(char *d, int motifLen, state *HMM_T,
        state *HMM_F, int kind)
{
        int i;
        int j = 0;
        char ch1, ch2;
        double pA, pG, pC, pT, fpA, fpG, fpC, fpT;
        if (kind == DONOR)
        {
           pA = dpA; pG = dpG; pC = dpC; pT = dpT;
           fpA = dfpA; fpG = dfpG; fpC = dfpC; fpT = dfpT;
```

```
          }
          else if (kind == ACCEPTOR)
          {
             pA = apA; pG = apG; pC = apC; pT = apT;
             fpA = afpA; fpG = afpG; fpC = afpC; fpT = afpT;
          }
          else if (kind == START)
          {
             pA = spA; pG = spG; pC = spC; pT = spT;
             fpA = sfpA; fpG = sfpG; fpC = sfpC; fpT = sfpT;
          }
          if (j < 1)
          {
                j++;
          }
p_t = p_f = 1.0;
for (i = 0; i < motifLen; i++)
{
          ch1 = d[i];
                  ch2 = d[i+1];
          switch (ch1)
{
case 'A':
if (i == motifLen-1)
{
p_t = p_t/pA;
p_f = p_f/fpA;
break;
}

switch (ch2)
{
case 'A':
p_t *=HMM_T[i].A.toA.tfreq/pA;
p_f *=HMM_F[i].A.toA.tfreq/fpA;
break;
case 'G':
p_t *=HMM_T[i].A.toG.tfreq/pA;
p_f *=HMM_F[i].A.toG.tfreq/fpA;
break;
case 'C':
p_t *=HMM_T[i].A.toC.tfreq/pA;
p_f *=HMM_F[i].A.toC.tfreq/fpA;
break;
```

```
case 'T':
p_t *=HMM_T[i].A.toT.tfreq/pA;
p_f *=HMM_F[i].A.toT.tfreq/fpA;
break;

}
break;
case 'G':
if (i == motifLen-1)
{
p_t = p_t/pG;
p_f = p_f/fpG;
break;
}
switch (ch2)
{
case 'A':
p_t *=HMM_T[i].G.toA.tfreq/pG;
p_f *=HMM_F[i].G.toA.tfreq/fpG;
break;
case 'G':
p_t *=HMM_T[i].G.toG.tfreq/pG;
p_f *=HMM_F[i].G.toG.tfreq/fpG;
break;
case 'C':
p_t *=HMM_T[i].G.toC.tfreq/pG;
p_f *=HMM_F[i].G.toC.tfreq/fpG;
break;
case 'T':
p_t *=HMM_T[i].G.toT.tfreq/pG;
p_f *=HMM_F[i].G.toT.tfreq/fpG;
break;

}
break;
case 'C':
if (i == motifLen-1)
{
p_t = p_t/pC;
p_f = p_f/fpC;
break;
}
switch (ch2)
{
```

```
case 'A':
p_t *=HMM_T[i].C.toA.tfreq/pC;
p_f *=HMM_F[i].C.toA.tfreq/fpC;
break;
case 'G':
p_t *=HMM_T[i].C.toG.tfreq/pC;
p_f *=HMM_F[i].C.toG.tfreq/fpC;
break;
case 'C':
p_t *=HMM_T[i].C.toC.tfreq/pC;
p_f *=HMM_F[i].C.toC.tfreq/fpC;
break;
case 'T':
p_t *=HMM_T[i].C.toT.tfreq/pC;
p_f *=HMM_F[i].C.toT.tfreq/fpC;
break;

}
break;
case 'T':
if (i == motifLen-1)
{
p_t = p_t/pT;
p_f = p_f/fpT;
break;
}
switch (ch2)
{
case 'A':
p_t *=HMM_T[i].T.toA.tfreq/pT;
p_f *=HMM_F[i].T.toA.tfreq/fpT;
break;
case 'G':
p_t *=HMM_T[i].T.toG.tfreq/pT;
p_f *=HMM_F[i].T.toG.tfreq/fpT;
break;
case 'C':
p_t *=HMM_T[i].T.toC.tfreq/pT;
p_f *=HMM_F[i].T.toC.tfreq/fpT;
break;
case 'T':
p_t *=HMM_T[i].T.toT.tfreq/pT;
p_f *=HMM_F[i].T.toT.tfreq/fpT;
break;
```

```
}
                break;
            }
        }
}

void SiteDetect(FILE *F_In, FILE *CdsOut, int type)
{
        int i,j,k,n,p,m,q,kd, at, ft, seqlen, exonlen;

        int thisSeqTotal;
        int thisSeqTotalD;
        int thisSeqTotalS;
        char accept[amotifLen + 1];
        char donor[dmotifLen + 1];
        char start[smotifLen + 1];
        char seq[MAXLENGTH], T[100];
        char ch;
        int TotalD = 0;
        int TotalDD = 0;
        int totalDTP = 0;
        int totalDFP = 0;
        int TotalS = 0;
        int flag = 0;
        int stop1 = 0;
        int stop2 = 0;
        int stop3 = 0;
        int frame;
        int f;
        int no_acc;       //number of acceptor and start sites
        int no_don;       //number of donor sites
        int no_sta;
        int stats[14];
        int statsT[14];
        double s;
        double ss;
        double score;
        char CDS_arry[5000];
        int site_arry[100];
        char *temp;
        char seqName[100];
        int site = 0;
        int siteD = 0;
int siteS = 0;

double SnSum = 0.0;
double SpSum = 0.0;

int *mycdsarry, mycdstotal;
/*2-dimantional array fo holding the frame length**/
/* Frame[no_acc][no_doc];  */
int **Frame;
spSite *aCurrent, *dCurrent, *sCurrent, *pred;
spSite **a_d_arry;
int currentHiScore;
acc = (spSite*)malloc(sizeof(spSite));
don = (spSite*)malloc(sizeof(spSite));
if (don == NULL)
  printf("Donor structure maloc error\n");
aCurrent = acc;
dCurrent = don;
dCurrent->next = NULL;
aCurrent->next = NULL;
seq[0] = '\0';
T[0] = '\0';
p_t = 1.0;
p_f = 1.0;

at = 0;
ft = 0;
rewind(F_In);
fprintf(CdsOut, "\nGeneScout output:");
if(type == 1)
{
   fprintf(CdsOut, "\nFormat: Simple\n\n");
}
else
   fprintf(CdsOut, "\nFormat: Exon\n\n");
fprintf(CdsOut, "\nInput sequence:");
TotalSeq = 1;
for (n = 0; n < TotalSeq; n++)
{
        flag = 0;
        no_acc = 0;
        no_don = 0;
        no_sta = 0;
        memset(seq, '\0', 50000);
```

```
          memset(seqName, '\0', 100);
          if (fgets(T, MAXLENGTH, F_In) != NULL)
{
printf(CdsOut, "\n%s",T);
if (T[0] == '>')
{
    for(i = 0; i < strlen(T) - 2; i++)
        seqName[i] = T[i+1];
    fgets(T, MAXLENGTH, F_In);
    fprintf(CdsOut, "%s",T);
}
if (T[strlen(T) -1] == '\n')
    T[strlen(T) - 1] = '\0';
strcat(seq, T);
while (flag == 0)
{
    if ((fgets(T, MAXLENGTH, F_In) == NULL) ||
       (T[0] == '>'))
        flag = 1;
    else
    {
       fprintf(CdsOut, "%s",T);
       if (T[strlen(T) -1] == '\n')
          T[strlen(T) - 1] = '\0';
       strcat(seq, T);
    }
}
seqlen = strlen(seq);
fprintf(CdsOut,
        "\n\nInput sequence length: %i.", seqlen);
fprintf(CdsOut, "\n\nPredicted Gene Structure:\n");
thisSeqTotalD = 0;
thisSeqTotalS = 0;
k = 1;
kd = 0;
for (i=3; i < seqlen - 7; i++)
{
   /****ACCEPTOR DETECT***************/
   if((i>=15)&&(seq[i] == 'A') && (seq[i+1] == 'G'))
   {
      TotalD++;
      p = i-13;
      for (m = 0; m < amotifLen; m++)
      {
            accept[m] = seq[p++];
        }
        accept[amotifLen] = '\0';

        SiteClassif(accept, amotifLen, aHMM_T, aHMM_F,
        ACCEPTOR);
        score = p_t/p_f;
        if (score >= 0.1)
        {
           tmp = (spSite*)malloc(sizeof(spSite));
           if (tmp == NULL)
              printf ("tmp1 memory allocate error\n");
           tmp->kind = ACCEPTOR;
           tmp->bp = i + 3;
           tmp->score = score;
           tmp->nextStop = 0;
           tmp->next = NULL;
           tmp->next_site = NULL;
           aCurrent->next = tmp;
           aCurrent = aCurrent->next;
           tmp = NULL;
           no_acc++;
           thisSeqTotal++;
        }
        //printf("Still in the aceptor\n");

    }
    //    printf("\nGot in here now");
          /********** START DETECT***********/

    else if ((i >= 17)
          && (seq[i] == 'A') && (seq[i+1] == 'T')
          && (seq[i+2] == 'G'))
    {
           //   printf("\nGot in here now");
         TotalS++;
         p = i - 12;
         for (m = 0; m < smotifLen; m++)
         {
              start[m] = seq[p++];
         }
         start[smotifLen] = '\0';
         SiteClassif(start, smotifLen, sHMM_T, sHMM_F, START);
         if (p_f <= 0)
```

```
       {
          score = -1;
          printf("\n ERROR, p_f: %f", p_f);
       }
       score = p_t/p_f;
       if (score  >= 0.01)
       {
           tmp = (spSite*)malloc(sizeof(spSite));
           if (tmp == NULL)
              printf ("tmp2 memory allocate error\n");
           tmp->kind = START;
           tmp->bp = i + 1;
           tmp->score = score;
           tmp->nextStop = 0;
           tmp->next_site = NULL;
           aCurrent->next = tmp;
           aCurrent = aCurrent->next;
           aCurrent->next = NULL;
           tmp = NULL;
           no_acc++;
           thisSeqTotalS++;
       }
    }
    /****END FOR START DETECT*****/
    /********** DONOR DETECT ***********/

   else if ((seq[i] == 'G') && (seq[i+1] == 'T'))
   {
       TotalDD++;
       p = i-3;
       for (m = 0; m < 9; m++)
       {
          donor[m] = seq[p++];
       }
       donor[9] = '\0';
       SiteClassif(donor, dmotifLen, dHMM_T,
                              dHMM_F, DONOR);
       score = p_t/p_f;
       if (score >= 0.1)
       {
          tmp = (spSite*)malloc(sizeof(spSite));
          if (tmp == NULL)
              printf ("tmp1 memory allocate error\n");
          tmp->kind = DONOR;
```

```
                tmp->bp = i;
                tmp->score = score;
                tmp->next = NULL;
                tmp->nextStop = 0;
                tmp->next_site = NULL;
                dCurrent->next = tmp;
                dCurrent = dCurrent->next;
                tmp = NULL;

                no_don++;
                thisSeqTotalD++;
             }
          }
       }
       /***populate the index field in the linked lists****/
       aCurrent = acc->next;
       for (i = 0; i < no_acc; i++)
       {
          aCurrent->index = i;
          aCurrent->HiScore = 0;
          aCurrent->pred = NULL;
          aCurrent->startFound = 0;
          aCurrent = aCurrent->next;
       }
       dCurrent = don->next;
       for (i = 0; i < no_don; i++)
       {
          dCurrent->index = i;
          dCurrent->HiScore = 0;
          dCurrent->pred = NULL;
          dCurrent->startFound = 0;
          dCurrent = dCurrent->next;
       }
       a_d_arry = malloc((no_acc+no_don)*sizeof(spSite*));
       aCurrent = acc->next;
       dCurrent = don->next;
       for (i = 0; i < no_acc + no_don; i++)
       {
          if (dCurrent == NULL)
          {
             a_d_arry[i] = aCurrent;
             aCurrent = aCurrent->next;
          }
          else if (aCurrent == NULL)
```

```
        {
           a_d_arry[i] = dCurrent;
           dCurrent = dCurrent->next;
        }
        else if (aCurrent->bp <= dCurrent->bp)
        {
           a_d_arry[i] = aCurrent;
           aCurrent = aCurrent->next;
        }
        else if (dCurrent->bp < aCurrent->bp)
        {
           a_d_arry[i] = dCurrent;
           dCurrent = dCurrent->next;
        }
     }
/********PLACE FOR EXON DETECT********/
  Frame = malloc(no_acc*sizeof(int*));
  for (i = 0; i< no_acc; i++)
    Frame[i] = malloc(no_don*sizeof(int));
  if (Frame == NULL)
     printf("Seq %d, Mem problem\n", n+1);
  for (i=0; i<no_acc; i++)
    for (j=0; j<no_don; j++)
      Frame[i][j] = 0;
  aCurrent = acc->next;
  for (i = 0; i< no_acc; i++)
  {
    dCurrent = don->next;
    for (j = 0; j < no_don; j++)
    {
       if (aCurrent->bp < dCurrent->bp)
       {
          frame = 1;
          if (aCurrent->kind == START)
          {
             for (m = aCurrent->bp-1;
                     m+3 <= dCurrent->bp - 1; m=m+3)
             {
                if (((seq[m] == 'T') && (seq[m+1] == 'A') &&
                  (seq[m+2] == 'A')) || ((seq[m] == 'T') &&
                  (seq[m+1] == 'A') && (seq[m+2] == 'G')) ||
                  ((seq[m] == 'T') && (seq[m+1] == 'G') &&
                    (seq[m+2] == 'A')))
                {
```

```
                frame = 0;
                break;
            }
        }
        if (frame == 1)
            Frame[i][j] = dCurrent->bp - aCurrent->bp;
}
else if (aCurrent->kind == ACCEPTOR)/*acceptor*/
{  frame = 1;
   for(m = aCurrent->bp; m+3 <= dCurrent->bp - 1; m=m+3)
   {
      frame = 1;
      if (((seq[m] == 'T') && (seq[m+1] == 'A') &&
         (seq[m+2] == 'A')) || ((seq[m] == 'T') &&
          (seq[m+1] == 'A') && (seq[m+2] == 'G')) ||
          ((seq[m] == 'T') && (seq[m+1] == 'G') &&
            (seq[m+2] == 'A')))
      {
          frame = 0;
          break;
      }
   }
   if (frame != 1)
   {
      frame = 1;
      for(m = aCurrent->bp+1;
             m+3 <= dCurrent->bp - 1; m=m+3)
      {
         if(((seq[m] == 'T') && (seq[m+1] == 'A') &&
          (seq[m+2] == 'A')) || ((seq[m] == 'T') &&
          (seq[m+1] == 'A') && (seq[m+2] == 'G')) ||
          ((seq[m] == 'T') && (seq[m+1] == 'G') &&
            (seq[m+2] == 'A')))
         {
             frame = 0;
             break;
         }
      }
   }
   if (frame != 1)
   {
      frame = 1;
      for (m = aCurrent->bp+2;
            m+3 <= dCurrent->bp - 1; m=m+3)
      {
         if(((seq[m] == 'T') && (seq[m+1] == 'A') &&
          (seq[m+2] == 'A')) || ((seq[m] == 'T') &&
          (seq[m+1] == 'A') && (seq[m+2] == 'G')) ||
          ((seq[m] == 'T') && (seq[m+1] == 'G') &&
            (seq[m+2] == 'A')))
         {
             frame = 0;
             break;
         }
      }
   }
   if (frame == 1)
   {
     stop1 = stop2 = stop3 = 0;
     Frame[i][j] = dCurrent->bp - aCurrent->bp;
     /* Add STOP codon (nextStop) infor here */
     for(m = aCurrent->bp; m+3 <= seqlen; m = m+3)
     {
        if(((seq[m] == 'T') && (seq[m+1] == 'A') &&
          (seq[m+2] == 'A')) || ((seq[m] == 'T') &&
          (seq[m+1] == 'A') && (seq[m+2] == 'G')) ||
          ((seq[m] == 'T') && (seq[m+1] == 'G') &&
            (seq[m+2] == 'A')))
         {
           aCurrent->nextStop = m+3;
           break;
         }
      }
     for(m = aCurrent->bp; m+3 <= seqlen; m = m+3)
     {
       if(((seq[m+1] == 'T') && (seq[m+2] == 'A') &&
         (seq[m+3] == 'A')) || ((seq[m+1] == 'T') &&
         (seq[m+2] == 'A') && (seq[m+3] == 'G')) ||
         ((seq[m+1] == 'T') && (seq[m+2] == 'G') &&
           (seq[m+3] == 'A')))
         {
           if (m+1 > aCurrent->nextStop)
             aCurrent->nextStop = m + 4;
           break;
         }
     }
     for (m = aCurrent->bp; m+3 <= seqlen; m = m+3)
     {
```

```
                  if (((seq[m+2] == 'T') && (seq[m+3] == 'A') &&
                     (seq[m+4] == 'A')) || ((seq[m+2] == 'T') &&
                      (seq[m+3] == 'A') && (seq[m+4] == 'G')) ||
                      ((seq[m+2] == 'T') && (seq[m+3] == 'G') &&
                        (seq[m+4] == 'A')))
                     {
                       if (m+2 > aCurrent->nextStop)
                         aCurrent->nextStop = m + 5;
                       break;
                     }
                 }
               }
            }/*else acceptor*/

         }/*if(acc[i].bp < ..)*/
         if (dCurrent->next != NULL)
            dCurrent = dCurrent->next;
      }/*for(j = 0; )*/
      if (aCurrent != NULL)
         aCurrent = aCurrent->next;
   }/*for(i=0;..*/


/******** REAL GENE STRUCTURE SITES ******************/
/* Here asume that a gene always start at START codon */
dCurrent = don->next;
for (i = 0; i < no_don; i++)
{
    aCurrent = acc->next;
    for (q = 0; q < no_acc; q++)
    {
       if (aCurrent->bp >= dCurrent->bp)
          break;
       if (aCurrent->kind == START &&
          (Frame[aCurrent->index][dCurrent->index]
           > dCurrent->HiScore))
       {
          dCurrent->HiScore
          = Frame[aCurrent->index][dCurrent->index];
          dCurrent->startFound = 1;
          dCurrent->pred = aCurrent;
       }
       aCurrent = aCurrent->next;
    }
    dCurrent = dCurrent->next;
}
for (i = 0; i < no_acc + no_don; i++)
{
    if(a_d_arry[i] != NULL)
    {
       if ( a_d_arry[i]->kind == START)
       {
          //Already did as below befor
          //a_d_arry[i]->pred = NULL;
          //a_d_arry[i]->HiScore = 0;
       }
       else if (a_d_arry[i]->kind == DONOR
               && a_d_arry[i]->score > 0.5)
       {
          aCurrent = acc->next;
          currentHiScore = 0;
          pred = NULL;
          for (q = 0; q < no_acc; q++)
          {
            if((aCurrent->bp < a_d_arry[i]->bp)
               && (aCurrent->startFound == 1))
            {
              if(Frame[aCurrent->index][a_d_arry[i]->index] > 60 &&
                 a_d_arry[i]->HiScore < aCurrent->HiScore +
                 Frame[aCurrent->index][a_d_arry[i]->index])
              {
                 a_d_arry[i]->HiScore = aCurrent->HiScore +
                  Frame[aCurrent->index][a_d_arry[i]->index];
                 a_d_arry[i]->pred = aCurrent;
                 a_d_arry[i]->startFound = 1;
              }
            }
            aCurrent = aCurrent->next;
          }
       }
       else if ( a_d_arry[i]->kind == ACCEPTOR)
       {
          dCurrent = don->next;
          currentHiScore = 0;
          pred = NULL;
          for (q = 0; q < no_don; q++)
          {
            if((dCurrent->bp < a_d_arry[i]->bp)
                 && (dCurrent->startFound == 1))
              {
                 if (a_d_arry[i]->HiScore < dCurrent->HiScore)
                 {
                    a_d_arry[i]->HiScore = dCurrent->HiScore;
                    a_d_arry[i]->pred = dCurrent;
                    a_d_arry[i]->startFound = 1;
                 }
              }
              dCurrent = dCurrent->next;
           }
         }
      }
      else
         break;
}
rrent = acc->next;
currentHiScore = aCurrent->HiScore;
dCurrent = aCurrent;
while (aCurrent != NULL)
{
   if (aCurrent->kind == ACCEPTOR && aCurrent->nextStop <= 0)
      aCurrent->nextStop = aCurrent->bp;
   if (aCurrent->HiScore > currentHiScore)
   {
      currentHiScore = aCurrent->HiScore ;
      dCurrent = aCurrent;
   }
   aCurrent = aCurrent->next;
}
if(dCurrent->pred != NULL)
{
   mycdstotal = 0;
   while(dCurrent->pred != NULL)
   {
      mycdstotal++;
      dCurrent->pred->next_site = dCurrent;
      //finally aCurrent will be the first for START
      aCurrent = dCurrent->pred;
      dCurrent = dCurrent->pred;
   }
   mycdstotal = mycdstotal + 2;
   mycdsarry = malloc(mycdstotal*sizeof(int));
   i = 0;
```

```
while (aCurrent != NULL)
{
   mycdsarry[i++] = aCurrent->bp;
   //fprintf(CdsOut, "%d ",aCurrent->bp);
   dCurrent = aCurrent;
   aCurrent = aCurrent->next_site;
}
//fprintf(CdsOut, " %d\n",dCurrent->nextStop);
mycdsarry[i] = dCurrent->nextStop;
if(mycdsarry[i] < mycdsarry[i-1] + 30)
   i = i - 2;
if(type == 1)
{
   fprintf(CdsOut,
      "(The numbers below are the site positions on the sequence");
   fprintf(CdsOut,
      "\n for start, donor, acceptor, donor...)\n\n");
   fprintf(CdsOut," %s: ", seqName);
   for(j = 0; j <= i; j++)
   {
      fprintf(CdsOut, "%d ", mycdsarry[j]);
   }
   fprintf(CdsOut, "\n");
}
else
{
   seqlen = 0;   // reuse this var for coding regin length
   fprintf(CdsOut,
       "\nSequnce Name: %s", seqName);
   fprintf(CdsOut,
       "\nNumber of exons detected: %i\n", (i+1)/2);
   fprintf(CdsOut,
       "\n\tExon Number  Start At  End At  Length");
   fprintf(CdsOut,
       "\n\t-----------------------------------");
   m = 1;
   for(j = 0; j <= i-1; j=j+2 )
   {
      exonlen = mycdsarry[j+1] - mycdsarry[j] + 1;
      seqlen = seqlen + exonlen;
      fprintf(CdsOut, "\n\t     %i\t\t%i\t%i\t%i",
         m++, mycdsarry[j], mycdsarry[j+1], exonlen);
   }
   fprintf(CdsOut,
        "\n\nLenth of Coding Region: %i bp\n\n", seqlen);
   }
   free (mycdsarry);
}
else // (dCurrent->pred == NULL)
{
    printf("\nHere is a NULL, Seq #%d\n", n+1);
    //continue;
}
/******** FREE THE  LIST MEMORY *******/
for (i =0; i < no_acc; i++)
  free(Frame[i]);
free(Frame);

//free acc and don linked list for this seq
Current = acc->next;
while (aCurrent != NULL)
{
   tmp = aCurrent->next;
   aCurrent->next = NULL;
   free(aCurrent);
   aCurrent = tmp;
}
aCurrent = don->next;
while (aCurrent != NULL)
{
   tmp = aCurrent->next;
   aCurrent->next = NULL;
   free(aCurrent);
   aCurrent = tmp;
}

free(a_d_arry);
//free(mycdsarry);
aCurrent = acc;
dCurrent = don;

}/* if (fgets(T, ....)*/
}/*for (n = 0,; n < TotalSeq;..)*/
}/*end of func*/

void HmmInitialize (state *HMM, int motifLen)
{
      int i;
      for ( i = 0; i < motifLen; i++ )
      {
            HMM[i].A.bcnt = 0;
            HMM[i].A.toA.tcnt = 0;
            HMM[i].A.toA.tfreq = 0.0;
            HMM[i].A.toG.tcnt = 0;
            HMM[i].A.toG.tfreq = 0.0;
            HMM[i].A.toC.tcnt = 0;
            HMM[i].A.toC.tfreq = 0.0;
            HMM[i].A.toT.tcnt = 0;
            HMM[i].A.toT.tfreq = 0.0;

            HMM[i].G.bcnt = 0;
            HMM[i].G.toA.tcnt = 0;
            HMM[i].G.toA.tfreq = 0.0;
            HMM[i].G.toG.tcnt = 0;
            HMM[i].G.toG.tfreq = 0.0;
            HMM[i].G.toC.tcnt = 0;
            HMM[i].G.toC.tfreq = 0.0;
            HMM[i].G.toT.tcnt = 0;
            HMM[i].G.toT.tfreq = 0.0;

            HMM[i].C.bcnt = 0;
            HMM[i].C.toA.tcnt = 0;
            HMM[i].C.toA.tfreq = 0.0;
            HMM[i].C.toG.tcnt = 0;
            HMM[i].C.toG.tfreq = 0.0;
            HMM[i].C.toC.tcnt = 0;
            HMM[i].C.toC.tfreq = 0.0;
            HMM[i].C.toT.tcnt = 0;
            HMM[i].C.toT.tfreq = 0.0;

            HMM[i].T.bcnt = 0;
            HMM[i].T.toA.tcnt = 0;
            HMM[i].T.toA.tfreq = 0.0;
            HMM[i].T.toG.tcnt = 0;
            HMM[i].T.toC.tcnt = 0;
            HMM[i].T.toC.tfreq = 0.0;
            HMM[i].T.toT.tcnt = 0;
            HMM[i].T.toT.tfreq = 0.0;
      }
}
```

B.2 Sample Program For Functional Site Detection

```
/*****************************************************************
*                                                                *
*           (c) Copyright 2002                                   *
*           All rights reserved                                  *
*                                                                *
*           Program written by Michael M. Yin, Ph. D student     *
*           in the group of Professor Jason T.L. Wang            *
*           Department of Computer Science                       *
*           College of Computing Sciences                        *
*           New Jersey Institute of Technology                   *
*           University Heights, Newark, NJ 07102, USA            *
*                                                                *
*      This is a code example for functional site detection      *
*      and it may not be the up-to-date version.                 *
*      Programmer(s) makes no representations about the          *
*      suitability of this software for any purpose.             *
*      It is provided "as is" without express or implied         *
*      warranty.                                                 *
*                                                                *
*****************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct Trans
{
long tcnt; //num of trans--edges
double tfreq;  //frequence of this trans
} trans;

typedef struct Base
{
long bcnt;     //base count
double bfreq;   //freq for this base
trans toA, toG, toC, toT;  //edges out from this base
} base;

typedef struct State  //HMM state
{
base A, G, C, T;
} state;
```

```
typedef struct SiteData
{
       int seqNum;
       int siteNum;
       double P_t;
       double P_f;
       double score;
} siteData;
siteData seqData[2500];
state HMM_T[9];
state HMM_F[9];
#define motifLen 9
#define MAXLENGTH 50000
#define MaxSeq 2000

int TotalSeq;
/*total number of false donor sites extracted*/
int totalF = 0;
int totalT = 0;
char seq[MAXLENGTH];


double pA, pG, pC, pT,
      fpA, fpG, fpC, fpT;
//double p_t = 0.0;
//double p_f = 0.0;
//double pcp_t = 0.0;
//double pcp_f = 0.0;

double p_t, p_f;

FILE *F_In, *F_Out, *CDS_In;

void HmmInitialize (state *HMM)
{
int i;
for ( i = 0; i < motifLen; i++ )
{
HMM[i].A.bcnt = 0;
HMM[i].A.toA.tcnt = 0;
HMM[i].A.toA.tfreq = 0.0;
HMM[i].A.toG.tcnt = 0;
HMM[i].A.toG.tfreq = 0.0;
HMM[i].A.toC.tcnt = 0;
HMM[i].A.toC.tfreq = 0.0;
HMM[i].A.toT.tcnt = 0;
HMM[i].A.toT.tfreq = 0.0;

HMM[i].G.bcnt = 0;
HMM[i].G.toA.tcnt = 0;
HMM[i].G.toA.tfreq = 0.0;
HMM[i].G.toG.tcnt = 0;
HMM[i].G.toG.tfreq = 0.0;
HMM[i].G.toC.tcnt = 0;
HMM[i].G.toC.tfreq = 0.0;
HMM[i].G.toT.tcnt = 0;
HMM[i].G.toT.tfreq = 0.0;

HMM[i].C.bcnt = 0;
HMM[i].C.toA.tcnt = 0;
HMM[i].C.toA.tfreq = 0.0;
HMM[i].C.toG.tcnt = 0;
HMM[i].C.toG.tfreq = 0.0;
HMM[i].C.toC.tcnt = 0;
HMM[i].C.toC.tfreq = 0.0;
HMM[i].C.toT.tcnt = 0;
HMM[i].C.toT.tfreq = 0.0;

HMM[i].T.bcnt = 0;
HMM[i].T.toA.tcnt = 0;
HMM[i].T.toA.tfreq = 0.0;
HMM[i].T.toG.tcnt = 0;
HMM[i].T.toC.tcnt = 0;
HMM[i].T.toC.tfreq = 0.0;
HMM[i].T.toT.tcnt = 0;
HMM[i].T.toT.tfreq = 0.0;
}
}


int HmmTrainBase(char *seq, state *HMM)
{
int len;
int i;
char b, nextb;
len = strlen(seq);
/*printf("donor Len : %i\n", len);*/
if (len != motifLen)
```

```
printf("donor lenth is wrong: %i\n", len);
/*return 1;*/

for (i = 0; i < len; i++)
{

b = seq[i];
switch (b)
{
case 'A':
HMM[i].A.bcnt++;
if (i < 8)
{
nextb = seq[i+1];
switch (nextb)
{
case 'A':
HMM[i].A.toA.tcnt++;
break;
case 'G':
HMM[i].A.toG.tcnt++;
break;
case 'C':
HMM[i].A.toC.tcnt++;
break;
case 'T':
HMM[i].A.toT.tcnt++;
break;

default:
printf("Train error: %s\n", seq);
}
}
break;

case 'G':
HMM[i].G.bcnt++;
if (i < 8)
{
nextb = seq[i+1];
switch (nextb)
{
case 'A':
HMM[i].G.toA.tcnt++;
break;
case 'G':
HMM[i].G.toG.tcnt++;
break;
case 'C':
HMM[i].G.toC.tcnt++;
break;
case 'T':
HMM[i].G.toT.tcnt++;
break;

default:
printf("Train error %s\n", seq);
}
}
break;

case 'C':
HMM[i].C.bcnt++;
if (i < 8)
{
nextb = seq[i+1];
switch (nextb)
{
case 'A':
HMM[i].C.toA.tcnt++;
break;
case 'G':
HMM[i].C.toG.tcnt++;
break;
case 'C':
HMM[i].C.toC.tcnt++;
break;
case 'T':
HMM[i].C.toT.tcnt++;
break;

default:
printf("Train error %s\n", seq);
}
}
break;

case 'T':
```

```
HMM[i].T.bcnt++;
if (i < 8)
{
nextb = seq[i+1];
switch (nextb)
{
case 'A':
HMM[i].T.toA.tcnt++;
break;
case 'G':
HMM[i].T.toG.tcnt++;
break;
case 'C':
HMM[i].T.toC.tcnt++;
break;
case 'T':
HMM[i].T.toT.tcnt++;
break;

default:
printf("Train error %s\n", seq);
}
}
break;
}
}

return 0;
}

int HmmTrainFreq(int totalSeq, state *HMM)
{
int num = totalSeq;
int i;

for (i = 0; i < motifLen; i++)
{
HMM[i].A.bfreq = (double)HMM[i].A.bcnt/num;
HMM[i].A.toA.tfreq = (double)HMM[i].A.toA.tcnt / num;
        printf ("\n%i:\t%d\t%d\t%f", i,
                  HMM[i].A.toA.tcnt,num,HMM[i].A.toA.tfreq);
HMM[i].A.toG.tfreq = (double)HMM[i].A.toG.tcnt / num;
HMM[i].A.toC.tfreq = (double)HMM[i].A.toC.tcnt / num;
HMM[i].A.toT.tfreq = (double)HMM[i].A.toT.tcnt / num;
```

```
HMM[i].G.bfreq = (double)HMM[i].G.bcnt/num;
HMM[i].G.toA.tfreq = (double)HMM[i].G.toA.tcnt / num;
HMM[i].G.toG.tfreq = (double)HMM[i].G.toG.tcnt / num;
HMM[i].G.toC.tfreq = (double)HMM[i].G.toC.tcnt / num;
HMM[i].G.toT.tfreq = (double)HMM[i].G.toT.tcnt / num;

HMM[i].C.bfreq = (double)HMM[i].C.bcnt/num;
HMM[i].C.toA.tfreq = (double)HMM[i].C.toA.tcnt / num;
HMM[i].C.toG.tfreq = (double)HMM[i].C.toG.tcnt / num;
HMM[i].C.toC.tfreq = (double)HMM[i].C.toC.tcnt / num;
HMM[i].C.toT.tfreq = (double)HMM[i].C.toT.tcnt / num;

HMM[i].T.bfreq = (double)HMM[i].T.bcnt/num;
HMM[i].T.toA.tfreq = (double)HMM[i].T.toA.tcnt / num;
HMM[i].T.toG.tfreq = (double)HMM[i].T.toG.tcnt / num;
HMM[i].T.toC.tfreq = (double)HMM[i].T.toC.tcnt / num;
HMM[i].T.toT.tfreq = (double)HMM[i].T.toT.tcnt / num;


}
return 0;
}

void BaseFreq_T(int totalSeq)
{
int i;
long Anum, Gnum, Cnum, Tnum;


long totalB = totalSeq * motifLen;

Anum = Gnum = Cnum = Tnum = 0;

for (i = 0; i < motifLen; i++)
{
Anum += HMM_T[i].A.bcnt;
Gnum += HMM_T[i].G.bcnt;
Cnum += HMM_T[i].C.bcnt;
Tnum += HMM_T[i].T.bcnt;
        /*
                Anum += HMM_F[i].A.bcnt;
                Gnum += HMM_F[i].G.bcnt;
                Cnum += HMM_F[i].C.bcnt;
                Tnum += HMM_F[i].T.bcnt;
        */
}

pA = (double)Anum/totalB;
pG = (double)Gnum/totalB;
pC = (double)Cnum/totalB;
 pT = (double)Tnum/totalB;
}


void BaseFreq_F(int totalSeq)
{
int i;
int Anum, Gnum, Cnum, Tnum;


int totalB = totalSeq * motifLen;

Anum = Gnum = Cnum = Tnum = 0;

for (i = 0; i < motifLen; i++)
{
Anum += HMM_F[i].A.bcnt;
Gnum += HMM_F[i].G.bcnt;
Cnum += HMM_F[i].C.bcnt;
Tnum += HMM_F[i].T.bcnt;
}

        fpA = (double)Anum/totalB;
        fpG = (double)Gnum/totalB;
        fpC = (double)Cnum/totalB;
        fpT = (double)Tnum/totalB;
}


/*Input sequences from a file*/
void HMM_T_Train(FILE *F_In, FILE *CDS_In)
{
int i, j , n, p, m,
        seqlen;  /*length of input seq*/
        int flag = 0;
        int mostSite = 0;
        int numSite = 0;
int totalDonor = 0;
        int site = 0;
char donor[10];
        char *temp;
        int Nbase;
 char seq[MAXLENGTH], T[100], ch;
         char CDS_arry[5000];
         int site_arry[100];
         seq[0] = '\0';
         T[0] = '\0';

 for (TotalSeq =0; ;)
 {
if ((fgets(T, MAXLENGTH, F_In)) == NULL)
break;
if (T[0] == '>')
TotalSeq++;
}
 /****/
 printf("\nTotalSeq: %i\n", TotalSeq);
if (TotalSeq > MaxSeq)
{
printf("\n%s\n", "Error: Too many sequences.");
exit(0);
}
rewind(F_In);
        for (n = 0; n < TotalSeq; n++)
{
           /*This is for the test*/
           //if (n < 57)
           //   continue;
           memset(seq, '\0', 50000);
           flag = 0;
           if (fgets(T, MAXLENGTH, F_In) != NULL)
   {
if (T[0] == '>')  /*for the first gene in the file*/
{
             /*skip the name line and get the next line*/
                     fgets(T, MAXLENGTH, F_In);
                }
                /*fgets put '\n' at the end of T */
                if (T[strlen(T) - 1] == '\n')
                    T[strlen(T) - 1] = '\0';
                strcat(seq, T);
        while(flag == 0)
        {
             if ((fgets(T, MAXLENGTH, F_In) == NULL) ||
                 (T[0] == '>'))
                 flag = 1;
             else
             {
                  if (T[strlen(T) - 1] == '\n')
                  T[strlen(T) - 1] = '\0';
                  strcat(seq, T);
             }
        }
  } /*if (fgets...)*/
 /*we got the sequence */
 /*get the CDS data from CDS.tbl file*/
 for (i = 0; i < 100; i++)
 {
     site_arry[i] = 0;
 }
 i = 0;
 fgets(CDS_arry, 5000, CDS_In);
 if ((n >=  508) && (n <= 569))
    continue;
 temp = strtok(CDS_arry, " ,\n");
 /*skip the gene name and get the site num*/
 temp = strtok(NULL, " ,");
 while (temp != NULL)
 {
    site_arry[i++] = atoi(temp);
    temp = strtok(NULL, " ,\n");
 }
 i = 0;
 site = 0;
/*the last one in the site_arry is the STOP codon*/
 while ( ((site =  site_arry[++i]) != 0)
                   && (site_arry[i+1] != 0))
 {
    i = i + 1; /*skip the acceptor site*/

    p = site - 3;
    for (m = 0; m < motifLen; m++)
    {
        donor[m] = seq[p++];
        if ((m == 4) && donor[m] != 'T')
```

```
            printf ("sequenc # %i error\n", n);
        }
        donor[9] = '\0';
        HmmTrainBase(donor, HMM_T);
        totalDonor++;
  }/*while ((site...*/

 /*for training  the HMM_F**/
 seqlen = strlen(seq);
 i = 1;
 flag = site_arry[i] - 1;
 numSite = 0; //number of sites in this seq
 for (j = 3; j < seqlen - 6; j++)
 {
    if (j == flag)
    {
       i = i + 2;
       numSite++;
       //printf("\nflag= %d", flag+1);
       flag = site_arry[i] - 1;
       continue;
    }
    if ((seq[j] == 'G') && (seq[j+1] == 'T'))
    {
       p = j - 3;
       Nbase = 0;
       for (m = 0; m < 9; m++)
       {
          if (seq[p] == 'N')
             Nbase = 1;  //base is 'N'
          donor[m] = seq[p++];
       }
      donor[9] = '\0';
      if (Nbase == 0)
      {
          HmmTrainBase(donor, HMM_F);
          numSite++;
          totalF++;
      }
    }
 }
 if (numSite > mostSite)
    mostSite = numSite;
    }/*for...*/

    /*Has to be Total Donor for the first argument)*/
    HmmTrainFreq(totalDonor, HMM_T);
    BaseFreq_T(totalDonor);

    HmmTrainFreq(totalF, HMM_F);
    BaseFreq_F(totalF);
    printf("\nTrain total f: %d, T: %d, mostSite: %d", totalF,
          totalDonor, mostSite);

 }

/*
 *  Train the false donor module
 */
int WriteDataToFile(char *filename, state *HMM, int flag)
{
int i;
if ((F_Out = fopen(filename, "w")) == NULL)
{
printf("\nError: open %s file", filename);
return 1;
}

for (i = 0; i < motifLen; i++)
{
fprintf(F_Out, "\n\nState %i", i+1);
fprintf(F_Out, "\nA Count:\t%l", HMM[i].A.bcnt);
fprintf(F_Out, "\nA Freq:\t%f", HMM[i].A.bfreq);
fprintf(F_Out,"\n\tA->A:\t%f", HMM[i].A.toA.tfreq);
fprintf(F_Out,"\n\tA->G:\t%f", HMM[i].A.toG.tfreq);
fprintf(F_Out,"\n\tA->C:\t%f", HMM[i].A.toC.tfreq);
fprintf(F_Out,"\n\tA->T:\t%f", HMM[i].A.toT.tfreq);


fprintf(F_Out,"\nG Freq:\t%f", HMM[i].G.bfreq);
fprintf(F_Out,"\n\tG->A:\t%f", HMM[i].G.toA.tfreq);
fprintf(F_Out,"\n\tG->G:\t%f", HMM[i].G.toG.tfreq);
fprintf(F_Out,"\n\tG->C:\t%f", HMM[i].G.toC.tfreq);
fprintf(F_Out,"\n\tG->T:\t%f", HMM[i].G.toT.tfreq);

fprintf(F_Out,"\nC Freq:\t%f", HMM[i].C.bfreq);
fprintf(F_Out, "\n\tC->A:\t%f", HMM[i].C.toA.tfreq);
fprintf(F_Out, "\n\tC->G:\t%f", HMM[i].C.toG.tfreq);
fprintf(F_Out, "\n\tC->C:\t%f", HMM[i].C.toC.tfreq);
fprintf(F_Out, "\n\tC->T:\t%f", HMM[i].C.toT.tfreq);

fprintf(F_Out,"\nT Freq:\t%f", HMM[i].T.bfreq);
fprintf(F_Out,"\n\tT->A:\t%f", HMM[i].T.toA.tfreq);
fprintf(F_Out,"\n\tT->G:\t%f", HMM[i].T.toG.tfreq);
fprintf(F_Out,"\n\tT->C:\t%f", HMM[i].T.toC.tfreq);
fprintf(F_Out,"\n\tT->T:\t%f", HMM[i].T.toT.tfreq);
}

fprintf(F_Out, "\n\nBase frequency:\n");
if (flag == 1)
{
fprintf(F_Out, "\np(A):\t%f", pA);
fprintf(F_Out, "\np(G):\t%f", pG);
fprintf(F_Out, "\np(C):\t%f", pC);
fprintf(F_Out, "\np(T):\t%f", pT);
}
else
{
fprintf(F_Out, "\np(A):\t%f", fpA);
fprintf(F_Out, "\np(G):\t%f", fpG);
fprintf(F_Out, "\np(C):\t%f", fpC);
fprintf(F_Out, "\np(T):\t%f", fpT);
}



fclose(F_Out);
}
/**********************/
void DonorClassif(char *d)
{
int i;
char ch1, ch2;
// double p_up;
p_t = p_f = 1.0;
for (i = 0; i < motifLen; i++)
{
ch1 = d[i];
ch2 = d[i+1];
switch (ch1)
{
```

```
case 'A':
if (i == motifLen-1)
{
p_t = p_t/pA;
p_f = p_f/fpA;
break;
}

switch (ch2)
{
case 'A':
//printf("A.toA freq: %f\n", HMM_T[i].A.toA.tfreq);
//printf("A.toA freq F: %f\n", HMM_F[i].A.toA.tfreq);
p_t *=HMM_T[i].A.toA.tfreq/pA;
p_f *=HMM_F[i].A.toA.tfreq/fpA;
//printf("p_t: %f\tp_f: %f\n", p_t, p_f);
break;
case 'G':
p_t *=HMM_T[i].A.toG.tfreq/pA;
p_f *=HMM_F[i].A.toG.tfreq/fpA;
break;
case 'C':
p_t *=HMM_T[i].A.toC.tfreq/pA;
p_f *=HMM_F[i].A.toC.tfreq/fpA;
break;
case 'T':
p_t *=HMM_T[i].A.toT.tfreq/pA;
p_f *=HMM_F[i].A.toT.tfreq/fpA;
break;

}
break;
case 'G':
if (i == motifLen-1)
{
p_t = p_t/pG;
p_f = p_f/fpG;
break;
}
switch (ch2)
{
case 'A':
p_t *=HMM_T[i].G.toA.tfreq/pG;
p_f *=HMM_F[i].G.toA.tfreq/fpG;
break;
case 'G':
p_t *=HMM_T[i].G.toG.tfreq/pG;
p_f *=HMM_F[i].G.toG.tfreq/fpG;
break;
case 'C':
p_t *=HMM_T[i].G.toC.tfreq/pG;
p_f *=HMM_F[i].G.toC.tfreq/fpG;
break;
case 'T':
p_t *=HMM_T[i].G.toT.tfreq/pG;
p_f *=HMM_F[i].G.toT.tfreq/fpG;
break;

}
break;
case 'C':
if (i == motifLen-1)
{
p_t = p_t/pC;
p_f = p_f/fpC;
break;
}
switch (ch2)
{
case 'A':
p_t *=HMM_T[i].C.toA.tfreq/pC;
p_f *=HMM_F[i].C.toA.tfreq/fpC;
break;
case 'G':
p_t *=HMM_T[i].C.toG.tfreq/pC;
p_f *=HMM_F[i].C.toG.tfreq/fpC;
break;
case 'C':
p_t *=HMM_T[i].C.toC.tfreq/pC;
p_f *=HMM_F[i].C.toC.tfreq/fpC;
break;
case 'T':
p_t *=HMM_T[i].C.toT.tfreq/pC;
p_f *=HMM_F[i].C.toT.tfreq/fpC;
break;

}
break;
```

```
case 'T':
if (i == motifLen-1)
{
p_t = p_t/pT;
p_f = p_f/fpT;
break;
}
switch (ch2)
{
case 'A':
p_t *=HMM_T[i].T.toA.tfreq/pT;
p_f *=HMM_F[i].T.toA.tfreq/fpT;
break;
case 'G':
p_t *=HMM_T[i].T.toG.tfreq/pT;
p_f *=HMM_F[i].T.toG.tfreq/fpT;
break;
case 'C':
p_t *=HMM_T[i].T.toC.tfreq/pT;
p_f *=HMM_F[i].T.toC.tfreq/fpT;
break;
case 'T':
p_t *=HMM_T[i].T.toT.tfreq/pT;
p_f *=HMM_F[i].T.toT.tfreq/fpT;
break;

}
break;
}
}
}

/********************/
void SiteSort(int);
void DonorDetect(FILE *F_In, FILE* F_Out, FILE *CDS_Out, FILE *CDS_In)
{
     int i,j,k,n,p,m,
     seqlen; /*length of input seq*/
     int thisSeqTotal;
     char donor[10];
     char seq[MAXLENGTH],
          T[100];
     char ch;
     int TotalD = 0;
int flag = 0;
int frame;
int f;
char CDS_arry[5000];
int site_arry[100];
char *temp;
char seqName[100];
int site = 0;
seq[0] = '\0';
T[0] = '\0';
seqName[0] = '\0';
p_t = 1.0;
p_f = 1.0;
rewind(F_In);
rewind(CDS_In);
for (n = 0; n < TotalSeq; n++)
{
   flag = 0;
   memset(seq, '\0', 50000);
   if (fgets(T, MAXLENGTH, F_In) != NULL)
   {
      if (T[0] == '>')
      {
          /*skip the name line and get the next line*/
          fgets(T, MAXLENGTH, F_In);
      }
      if (T[strlen(T) -1] == '\n')
           T[strlen(T) - 1] = '\0';
      strcat(seq, T);
      while (flag == 0)
      {
            if ((fgets(T, MAXLENGTH, F_In) == NULL) ||
                 (T[0] == '>'))
                flag = 1;
            else
            {
                 if (T[strlen(T) -1] == '\n')
                 T[strlen(T) - 1] = '\0';
                 strcat(seq, T);
            }

      }
      /*we got the sequence*/
      seqlen = strlen(seq);
```

```
for (i = 0; i <100; i++)
{
    site_arry[i] = 0;
}
fgets(CDS_arry, 5000, CDS_In);

if(n < 508)
   continue;
if(n > 569)
   break;
temp = strtok(CDS_arry, " ,\n");
strcpy(seqName, temp);
temp = strtok(NULL, "  ,\n");
i = 0;

while (temp != NULL)
{
    site_arry[i++] = atoi(temp);
    temp = strtok(NULL, " ,\n");
}
thisSeqTotal = 0;
site = 0;
k = 0;
site = site_arry[++k];
for (i=3; i < seqlen - 6; i++)
{
if ((seq[i] == 'G') && (seq[i+1] == 'T'))
{
         TotalD++;
 p = i-3;
 for (m = 0; m < 9; m++)
 {
    donor[m] = seq[p++];
         }
          donor[9] = '\0';
          DonorClassif(donor);
          seqData[thisSeqTotal].seqNum = n + 1;
          seqData[thisSeqTotal].siteNum = i;
          seqData[thisSeqTotal].P_t = p_t;
          seqData[thisSeqTotal].P_f = p_f;
          seqData[thisSeqTotal].score = p_t/p_f;
          thisSeqTotal++;
}
//Sort the seq scores
```

```
            site = site_arry[++m];
            SiteSort(thisSeqTotal);
            for (j = 0; j < thisSeqTotal; j++)
            {
                 fprintf(F_Out, "\n%d\tSeq #%d\tSite
                  #%d\t%f\t%f\t%f", j+1, seqData[j].seqNum,
                  seqData[j].siteNum, seqData[j].P_t, seqData[j].P_f,
                  seqData[j].score);
            }
        }
    }
    fprintf(F_Out, "\n\nTotal Donor detected: %d", TotalD);

    printf("Total donor detected: %i\n", TotalD);
}


main()
{
char *donor_t = "DNASequences.fasta";
char *donor_f = "donofals.acp";
char *dataT_file = "datatru9.508_569.txt";
char *dataF_file = "datafal9.508_569.txt";
char *donordet = "DNASequences.fasta";
char *detedata = "detedat9.508_569";
        char *cds_tbl = "CDS.tbl";
        char *cds_out = "cdsOut9.508_569";
        FILE * CDS_Out;
HmmInitialize(HMM_T);
HmmInitialize(HMM_F);

if ((F_In = fopen (donor_t, "r")) == NULL)
{
      printf("ERROR: %s file open for DNA Sequences.fasta.\n", donor_t);
      exit(0);
}
if ((CDS_In = fopen (cds_tbl, "r")) == NULL)
{
      printf("ERROR: %s file open for CDS_In.\n", cds_tbl);
      exit(0);
}
HMM_T_Train(F_In, CDS_In);

WriteDataToFile(dataT_file, HMM_T, 1);
```

```
fclose(F_In);
WriteDataToFile(dataF_file, HMM_F, 0);

if ((F_In = fopen (donordet, "r")) == NULL)
{
        printf("ERROR: %s file open.\n", donordet);
        exit(0);
}

if ((F_Out = fopen (detedata, "w")) == NULL)
{
        printf("ERROR: %s file open.\n", detedata);
         exit(0);
}
if ((CDS_Out = fopen (cds_out, "w")) == NULL)
{
        printf("ERROR: %s file open.\n", cds_out);
        exit(0);
}

DonorDetect(F_In, F_Out, CDS_Out, CDS_In);

fclose(F_In);
fclose(F_Out);
        fclose(CDS_In);
        return 0;
}

//Selection Sort
void SiteSort(int totalSite)
{
    int i, j, hiIndex;
    double hiScore;
    siteData sd;
    for (i = 0; i < totalSite - 2; i++)
    {
        hiIndex = i;
        hiScore = seqData[i].score;
        for (j = i + 1; j <totalSite - 1; j++)
        {
           if (seqData[j].score > hiScore)
           {
               hiScore = seqData[j].score;
               hiIndex = j;
```

```
            }
        }
        //swap
        sd.seqNum = seqData[i].seqNum;
        sd.siteNum = seqData[i].siteNum;
        sd.P_t = seqData[i].P_t;
        sd.P_f = seqData[i].P_f;
        sd.score = seqData[i].score;

        seqData[i].seqNum = seqData[hiIndex].seqNum;
        seqData[i].siteNum = seqData[hiIndex].siteNum;
        seqData[i].P_t = seqData[hiIndex].P_t;
        seqData[i].P_f = seqData[hiIndex].P_f;
        seqData[i].score = seqData[hiIndex].score;

        seqData[hiIndex].seqNum = sd.seqNum;
        seqData[hiIndex].siteNum = sd.siteNum;
        seqData[hiIndex].P_t = sd.P_t;
        seqData[hiIndex].P_f = sd.P_f;
        seqData[hiIndex].score = sd.score;
    }
}
```

REFERENCES

1. B. Alberts, D. Bray, J. Lewis, M. Raff, K. Roberts, and J. D. Watson, *Molecular Biology of the Cell*, 3rd ed. Garland Publishing, Inc., New York and London, 1989.

2. S. Audic and J. Claverie, "Detection of eukaryotic promoters using Markov transition matrices," *Computers and Chemistry*, 21:223–227, 1997.

3. T. L. Bailey, M. E. Baker, and C. Elkan, "An artificial intelligence approach to motif discovery in protein sequences: Application to steroid dehydrogenases," *J. Steroid Biochemistry*, 62(1):29–44, 1997.

4. J. J. W. Baker and G. E. Allen, *The Study of Biology*, 4th ed. Addison-Wesley Publishing Company, Inc., 1982.

5. M. Borodovsky and J. McIninch, "GENMARK: Parallel gene recognition for both DNA strands," *Computers and Chemistry*, 17:123–133, 1993.

6. C. Burge and S. Karlin, "Prediction of complete gene structures in human genomic DNA," *J. Mol. Biol.*, 268:78–94, 1997.

7. M. Burset and R. Guigo, "Evaluation of gene structure prediction programs," *Genomics*, 34(3):353–367, 1996.

8. J. Claverie, O. Poirot, and F. Lopez, "The difficulty of identifying genes in anonymous vertebrate sequences," *Computers and Chemistry*, 21(4):203–214, 1997.

9. W. H. E. Day and F. R. McMorris, "Critical compatrison of consensus methods for molecular sequences," *Nucleic Acides Research*, 20, 1093–1099, 1992.

10. S. Dong and G. D. Stormo, "Gene structure prediction by linguistic methods," *Genomics*, 23:540–551, 1994.

11. A. Fedorov, G. Suboch, and L. Fedorova, "Analysis of nonuniformity in intron phase distribution," *Nucleic Acids Research*, 20:2553–2557, 1992.

12. M. S. Gelfand, "Prediction of function in DNA sequence analysis," *Journal of Computational Biology*, 2(1):87–115, 1995.

13. R. Guigo, "Computational gene identification: An open problem," *Computers and Chemistry*, 21(4):215–222, 1997.

14. R. Guigo, S. Knudsen, N. Drake, and T. F. Smith, "Prediction of gene structure," *Journal of Molecular Biology*, 226:141–157, 1995.

15. J. Han and M. Kamber, *Data Mining: Concepts and Techniques.* Morgan Kaufmann Publishers, San Francisco, California, 2000.

16. 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.

17. N. D. Herman and T. D. Schneider, "High information conservation implies that at least three proteins bind independently to F Plasmid *incD* repeats," *Journal of Bacteriology*, 174, 3558–3560, 1992.

18. J. D. Hawkins, "A survey on intron and exon lengths," *Nucleic Acids Research*, 11:9893–9905, 1988.

19. 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.

20. G. B. Hutchinson and M. R. Hayden, "The prediction of exons through an analysis of spliceable open reading frames," *Nucleic Acids Research*, 20:3453–3462, 1992.

21. K. F. Lee, *Automatic Speech Recognition: The Development of the SPHINX System.* Kluwer Academic, Boston, Massachusetts, 1989.

22. Y. Lida, "DNA sequences and multivariate statistical analysis . Categorical discrimination approach to 5' splice site signals of mmRNA precursors in higher eukaryotes' genes," *Comput. Appl. Biosci*, 3:93–98, 1987.

23. A. V. Lukashin and M. Borodovsky, "GeneMark.hmm: New solutions for gene finding," *Nucleic Acids Research*, 26(4):1107–1115, 1998.

24. Q. Ma and J. T. L. Wang, "Biological data mining using Bayesian neural networks: A case study," *International Journal on Artificial Intelligence Tools*, 8(4):433–451, 1999.

25. Q. Ma, J. T. L. Wang, D. Shasha, and C. H. Wu, "DNA sequence classification via an expectation maximization algorithm and neural networks: A case study," *IEEE Transactions on Systems, Man, and Cybernetics*, Part C: Applications and Reviews, 31(4):468–475, 2001.

26. L. Milanesi, N. A. Kolchanov, I. B. Rogozin, I. V. Ischenlo, A. E. Kel, Y. L. Orlov, M. P. Ponomarenko, and P. Vezzoni, "GenViewer: A computing tool for protein coding regions prediction in nucleotide sequences," *Proceedings of the 2nd International Congress on Bioinformatics, Supercomputing and Complex Genome Analysis*, World Scientific, Singapore, pages 573–587, 1993.

27. P. P. Papp, D. K. Chattoraj and T. D. Schneider, "Information analysis of sequences that bind the replication initiator *repA*," *Jounal of Molecular Biology*, 233, 219–230, 1993.

28. M. A. Roytberg, T. V. Astakhova, and M. S. Gelfand, "Combinatorial approaches to gene recognition," *Computers Chem.*, 21(4):229–235, 1997.

29. S. L. Salzberg, "A method for identifying splice sites and translational start sites in eukaryotic mRNA," *Computer Applications in the Biosciences*, 13(4):365–376, 1997.

30. S. L. Salzberg, M. Pertea, A. Delcher, M. J. Gardner, and H. Tetterlin, "Interpolated Markov models for eukaryotic gene finding," *Genomics*, 59:24–31, 1999.

31. T. D. Schneider and R. M. Stephens, "Sequence logos: a new way to display consensus structures," *Nucleic Acids Research*, 18, 6097–6100, 1990.

32. E. E. Snyder and G. D. Stormo, "Identification of coding regions in genomic DNA sequences: An application of dynamic programming and neural networks," *Nucleic Acids Research*, 21:607–613, 1993.

33. V. V. Solovyev, A. A. Salamov, and C. B. Lawrence, "Prediction of internal exons by oligonucleotide composition and discriminant analysis of spliceable open reading frames," *Nucleic Acids Research*, 22:5156–5163, 1994.

34. R. Staden, "Computer methods to locate singals in nucleic acid sequences," *Nucleic Acids Research*, 12, 505–519, 1984.

35. A. Thomas and M. H. Skolnick, "A probabilistic model for detecting coding regions in DNA sequences," *IMA J. Math. Appl. Med. Biol.*, 11:149–160, 1992.

36. E. C. Uberbacher, J. R. Einstein, X. Fuan, and R. J. Mural, "Gene recognition and assembly in the GRAIL system: Progress and challenges," *Proceedings of the 2nd International Congress on Bioinformatics, Supercomputing and Complex Genome Analysis*, pages 465–476, World Scientific, Singapore, 1993.

37. J. T. L. Wang, Q. Ma, D. Shasha, and C. H. Wu, "Application of neural networks to biological data mining: A case study in protein sequence classification," *Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 305–309, Boston, Massachusetts, August 2000.

38. J. T. L. Wang, Q. Ma, D. Shasha, and C. H. Wu, "New techniques for extracting features from protein sequences," *IBM Systems Journal*, 40(2):426–441, 2001.

39. J. T. L. Wang, T. G. Marr, D. Shasha, B. A. Shapiro, G. W. Chirn, and T. Y. Lee, "Complementary classification approaches for protein sequences," *Protein Engineering*, 9(5):381–386, 1996.

40. J. T. L. Wang, S. Rozen, B. A. Shapiro, D. Shasha, Z. Wang, and M. Yin, "New techniques for DNA sequence classification," *Journal of Computational Biology*, 6(2):209–218, 1999.

41. J. T. L. Wang, B. A. Shapiro, and D. Shasha, editors, *Pattern Discovery in Biomolecular Data: Tools, Techniques and Applications.* Oxford University Press, New York, New York, 1999.

42. J. T. L. Wang, C. H. Wu, and P. P. Wang, editors, *Computational Biology and Genome Informatics.* World Scientific Publishers, Singapore, 2002.

43. Y. Xu, R. J. Mural, and E. C. Uberbacher, "Constructing gene models from accurately predicted exons: An application of dynamic programming," *Comput. Appl. Biosci.*, 10:613–623, 1994.

44. M. M. Yin, "Algorithms and tools for splicing junction donor recognition in genomic DNA sequences," M.S. Thesis, Department of Computer Science, New Jersey Institute of Technology, 1997.

45. M. Yin and J. T. L. Wang, "Algorithms for splicing junction donor recognition in genomic DNA sequences," *Proceedings of the IEEE International Joint Symposia on Intelligence and Systems*, pages 169–176, Rockville, Maryland, May 1998.

46. M. M. Yin and J. T. L. Wang, "Application of hidden Markov models to gene prediction in DNA," *Proceedings of the IEEE International Conference on Information, Intelligence and Systems*, pages 40–47, Bethesda, Maryland, November 1999.

47. M. M. Yin and J. T. L. Wang, "Recognizing splicing junction acceptors in eukaryotic genes using hidden Markov models and machine learning methods," *Proceedings of the 5th Joint Conference on Information Sciences*, pages 786–789, Atlantic City, New Jersey, February 2000.

48. M. M. Yin and J. T. L. Wang, "Application of hidden Markov models to biological data mining: A case study," *Data Mining and Knowledge Discovery: Theory, Tools, and Technology II. Proceedings of SPIE*, B. V. Dasarathy (ed.), Vol. 4057, pages 352–358, SPIE-The International Society for Optical Engineering, USA, 2000.

49. M. M. Yin and J. T. L. Wang, "Effective hidden Markov models for detecting splicing junction sites in DNA sequences," *Information Sciences*, 139(1-2):139–163, 2001.